Parallelised ABox Reasoning and Query Answering with Expressive Description Logics – Technical Report

Andreas Steigmiller⋆ and Birte Glimm

Ulm University, Ulm, Germany, <first name>.<last name>@uni-ulm.de

Abstract. Automated reasoning support is an important aspect of logic-based knowledge representation. The development of specialised procedures and sophisticated optimisation techniques significantly improved the performance even for complex reasoning tasks such as conjunctive query answering. Reasoning and query answering over knowledge bases with a large number of facts and expressive schemata remains, however, challenging.
We propose a novel approach where the reasoning over assertional knowledge is split into small, similarly sized work packages to enable a parallelised processing with tableau algorithms, which are dominantly used for reasoning with more expressive Description Logics. To retain completeness in the presence of expressive schemata, we propose a specifically designed cache that allows for controlling and synchronising the interaction between the constructed partial models. We further report on encouraging performance improvements for the implementation of the techniques in the tableau-based reasoning system Konclude.

1 Introduction

Description Logics (DLs) are a family of logic-based representation formalisms that provide the logical underpinning of the well-known Web Ontology Language (OWL). The knowledge expressed with DLs is typically separated into terminological (aka TBox or schema) and assertional knowledge (aka ABox or facts), where the former describes the relationships between concepts (representing sets of individuals with common characteristics) as well as roles (specifying the relationships between pairs of individuals) and the latter asserts these concepts and roles to concrete individuals of the application domain. Automated reasoning systems derive implicit consequences of the explicitly stated information, which, for example, allows for detecting modelling errors and for enriching queries by additional answers that are implied by the knowledge. Expressive DLs, such as $\mathcal{SROIQ}$ [16], allow for describing the application domain in more detail, but require sophisticated reasoning algorithms and are typically more costly in terms of computational resources. Nevertheless state-of-the-art reasoning systems (e.g., FaCT++ [34], HermiT [12], Konclude [33], Sequoia [8], or Pellet [27]) are usually able to handle real-world ontologies, which often also use expressive language features, due to a large range of developed optimisation techniques (e.g., dependency directed backtracking [35], absorption [17,30], caching [32], pseudo model merging [15], or blocking strategies [24]).

⋆ Funded by the German Research Foundation (Deutsche Forschungsgemeinschaft, DFG) in project number 330492673

The increasing volume of data in many application domains leads, however, also to larger amounts of assertional knowledge. For less expressive schemata (where reasoning is usually deterministic), the interest in ontology-based data access (OBDA) led to several advancements, e.g., via query rewriting [5], materialization techniques [20,21,23], or combined approaches [19,22]. To cope with the reasoning challenges in the presence of an expressive schema, where reasoning has a high worst-case complexity [18], several techniques have been developed, which often complement each other. There are, for example, summarisation [3,9] and abstraction techniques [13], which derive consequences for representative individuals and transfer the results to many other individuals with the same or a similar (syntactical) structure. These techniques do not necessarily work well for all ontologies, may be limited to certain queries or (fragments of) DLs, or require expensive computations, e.g., justifications. Several techniques also reduce reasoning to datalog [1,7,43] since datalog engines are targeted towards data intensive applications. This reduction, however, often leads to some additional overhead and, in some cases, it can be necessary to fall back to a fully-fledged DL reasoner, e.g., for handling non-deterministic features. Other approaches partition the ABox or extract modules out of it [38] such that each part can be processed independently [37]. Moreover, approaches based on big data principles such as map and reduce have been proposed [36]. However, they are typically also limited to specific language features and/or queries and do not work for arbitrary ontologies. In some cases, one can further rewrite the knowledge base such that the reasoner only has to consider a few assertions for answering queries [39], but this rewriting introduces additional non-determinism, which can be problematic for the performance, and presumes that the knowledge base is consistent. Particularly challenging is the support of conjunctive queries with complex concept atoms or with existential variables that may bind to anonymous individuals, which are only implied by the knowledge base. These features typically make it difficult to split the ABox upfront in such a way that queries can correctly be answered without too much data exchange.

Many state-of-the-art reasoners directly integrate techniques that improve ABox reasoning, e.g., (pseudo) model checking [15] or bulk processing with binary retrieval [14]. Most reasoners for expressive DLs are based on (variants of) tableau algorithms, which construct abstractions of models called completion graphs. By caching (parts of) the completion graph from the initial consistency check, subsequent reasoning tasks and queries can be answered more efficiently [26,31]. However, constructing and caching entire completion graphs for knowledge bases with large ABoxes requires significant amounts of (main) memory, which may be more than what is typically available.

There exist several attempts to parallelise tableau-based systems in order to speed up reasoning, but they typically focus on parallelism on a higher level. For example, classification is the reasoning task where we are interested in the subsumption hierarchy of the (atomic) concepts of an ontology and the reasoner usually has to solve many satisfiability/consistency check in order to determine this hierarchy. Hence, one can compute several of these satisfiability/consistency checks in parallel (see, e.g., [25,33,42]), but these techniques are not applicable to all reasoning tasks and do not improve the handling of large ABoxes. Moreover, it has been attempted to directly parallelise the rule applications for tableau algorithms [40,41], but they are restricted to certain DLs and/or

do typically not consider all optimisations that are usually required to handle real-world ontologies. One can even parallelise more specific aspects of tableau algorithms, such as the non-deterministic branching [11], but this also limits the applicability to certain ontologies/test cases. Although other reasoning techniques, such as consequence-based reasoning [4,8], have been extended to very expressive DLs, seem more suitable for parallelisation, and have successfully been implemented in practical systems, it is unclear so far how more sophisticated reasoning task, such as conjunctive query answering, can efficiently be realised with them.

In this paper, we propose to dynamically split the model construction process for tableau algorithms. This allows for (i) handling larger ABoxes since not everything has to be handled at once and for (ii) exploiting parallelisation. The proposed splits lead to similarly sized work packages that can be processed concurrently without direct synchronisation. To ensure that the partial models constructed in parallel are "compatible" with each other, we employ a cache where selected consequences for individuals are stored. For processing new or reprocessing incompatible parts of the knowledge base, we retrieve cached consequences and ensure with appropriate reuse and expansion strategies that the constructed partial models are eventually compatible with the cache, such that it can (asynchronously) be updated. Conjunctive query answering is supported by adapting the expansion criteria and by appropriately splitting the propagation work through the (partial) models for determining query answers.

The paper is organised as follows: Section 2 introduces some preliminaries about DLs and tableau algorithms; Section 3 describes the cache; Section 4 discusses the adaptations for query answering and Section 5 presents implementation details and results of experiments before we conclude in Section 6.

2 Preliminaries

We only give a brief introduction into DLs and reasoning techniques (see, e.g., [2], for more details).

2.1 Description Logics and Conjunctive Queries

The syntax of DLs is defined using a vocabulary consisting of countably infinite pairwise disjoint sets N_C of *atomic concepts*, N_R of *atomic roles*, and N_I of *individuals*. A role is either atomic or an *inverse role* r^-, $r \in N_R$. The syntax and semantics of complex *concepts* and *axioms* are defined in Table 1. Note that we omit the presentation of some features (e.g., datatypes) and restrictions (e.g., number restrictions may not use "complex roles", i.e., roles that occur on the right-hand side of role chains or are implied by such roles) for brevity. A knowledge base/ontology $\mathcal{K}$ is a finite set of axioms. One typically distinguishes terminological axioms in the TBox $\mathcal{T}$ (e.g., $C \sqsubseteq D$) and assertions in the ABox $\mathcal{A}$ (e.g., $C(a)$) of $\mathcal{K}$, i.e., $\mathcal{K} = (\mathcal{T}, \mathcal{A})$. We use $\mathsf{inds}(\mathcal{K})$ to refer to the individuals of $\mathcal{K}$. An *interpretation* $\mathcal{I} = (\Delta^\mathcal{I}, \cdot^\mathcal{I})$ consists of a non-empty *domain* $\Delta^\mathcal{I}$ and an *interpretation function* $\cdot^\mathcal{I}$. We say that $\mathcal{I}$ *satisfies* a general concept inclusion (GCI) $C \sqsubseteq D$, written $\mathcal{I} \models C \sqsubseteq D$, if $C^\mathcal{I} \subseteq D^\mathcal{I}$ (analogously for other axioms and assertions as shown in Table 1). If $\mathcal{I}$ satisfies all axioms of $\mathcal{K}$, $\mathcal{I}$ is a *model* of $\mathcal{K}$ and $\mathcal{K}$ is *consistent/satisfiable* if it has a model.

Table 1: Core features of $\mathcal{SROIQ}$ (#M denotes the cardinality of the set M)

		Syntax	Semantics
Individuals:	individual	a	$a^{\mathcal{I}} \in \Delta^{\mathcal{I}}$
Roles:	atomic role	r	$r^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$
	inverse role	r^{-}	$\{\langle \gamma, \delta \rangle \mid \langle \delta, \gamma \rangle \in r^{\mathcal{I}}\}$
Concepts:	atomic concept	A	$A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$
	nominal	$\{a\}$	$\{a^{\mathcal{I}}\}$
	top	$\top$	$\Delta^{\mathcal{I}}$
	bottom	$\bot$	$\emptyset$
	negation	$\neg C$	$\Delta^{\mathcal{I}} \setminus C^{\mathcal{I}}$
	conjunction	$C \sqcap D$	$C^{\mathcal{I}} \cap D^{\mathcal{I}}$
	disjunction	$C \sqcup D$	$C^{\mathcal{I}} \cup D^{\mathcal{I}}$
	existential restriction	$\exists r.C$	$\{\delta \mid \exists \gamma \in C^{\mathcal{I}} : \langle \delta, \gamma \rangle \in r^{\mathcal{I}}\}$
	universal restriction	$\forall r.C$	$\{\delta \mid \langle \delta, \gamma \rangle \in r^{\mathcal{I}} \rightarrow \gamma \in C^{\mathcal{I}}\}$
	number restriction, $\bowtie \in \{\leqslant, \geqslant\}$	$\bowtie n\, r.C$	$\{\delta \mid \#\{\langle \delta, \gamma \rangle \in r^{\mathcal{I}} \text{ and } \gamma \in C^{\mathcal{I}}\} \bowtie n\}$
Axioms:	general concept inclusion	$C \sqsubseteq D$	$C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$
	role inclusion	$r \sqsubseteq s$	$r^{\mathcal{I}} \subseteq s^{\mathcal{I}}$
	role chains	$r_1 \circ \ldots \circ r_n \sqsubseteq S$	$r_1^{\mathcal{I}} \circ \ldots \circ r_n^{\mathcal{I}} \subseteq S^{\mathcal{I}}$
Assertions:	concept assertion	$C(a)$	$a^{\mathcal{I}} \in C^{\mathcal{I}}$
	role assertion	$r(a, b)$	$\langle a^{\mathcal{I}}, b^{\mathcal{I}} \rangle \in r^{\mathcal{I}}$
	equality assertion	$a \approx b$	$a^{\mathcal{I}} = b^{\mathcal{I}}$
	inequality assertion	$a \not\approx b$	$a^{\mathcal{I}} \neq b^{\mathcal{I}}$

A conjunctive query $Q(X, Y)$ consists of a set of query terms $q_1, \ldots, q_k$, where X denotes the tuples of answer variables, Y the tuple of existential variables (disjoint to X), and each q_i is either a concept term of the form $C(z)$ or a role term of the form $r(z_1, z_2)$ with z, z_1, z_2 variables from X or Y. Note that we omit the tuple of existential variables, e.g., by writing $Q(X)$, if they are clear from the context. A *Boolean query* $Q(\langle\rangle, Y)$, short Q, is a query without answer variables. To simplify the handling with inverse roles, we consider $r(x, y) \in Q$ as equivalent to $r^{-}(y, x) \in Q$. For an interpretation $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ and a total function $\pi : \mathsf{vars}(Q) \mapsto \Delta^{\mathcal{I}}$, we say that π is a *match* for $\mathcal{I}$ and Q if, for every $C(z) \in Q$, $\pi(z) \in C^{\mathcal{I}}$ and, for every $r(z_1, z_2) \in Q$, $\langle \pi(z_1), \pi(z_2) \rangle \in r^{\mathcal{I}}$. We say that an n-ary tuple A of the form $\langle a_1, \ldots, a_n \rangle$ with $a_1, \ldots, a_n$ individuals of $\mathcal{K}$ is an *answer* for $Q(\langle x_1, \ldots, x_n \rangle, Y)$ w.r.t. $\mathcal{K}$ if, for every model $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ of $\mathcal{K}$, there exists a total function π that is a match for $\mathcal{I}$ and Q and for which $\pi(x_i) = a_i^{\mathcal{I}}$ for $1 \leq i \leq n$. If a query $Q(X, Y)$ ($Q(\langle\rangle, Y)$) has an answer (the empty answer $\langle\rangle$) w.r.t. $\mathcal{K}$, then we say that $\mathcal{K}$ *entails* Q and with *query answering* (*query entailment checking*) we refer to the reasoning task that computes all answers (the entailment of the empty answer). W.l.o.g. we use individual names only in nominal concept terms, i.e., not as constants in query terms, and we assume that all variables are connected via role terms.

2.2 Tableau Algorithm

Tableau algorithms are dominantly used for reasoning with more expressive DLs and they decide the consistency of a knowledge base $\mathcal{K}$ by trying to construct an abstraction of a model for $\mathcal{K}$, a so-called "completion graph". A completion graph G is a tuple $(V, E, \mathcal{L}, \dot{\neq})$, where each node $v \in V$ (edge $\langle v, w \rangle \in E$) represents one or more (pairs

of) individuals. Each node v (edge $\langle v, w\rangle$) is labelled with a set of concepts (roles), $\mathcal{L}(v)$ ($\mathcal{L}(\langle v, w\rangle)$), which the individuals represented by v ($\langle v, w\rangle$) are instances of. The relation $\dot{\neq}$ records inequalities between nodes. We call $C \in \mathcal{L}(v)$ ($r \in \mathcal{L}(\langle v, w\rangle)$) a concept (role) fact, for which we also use the notation $C(v)$ ($r(v, w)$). We say a node v is a nominal node if $\{a\} \in \mathcal{L}(v)$ and, otherwise, a blockable node. For $r \in \mathcal{L}(\langle v, w\rangle)$ and $r \sqsubseteq^* s$, with $\sqsubseteq^*$ the reflexive, transitive closure over role inclusions of $\mathcal{K}$ (including their inverses), we call w an *s-successor* of v and v an *s-predecessor* of w. A node w is called an *s-neighbour* of v if w is an s-successor of v or v an *s^--successor* of w. We use *ancestor* and *descendant* as the transitive closure of the predecessor and successor relation, respectively. We say that v_n is an *implied descendant* of v_0 if there is a path $v_0, v_1, \ldots, v_n$ such that v_{i+1} is a successor of v_i for $0 \leq i < n$ and each v_j with $j > 0$ does not represent an individual of $\mathsf{inds}(\mathcal{K})$. Analogously, w is an *implied neighbour* of v if w is a neighbour of v and does not represent an individual of $\mathsf{inds}(\mathcal{K})$, i.e., w is blockable or a new nominal node.

The algorithm works by initialising the graph with one nominal node for each individual in the input knowledge base and adding concepts and roles to the node and edge labels as specified by concept and role assertions. For simplicity, we assume that, for each individual $a \in \mathsf{inds}(\mathcal{K})$, a nominal $\{a\}$ is added to $\mathcal{L}(v_a)$. This allows for easily handling inequality and equality assertions, e.g., by adding $\neg\{b\}$ to $\mathcal{L}(v_a)$ for $a \not\approx b \in \mathcal{A}$ and $\{b\}$ to $\mathcal{L}(v_a)$ for $a \approx b \in \mathcal{A}$. As a convention, we write v_a to refer to the node representing $a \in \mathsf{inds}(\mathcal{K})$, i.e., $\{a\} \in \mathcal{L}(v_a)$. Note that v_a and v_b can refer to the same node if $\{a\}$ and $\{b\}$ are in its label. Complex concepts are then decomposed using expansion rules of the tableau algorithm, where each rule application can add new concepts to node labels and/or new nodes and edges to the completion graph, thereby explicating the structure of a model. The rules are applied until either the graph is fully expanded (no more rules are applicable), in which case the graph can be used to construct a model that is a *witness* to the consistency of $\mathcal{K}$, or an obvious contradiction (called a *clash*) is discovered (e.g., both C and $\neg C$ in a node label), proving that the completion graph does not correspond to a model. $\mathcal{K}$ is consistent if the rules (some of which are non-deterministic) can be applied such that they build a fully expanded, clash-free completion graph. The infinite generation of new nodes is prevented with cycle detection techniques such as *pairwise blocking* [16].

For a concept of the form $\leqslant n\, r.C$ in the label of a node v, the tableau algorithm has to ensure that v has at most r-neighbours with C in their label. This is realised by (non-deterministically) choosing C or $\neg C$ for each r-neighbour and then by merging some neighbours (if there are more than n). If v is a nominal node and there exists a blockable r^--predecessor, i.e., an edge to the nominal node was created from a blockable node, then the tableau algorithm has to fix the number of potential neighbour nodes. This is realised with a fixed number of new nominal nodes that are added as r-neighbours of v, i.e., nodes with new nominals in their label that do not yet occur in the completion graph. In particular, the blockable node could be caused from a cyclic concept such that identical blockable nodes are repeatedly required. Since these blockable node have to be merged into the new nominal nodes, pairwise blocking cannot prevent an expansion that could result in a clash (e.g., if the number of neighbours for a nominal is limited with an atmost restriction, but a cyclic concept requires an infinite path of certain successors with links to the nominal). As one can see, new nominals may only be required if inverse

roles, atmost number restrictions, and nominals are used in certain combinations in the knowledge base. For $\mathcal{SROIQ}$, there exists an upper bound of potentially required new nominals [16], which depends on the maximum length of paths of blockable nodes that can be constructed before they are blocked.

For handling axioms of the form $A \sqsubseteq C$, one typically uses special lazy unfolding rules in the tableau algorithm, which add the concept C to a node label if it contains the atomic concept A. Axioms that cannot directly be handled with these lazy unfolding rules must be internalised, which can be realised by expressing a GCI $C \sqsubseteq D$ by $\top \sqsubseteq \neg C \sqcup D$. Given that $\top$ is satisfied at each node, the disjunction is then also added to all node labels. Since internalisation is quite inefficient, one typically uses a preprocessing step called absorption. Basically, axioms are rewritten into (possibly several) simpler concept inclusion axioms such that lazy unfolding rules in the tableau algorithm can be used and, therefore, internalisation of axioms is often not required, which typically results in less non-determinism. Absorption algorithms based on binary absorption [17] allow for and create axioms of the form $A_1 \sqcap A_2 \sqsubseteq C$, whereby also more complex axioms can be absorbed. This requires the addition of a (binary) unfolding rule that adds C to node labels if A_1 and A_2 are present.

3 Caching Individual Derivations

Since tableau-based reasoning algorithms reduce (most) reasoning tasks to consistency checking, parallelising the completion graph construction has general benefits on the now ubiquitous multi-core processor systems. Partitioning the ABox upfront such that no or little interaction is required between the partitions [37] no longer works for expressive DLs, such as $\mathcal{SROIQ}$, or complex reasoning tasks, such as conjunctive query answering (with complex concepts and/or existential variables). This is, for example, due to implied connections between individuals (e.g., due to nominals) or due to the consideration of new concept expressions at query time. The effect of parallelisation is further hindered by the multitude of optimisations, required to properly deal with real-world ontologies, which often introduce dependencies between rules and (parts of) completion graphs, resulting in the need of data synchronisation. For example, the anywhere blocking optimisation (cycle detection) [24] investigates all previously constructed nodes in the completion graph in order to determine whether a node is blocked. Hence, a parallelisation approach where a completion graph is modified in parallel can be difficult to realise since it could require a lot of synchronisation.

For ontologies with large ABoxes, it seems more suitable to build completion graphs for parts of the ABox separately (by independent threads) and, since independence of the parts cannot be assumed, to align the results afterwards. Such an alignment can, however, be non-trivial on several levels: For example, if different non-deterministic decisions have been made for individuals in overlapping parts or due to technical details of the often complex data structures for completion graphs, e.g., efficient processing queues, caching status of node labels, etc.

Our parallelisation approach focuses on aligning completion graphs for ABox parts and we address the challenges by employing a cache for certain derivations for individuals, which facilitates the alignment process. For this, consistency checking roughly

proceeds as follows: We randomly split the ABox into equally sized parts that are distributed to worker threads. When a thread begins to process one of these ABox parts, it retrieves stored derivations from the cache for (possibly) affected individuals in that part. The thread then tries to construct a fully expanded and clash-free *local* completion graph for the ABox part by reusing cached derivations and/or by expanding the processing to individuals until they are "compatible" with the cache. Compatibility requires that the local completion graph is fully expanded as well as clash-free and that it can be expanded such that it matches the derivations for the remaining individuals in the cache. If it is required to extend the processing to some "neighbouring" individuals for achieving compatibility (e.g., if different non-deterministic decisions are required for the already processed individuals), then also the cached derivations for these individuals are retrieved and considered. If this process succeeds, the cache is updated with the new or changed derivations for the processed individuals.

If compatibility cannot be obtained (e.g., due to expansion limitations that ensure similarly sized work packages), then the cache entries of incompletely handled individuals are marked such that they are considered later separately. For this, a thread loads the data for (some) marked individuals and tries to construct a fully expanded and clash-free completion graph for them until full compatibility is obtained. If clashes occur that depend on reused (non-deterministic) derivations from the cache, then the corresponding individuals can be identified such that their expansion can be prioritized and/or the reuse of their derivations can be avoided. As a result, (in)consistency of the knowledge base can eventually be detected, as soon as all problematic individuals are directly expanded and all relevant non-deterministic decisions are investigated together.

The relatively simple structure of the cache is a suitable basis for establishing a completion graph in parallel as it allows for efficient access and updates. In particular, asynchronous updates (by marking individuals if derivations do not match) avoid that threads must be blocked for read and write access. Moreover, the cache entries can be used as an index to obtain suitable candidates for query answering.

Before describing the different aspects of the approach and the work-flow in more detail, we define a basic version of the cache and how derivations are stored and used.

Definition 1 (Individual Derivations Cache). *Let $\mathcal{K}$ be a knowledge base. We use $\mathsf{fclos}(\mathcal{K})$, $\mathsf{rols}(\mathcal{K})$, and $\mathsf{inds}(\mathcal{K})$ for the sets of concepts, roles, and individuals that can occur in $\mathcal{K}$ or in a completion graph for $\mathcal{K}$. An* individual derivations cache *C is a (partial) mapping of individuals from $\mathsf{inds}(\mathcal{K})$ to* cache entries*, where the cache entry for an individual $a \in \mathsf{inds}(\mathcal{K})$ consists of:*

- *$K^C \subseteq 2^{\mathsf{fclos}(\mathcal{K})}$ and $P^C \subseteq 2^{\mathsf{fclos}(\mathcal{K})}$: the sets of known and possibly instantiated concepts of a, respectively,*
- *$I \subseteq 2^{\mathsf{inds}(\mathcal{K})}$: the individuals that are (indirectly) connected via nominals to a,*
- *$\exists \colon \mathsf{rols}(\mathcal{K}) \to \mathbb{N}_0$: mapping a role r to the number of existentially derived successors for a and r, and*
- *$K^R \colon \mathsf{rols}(\mathcal{K}) \to 2^{\mathsf{inds}(\mathcal{K})}$ and $P^R \colon \mathsf{rols}(\mathcal{K}) \to 2^{\mathsf{inds}(\mathcal{K})}$: mapping a role r to the sets of known and possible neighbours of a and r, respectively.*

We write $K^C(a, C)$, $P^C(a, C)$, $I(a, C)$, $\exists(a, C)$, $K^R(a, C)$, and $P^R(a, C)$ to refer to the individual parts of the cache entry $C(a)$. We write $a \in C$ if C is defined for a.

If the cache is clear from the context, we write, for example, just $K^C(a)$ or $K^R(a)(r)$, where the latter returns the known (deterministically derived) r-neighbours of a. We use subscripts for the mappings as shortcut for different versions of the cache C. For example, $K_i^R(a)(r)$ refers to the known r-neighbours of a in the (individual derivations) cache C_i, i.e., to $K^R(a, C_i)(r)$. Note that we distinguish between known and possible information in the cache, which mostly correspond to the deterministically and non-deterministically derived consequences in completion graphs. The precise extraction and separation of known and possible information from completion graphs is, however, difficult in some cases. In fact, the set $I(a)$ stands for the individuals for which the nominal $\{a\}$ occurred in the label of an implied descendant node, i.e., individuals with a "nominal dependence". This nominal dependence information is typically only collected by propagating one set of (possibly) used nominals to ancestors (until the nodes representing individuals are reached) since it is quite rare that these dependencies are derived deterministically (e.g., due to blocking, where we simply assume that the node can be expanded analogously with the same nominal dependence). Non-deterministically derived facts in completion graphs can usually be identified via branching tags for dependency directed backtracking [2,35]. We use the function detm that takes a set of facts and a completion graph G as input and returns true iff all given facts have been derived deterministically in G. For example, $\mathsf{detm}(\{C(v), r(v, w), v \dot{\neq} w\}, G)$ returns true if C has deterministically been added to $\mathcal{L}(v)$, s deterministically to $\mathcal{L}(\langle v, w\rangle)$ for $s \sqsubseteq^* r \in \mathcal{K}$, and $v \dot{\neq} w$ has been derived deterministically. We omit the set notation for single facts and G if the completion graph is clear from the context.

Let $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ be a knowledge base and $\mathcal{A}_j \subseteq \mathcal{A}$ the processed ABox part. In addition to the usual initialisation of a completion graph $G = (V, E, \mathcal{L}, \dot{\neq})$ for $\mathcal{K} = (\mathcal{T}, \mathcal{A}_j)$, we add $K^C(a)$ to $\mathcal{L}(v_a)$ and r to $\mathcal{L}(\langle v_a, v_b\rangle)$ if $b \in K^R(a)(r)$, for each $v_a, v_b \in V$. If a node $v \in V$ exists with $\{c\} \in \mathcal{L}(v)$ or $\neg\{c\} \in \mathcal{L}(v)$, but $v_c \notin V$, then we add v_c with $\{c\} \in \mathcal{L}(v_c)$ to V and initialise v_c analogously. Once G is extended into a fully expanded and clash-free completion graph, we identify the derivations for cache entries for each individual a with $v_a \in V$ via the following auxiliary functions:

- $\mathsf{cons}^k_{\vdash}(v_a) = \{C \in \mathcal{L}(v_a) \mid \mathsf{detm}(\{C(v_a), \{a\}(v_a)\})\} \cup \{\neg\{b\} \mid \mathsf{detm}(\{\{a\}(v_a), v_b \dot{\neq} v_a, \{b\}(v_b)\})\}$, i.e., the function returns the "subset of the label of v_a" that corresponds to deterministically derived concepts;
- $\mathsf{cons}^p_{\vdash}(v_a)$ corresponds, analogously to $\mathsf{cons}^k_{\vdash}(v_a)$, to the non-deterministically derivable concepts, i.e., $\mathsf{cons}^p_{\vdash}(v_a) = \{C \in \mathcal{L}(v_a) \mid \neg\mathsf{detm}(\{C(v_a), \{a\}(v_a)\})\} \cup \{\neg\{b\} \mid v_b \dot{\neq} v_a \wedge \neg\mathsf{detm}(\{\{a\}(v_a), v_b \dot{\neq} v_a, \{b\}(v_b)\})\}$;
- $\mathsf{ind2O}(v_a) = \{b \mid \{b\} \in \mathcal{L}(v_b) \wedge v$ is an implied descendant of $v_b \wedge v$ is a predecessor of $v_a\}$, i.e., the function returns the set of individuals that are (possibly indirectly) connected to v_a by using nominals;
- $\#\mathsf{exrols}_r(v_a) = \#\{v \mid v$ is an r-neighbour of $v_a \wedge$ there is no $\{o\} \in \mathcal{L}(v)$ with $o \in \mathsf{inds}(\mathcal{K})\}$, i.e., the function returns the number of v_a's implied r-neighbour nodes (i.e., neighbours that do only represent anonymous individuals);
- $\mathsf{neighb}^k_r(v_a) = \{b \mid \mathsf{detm}(\{\{a\}(v_a), r(v_a, v_b), \{b\}(v_b)\})\}$, i.e., the function returns the deterministically derived neighbour individuals for r;
- $\mathsf{neighb}^p_r(v_a) = \{b \mid v_b$ is an r-neighbour of $v_a \wedge \neg\mathsf{detm}(\{\{a\}(v_a), r(v_a, v_b), \{b\}(v_b)\})\}$, i.e., the function identifies the non-deterministically derived neighbours for r;

These auxiliary functions directly deliver the elements for cache entries in order to update the cache. Updating a cache C_i can be seen as creating a new version C_{i+1}, where the processed individuals are mapped to new cache entries with some changed and some new elements. Since deterministically derived consequences remain valid for all possible completion graphs, we update corresponding elements by adding the deterministically derived consequences (e.g., $K^C(a, C_{i+1}) = K^C(a, C_i) \cup \mathsf{cons}^k_{\vdash}(v_a)$), whereas non-deterministically derived consequences may change and, thus, must be replaced (e.g., $P^C(a, C_{i+1}) = \mathsf{cons}^p_{\vdash}(v_a)$). Note that previous versions of the cache can be discarded after the update and, hence, consequences of previously chosen non-deterministic decisions may be lost. The indirectly connected individuals must only be updated for the processed individuals, i.e., $I(a, C_{i+1}) = (I(a, C_i) \setminus \{a_1, \ldots, a_n\}) \cup \mathsf{ind2O}(v_a)$, where $a_1, \ldots, a_n$ are the processed individuals in the constructed completion graph.

A completion graph for $(\mathcal{T}, \mathcal{A}_j)$ is *compatible* with the cache if it can be extended to a fully expanded and clash-free completion graph for $(\mathcal{T}, \mathcal{A}_1 \cup \ldots \cup \mathcal{A}_j)$, where $\mathcal{A}_1 \cup \ldots \cup \mathcal{A}_{j-1}$ are the previously processed (and cached) ABox parts. As argued above, this might require the integration and processing of individuals from the cache during the completion graph expansion. Hence, we define when individuals in the cache *potentially influence* or are *influenced by* the completion graph.

Definition 2 (Cache Influence and Compatibility). *Let $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ be a knowledge base, $G = (V, E, \mathcal{L}, \dot{\neq})$ a completion graph for $(\mathcal{T}, \mathcal{A}_j)$ with $\mathcal{A}_j \subseteq \mathcal{A}$, and $v_a \in V$. For an individual derivations cache C (c.f. Def. 1), an individual $b \in C$ with $v_b \notin V$ is* potentially influenced *by G if*

- D1 $\forall r.C \in \mathcal{L}(v_a), b \in K^R(a)(r) \cup P^R(a)(r)$, *and* $C \notin K^C(b) \cup P^C(b)$;
- D2 $\leqslant n\, r.C \in \mathcal{L}(v_a)$, $b \in K^R(a)(r) \cup P^R(a)(r)$, *and* $\{C, \neg C\} \cap (K^C(b) \cup P^C(b)) = \emptyset$;
- D3 $\leqslant n\, r.C \in \mathcal{L}(v_a), b \in K^R(a)(r) \cup P^R(a)(r)$, *and* $\#[\{d \mid d \in K^R(c)(r) \cup P^R(c)(r)$ *with* $\{c\} \in \mathcal{L}(v_a)\} \cup \mathsf{neighb}^k_r(v_a) \cup \mathsf{neighb}^p_r(v_a)] + \#\mathsf{exrols}_r(v_a) > n$;
- D4 $b \in I(a)$ *and* $C \in \mathcal{L}(v_a), C \notin K^C(a) \cup P^C(a)$ *or* $\leqslant n\, r.C \in \mathcal{L}(v_a)$ *with* $\exists(a)(r) > 0$; *or*
- D5 $\{c\} \in \mathcal{L}(v_a), \{c\} \notin K^C(a) \cup P^C(a)$, $a \in K^R(b)(r) \cup P^R(b)(r)$ *and* $c \notin K^R(b)(r) \cup P^R(b)(r)$ *or* $b \in K^R(a)(s) \cup P^R(a)(s)$ *and* $b \notin K^R(c)(s) \cup P^R(c)(s)$.

An individual $b \in C$ with $v_b \notin V$ is potentially influencing *the completion graph G if*

- G1 $b \in P^R(a)(r)$;
- G2 $b \in K^R(a)(r)$, $C \in P^C(a)$, $C \notin \mathcal{L}(v_a)$;
- G3 $b \in I(a)$, $C \in P^C(a)$, $C \notin \mathcal{L}(v_a)$; *or*
- G4 $\{a\} \in P^C(b)$ *or* $\neg\{a\} \in P^C(b)$;

We say that G is compatible *with a cache C if there is no individual b that is potentially influenced by G or potentially influencing G.*

Roughly speaking, an individual is potentially influenced by a completion graph if integrating it into the completion graph could lead to new consequences being propagated to it. In contrast, an individual potentially influences a completion graph if the integration of it could result in new consequences for the local completion graph (which could, for example, result in unsatisfiability).

Most of the conditions for determining influenced individuals are straightforward. Condition D1 checks whether a universal restriction could propagate a concept to a neighbour node if it is expanded with the neighbouring information (encoded via K^R and P^R) of the cache. Analogously, Condition D2 checks whether the choose-rule of the

tableau algorithm is applicable, i.e., whether it has to be decided whether the neighbour is an instance of C or $\neg C$ for an at-most restriction of the form $\leqslant n\, r.C$. Complementary to Condition D2, D3 determines whether some merging of individual neighbours from the cache could be required for an at-most restriction by counting corresponding neighbours in the completion graph together with the neighbours in the cache. The first part of Condition D4 checks whether some individual is indirectly connected to a via a nominal in an implied descendant of an individual b and whether the label for a differs to the consequences in the cache such that new consequences could be propagated to b (or a descendant of b). The second part handles potential cases where new nominals may have to be introduced and may influence b or descendants of b. Finally, Condition D5 ensures that neighbours are integrated if individuals are newly merged such that their neighbour relations in the cache can be updated.

Conditions D1–D4, could, in principle, just be mirrored for determining influencing individuals. However, the structure of the cache must be kept simple such that updates are efficient and, hence, not all checks can easily and efficiently be supported. For example, every time that the conditions have to be checked for a node in the completion graph, it would be necessary to iterate through the neighbour individuals in the cache and to test all their concepts whether there is a universal restriction that could propagate a new concept to the node. Clearly, there is a lot of room for optimisations, but they seem to require sophisticated data structures. In addition, once the cache entry for an individual is retrieved, the creation and initialisation of a corresponding node in the completion graph is of little effort. As a consequence, we use the relatively simple Conditions G1–G4. More precisely, Condition G1 simply checks whether the connection to a neighbour individual constitutes a possible instance of a role. In addition, if a concept is missing that has been derived previously for an individual, then Condition G2 identifies all neighbouring individuals as potentially influencing. In fact, a neighbour could have a (non-deterministically) derived universal or at-most cardinality restriction that could propagate consequences to the node in the completion graph. Condition G3 analogously checks for a potentially influencing individual b that is indirectly connected via the nominal $\{a\}$ in the label of an implied descendant of b. Last but not least, Condition G4 checks for merges and inequality information caused by non-deterministically derived nominal expressions for other individuals.

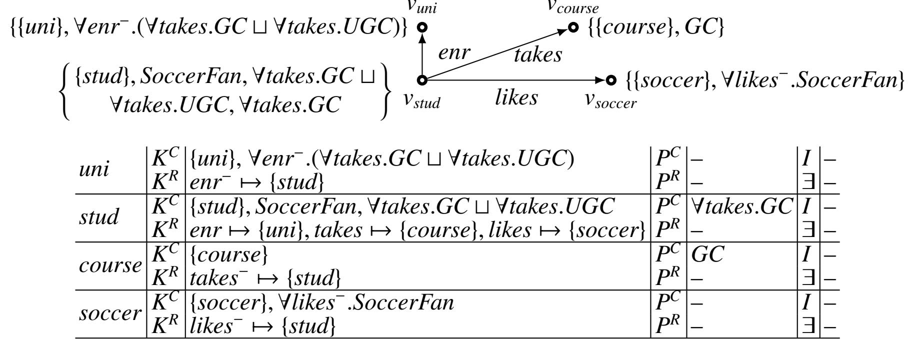
The following example, inspired by the well-known UOBM ontology, illustrates consistency checking with the cache.

Example 1. Suppose an ABox consisting of the two parts:

$\mathcal{A}_1 = \{\ \forall enr^-.(\forall takes.GC \sqcup \forall takes.UGC)(uni),\ \ likes(stud, soccer),\ \ enr(stud, uni),$
$\quad \forall likes^-.SoccerFan(soccer),\ \ takes(stud, course)\},$

$\mathcal{A}_2 = \{\ \exists hc.\exists likes.\{soccer\}(prof),\ \ teaches(prof, course),$
$\quad \forall teaches.\forall takes^-.\neg TennisFan(prof),\ \ likes(prof, soccer)\}.$

We abbreviate *Undergraduate Course* as *UGC*, *Graduate Course* as *GC*, *enrolled in* as *enr*, *has child* as *hc*, and *student* as *stud*. For checking $\mathcal{A}_1$, we initialise a completion graph with nodes and edges that reflect the individuals and assertions in $\mathcal{A}_1$ (cf. upper part of Figure 1). To satisfy the universal restriction for v_{soccer}, which encodes that everyone who likes soccer is a soccer fan, we apply the $\forall$-rule, which propagates

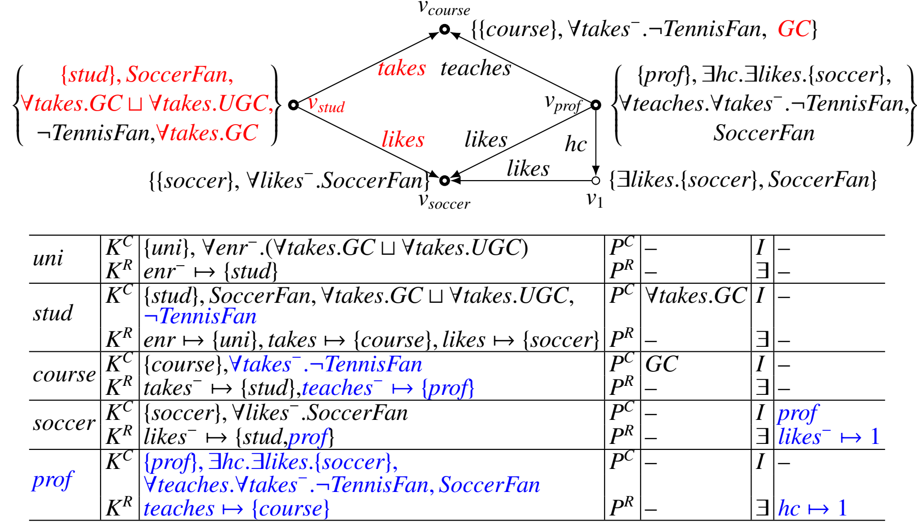


uni	K^C	$\{uni\}, \forall enr^{-}.(\forall takes.GC \sqcup \forall takes.UGC)$	P^C	–	I	–
	K^R	$enr^{-} \mapsto \{stud\}$	P^R	–	$\exists$	–
stud	K^C	$\{stud\}, SoccerFan, \forall takes.GC \sqcup \forall takes.UGC$	P^C	$\forall takes.GC$	I	–
	K^R	$enr \mapsto \{uni\}, takes \mapsto \{course\}, likes \mapsto \{soccer\}$	P^R	–	$\exists$	–
course	K^C	$\{course\}$	P^C	GC	I	–
	K^R	$takes^{-} \mapsto \{stud\}$	P^R	–	$\exists$	–
soccer	K^C	$\{soccer\}, \forall likes^{-}.SoccerFan$	P^C	–	I	–
	K^R	$likes^{-} \mapsto \{stud\}$	P^R	–	$\exists$	–

Fig. 1: Local completion graph (upper part) and entries of the individual derivations cache (lower part) for handling ABox $\mathcal{A}_1$ of Example 1

SoccerFan to v_{stud}. Analogously, the universal restriction for the university v_{uni}, which states that each student enrolled in the university has to take either only graduate or only undergraduate courses, propagates $\forall takes.GC \sqcup \forall takes.UGC$ to v_{stud}. We assume that the disjunct $\forall takes.GC$ is checked first, i.e., it is non-deterministically added to $\mathcal{L}(v_{stud})$. Then the concept GC is propagated to v_{course}. The completion graph for $\mathcal{A}_1$ is now fully expanded and clash-free. We next extract the data for the cache (as shown in the lower part of Figure 1).

The completion graph for $\mathcal{A}_2$ is analogously initialised (cf. upper part of Figure 2). For the concept $\exists hc.\exists likes.\{soccer\} \in \mathcal{L}(v_{prof})$, stating that the professor *prof* has a child that likes soccer, the $\exists$-rule of the tableau algorithm builds a blockable *hc*-successor for v_{prof} with $\exists likes.\{soccer\}$ in its label, for which another successor is created that is merged with v_{soccer} (due to the nominal) leading to the depicted edge to v_{soccer}. Due to the universal restriction $\forall likes^{-}.SoccerFan$ in $\mathcal{L}(v_{soccer})$, *SoccerFan* is propagated to v_1 and to v_{prof}. For the universal restriction $\forall teaches.\forall takes^{-}.\neg TennisFan \in \mathcal{L}(v_{prof})$, stating that all students that take a course taught by him/her must not be a tennis fan, we propagate $\forall takes^{-}.\neg TennisFan$ to v_{course}. Now, there are no more tableau expansion rules applicable to the constructed completion graph, but it is not yet compatible with the cache and we have to integrate (potentially) influenced or influencing individuals. In fact, *course* causes two incompatibilities: On the one hand, Condition D1 identifies *stud* as (potentially) influenced due to $\forall takes^{-}.\neg TennisFan \in \mathcal{L}(v_{course})$ and because *stud* is a $takes^{-}$-neighbour of *course* according to the cache (cf. Figure 1). On the other hand, Condition G2 is satisfied (since $GC \notin \mathcal{L}(v_{course})$ but $GC \in P^C(course)$) and, therefore, the neighbour *stud* listed in $K^R(course)(takes^{-})$ is identified as potentially influencing. We integrate *stud* by creating the node v_{stud}, by adding the concepts $\{stud\}$ and $\forall takes.GC \sqcup \forall takes.UGC$ from the cache to $\mathcal{L}(v_{stud})$, and by creating an edge to v_{course} labelled with *takes* as well as an edge to v_{soccer} labelled with *likes*. Now, the rule application for $\forall takes^{-}.\neg TennisFan \in \mathcal{L}(course)$ propagates $\neg TennisFan$ to v_{stud}. In addition, by reprocessing the disjunction $\forall takes.GC \sqcup \forall takes.UGC$ for v_{stud}, we obtain $GC \in \mathcal{L}(v_{course})$ if the same disjunct is chosen. As a result, the completion graph is fully expanded and clash-free w.r.t. $\mathcal{A}_2$ and it is compatible with the cache. Hence, the cache



uni	K^C	{*uni*}, $\forall enr^-.(\forall takes.GC \sqcup \forall takes.UGC)$	P^C	–	I	–
	K^R	$enr^- \mapsto \{stud\}$	P^R	–	$\exists$	–
stud	K^C	{*stud*}, *SoccerFan*, $\forall takes.GC \sqcup \forall takes.UGC$, $\neg TennisFan$	P^C	$\forall takes.GC$	I	–
	K^R	$enr \mapsto \{uni\}$, $takes \mapsto \{course\}$, $likes \mapsto \{soccer\}$	P^R	–	$\exists$	–
course	K^C	{*course*}, $\forall takes^-.\neg TennisFan$	P^C	GC	I	–
	K^R	$takes^- \mapsto \{stud\}$, $teaches^- \mapsto \{prof\}$	P^R	–	$\exists$	–
soccer	K^C	{*soccer*}, $\forall likes^-.SoccerFan$	P^C	–	I	*prof*
	K^R	$likes^- \mapsto \{stud, prof\}$	P^R	–	$\exists$	$likes^- \mapsto 1$
prof	K^C	{*prof*}, $\exists hc.\exists likes.\{soccer\}$, $\forall teaches.\forall takes^-.\neg TennisFan$, *SoccerFan*	P^C	–	I	–
	K^R	$teaches \mapsto \{course\}$	P^R	–	$\exists$	$hc \mapsto 1$

Fig. 2: Local completion graph (upper part, expansions from cache due to incompatibilities in red) and entries of the individual derivations cache (lower part, changes in blue) for handling ABox $\mathcal{A}_2$ of Example 1

can be updated resulting in the entries depicted in the lower part of Figure 2. Note that only v_{uni} has not been integrated in the completion graph for $\mathcal{A}_2$, but there is usually a bigger gain for larger ABoxes.

It is clear that the processing of the local completion graph should directly be extended to (potentially) influenced individuals from the cache. The new consequences that are likely propagated to them, have to be considered in the model construction process and might lead to changed entries in a subsequent cache update. Hence, influenced individuals should be integrated in the completion graph with a high priority by adding their known, deterministically derived consequences from the cache. In contrast, the expansion to (potentially) influencing individuals should usually be delayed since they can require many additional expansions (until the cause of the non-deterministically derived consequences is integrated in the local completion graph) and there are other, more efficient ways to get and integrate these consequences. For example, it might be possible that the missing consequences are derived locally in other ways and we can also (non-deterministically) reuse consequences from the cache. Nevertheless, even just expanding all (potentially) influenced individuals can lead to very large completion graphs, e.g., if new universal restrictions propagate concepts over complex (transitive) roles. This can obviously also be problematic for parallelisation since one thread might be required to access and update all cache entries. In the following, we present several extensions of the presented approach in order to address these issues and make it more suitable for parallelisation.

3.1 Parallelised Work-Flow

To make the described model construction process with the individual derivations cache well-suited for parallelisation, some minor adjustments and extensions are useful. As mentioned, our goal is it to handle multiple parts of the ABox in parallel by building local completion graphs that are expanded until compatibility with the cache is achieved. Since the criteria for this depend on non-deterministically derived consequences in cache entries (which can change through an update), the access of such entries must be synchronised in order to know or avoid that other threads modify them while they are used by one thread in the completion graph for guaranteeing compatibility. For example, if the cache has $P^C(c) = \{A, \neg B\}$, $K^R(c)(r^-) = \{a, b\}$, $K^R(a)(r) = K^R(b)(r) = \{c\}$, and we process, in parallel, the ABoxes $\mathcal{A}_1 = \{(\forall r.A \sqcup \forall r.B)(a)\}$ and $A_2 = \{(\forall r.A \sqcup \forall r.B)(b)\}$, then the thread that handles $\mathcal{A}_1$, say T_1, initialises a completion graph and may chose the disjunct $\forall r.A$ for the node representing a. Since the cache contains A as a possible/non-deterministic consequence for the individual c, Condition D1 is not satisfied although c is an r-neighbour of a. Hence, c is not influenced by the completion graph that is constructed by T_1 and, therefore, it is not required to extend the processing to c. Since the constructed completion graph can be considered fully expanded and clash-free (w.r.t. $\mathcal{A}_1$), T_1 prepares to update the cache. Let us assume that, in the meantime, T_2 constructs a completion graph for $\mathcal{A}_2$ and chooses the disjunct $\forall r.B$ for the node representing b. Since c is influenced by this completion graph, T_2 extends the processing to c and now derives B for the node v_c representing c. Since some cached consequences are missing for v_c, the remaining neighbours of c must also be integrated (due to Condition G2), i.e., a node for a is initialised and this completion graph can now also be considered fully expanded and clash-free (regarding $\mathcal{A}_2$). Now, if T_2 updates the cache first, then we have $P^C(c) = \{B\}$, $P^C(b) = \{\forall r.B\}$. If the changes for c are not recognised by T_1, then it would just further update the cache with $P^C(a) = \{\forall r.A\}$, which obviously leaves c in an inconsistent/problematic state, especially if we have an axiom of the form $A \sqsubseteq \neg B$ in the knowledge base. Consequently, also any read access must be logged and appropriately considered in cache updates.

It can obviously also be problematic if several threads update different parts of the cache concurrently since this could also leave the cache in an inconsistent state. Suppose, for example, that the threads T_1 and T_2 finished the construction of completion graphs containing nodes for the individuals a and b and that T_1 begins modifying the cache entry for a while thread T_2 first modifies the entry for b. After that, they could both update the entries for the other individuals simultaneously, i.e, T_1 could modify the entry for b and T_2 for a, which could also result in an inconsistent state since some consequences were obtained from the completion graph constructed by T_1 and some from the one constructed by T_2. Since the updates can compromise many individuals, it also does not seem to be a good idea to lock the entire cache or all involved individuals for each update (where the latter also bears the danger of deadlocks if realised naively). However, if we want to avoid general locks, then we need to handle cases where updates cannot directly be integrated in case cache entries got changed after they were retrieved by the updating thread. Although one could reconstruct the local completion graph with the new information in the cache until an update is possible, this can easily

be problematic if the completion graphs compromise (many) overlapping individuals, i.e., individuals that have to be considered in many or all local completion graphs.

To address these issues, we allow some "inconsistency" for cache entries for individuals, but we mark these entries such that we can reprocess them later. In addition, we annotate each entry with an update id, which is used in updates to check for modifications by other threads. Last but not least, we propose to (asynchronously) "outsource" the cache updates, i.e., threads merely extract updates for the constructed completion graphs and send these in form of "messages" to a designated thread that integrates these updates. While these updates are being integrated, the other threads can continue to construct completion graphs for new parts of the ABox. This makes locking individuals or cache entries unnecessary and further facilitates more sophisticated update mechanisms (see Section 3.4). In the following, we define the modifications for the cache in more detail since it is also used by other extensions.

Definition 3 (Cache Inconsistency and Update Ids). *Let $\mathcal{K}$ be a knowledge base and* $\mathsf{inds}(\mathcal{K})$ *the individuals that occur in $\mathcal{K}$. We call C an* (extended) individual derivations cache*, if the cache entries (as specified in Definition 1) are annotated by an* update id $u \in \mathbf{N}_0$. *An individual a or its cache entry has an* inconsistent state *if* $\bot \in P^C(a)$.

While building a (local) completion graph, we collect all accessed cache entries in a separate, initially empty "update cache" $\mathcal{U}$. For example, if we check whether $a \in K^R(b)(r)$, then we atomically copy the cache entry for a to the update cache $\mathcal{U}$. Furthermore, we ensure that all cache entry accesses for the same individual stem from the same update, e.g., by first trying to retrieve the entry from the update cache. If the local completion graph is fully expanded and clash-free, then we extract the update data with the defined auxiliary functions and we replace the corresponding elements in the entries for the individuals in the update cache. For newly handled individuals, we use 0 for the update id in the newly extracted cache entries. Now, the actual cache can basically be updated by "merging" the update cache into it, which can be realised by a separate thread. If the update id for a cache entry is the same, then (most) possible information is simply replaced and, otherwise, the possible information is merged (known consequences are always merged) and the inconsistency status flag is set.

Definition 4 (Cache Update). *Let $\mathcal{K}$ be a knowledge base and C_j and $\mathcal{U}$ (extended) individual derivations caches for $\mathcal{K}$. The* update of C_j with $\mathcal{U}$ *is an extended cache C_{j+1} that contains, for each individual $a \in C_j \setminus \mathcal{U}$, $C_j(a)$ with the same update id as annotated with $C_j(a)$; for each individual $a \in \mathcal{U} \setminus C_j$, $\mathcal{U}(a)$ with the update id set to* 1*; for each individual $a \in \mathcal{U} \cap C_j$ with update id u for $\mathcal{U}(a)$ and u_j for $C_j(a)$, a cache entry such that*

$$K^C(a, C_{j+1}) = K^C(a, C_j) \cup K^C(a, \mathcal{U})$$

$$P^C(a, C_{j+1}) = \begin{cases} P^C(a, \mathcal{U}) & \text{if } u = u_j \\ P^C(a, C_j) \cup P^C(a, \mathcal{U}) \cup \{\bot\} & \text{otherwise} \end{cases}$$

$$I(a, C_{j+1}) = (I(a, C_j) \setminus \{b \mid b \in \mathcal{U}\}) \cup I(a, \mathcal{U})$$

and for each role r

$$\exists(a, C_{j+1})(r) = \begin{cases} \exists(a, \mathcal{U})(r) & \textit{if } u = u_j \\ \max(\exists(a, C_j)(r), \exists(a, \mathcal{U})(r)) & \textit{otherwise} \end{cases}$$
$$K^R(a, C_{j+1})(r) = K^R(a, C_j)(r) \cup K^R(a, \mathcal{U})(r)$$
$$P^R(a, C_{j+1})(r) = \begin{cases} P^R(a, \mathcal{U})(r) & \textit{if } u = u_j \\ P^R(a, C_j)(r) \cup P^R(a, \mathcal{U})(r) & \textit{otherwise} \end{cases}$$

annotated with the update id $u_j + 1$ *if the cache entries (modulo their update id)* $C_{j+1}(a)$ *and* $\mathcal{U}(a)$ *differ and with* u_j *otherwise.*

This asynchronous update procedure requires a few more adaptations to the processing work-flow. It is now required to (repeatedly) retrieve inconsistent entries from the cache and reschedule the corresponding individuals for processing. Since most ontologies are consistent, it makes sense to do this after all ABox parts are processed. Otherwise it may be required to reprocess the same individuals multiple times (if more and more restrictions from the ABox are being added). To enable parallelisation, one can split the individuals with inconsistent entries in multiple work packaged that can be processed by different threads. If the next round of inconsistent entries is retrieved from the cache (while some threads still try to construct completion graphs), then it has to be ensured that individuals are not rescheduled while the update from the previous round has not been integrated. If the inconsistency of cache entries cannot be resolved even after several reprocessing attempts (e.g., since they have to be used differently in concurrent completion graph construction processes), then we increase the limit of individuals that are scheduled to be reprocessed by one thread. In the worst case, one thread finally reprocesses all individuals with inconsistent cache entries at the same time and either detects inconsistency of the ontology or extracts an update that resolves all inconsistent entries.

Note that the update procedure as a whole does not have to be atomic, i.e., we can simply maintain one cache, where only the entries are (atomically) exchanged through the integration of updates. Consequently, other threads may access some new and some old entries for constructing the next completion graphs, but since the access of entries is recoded through the update id, inconsistent states can be recognised in subsequent updates.

It is worth discussing that the update procedure can cause some inefficiency for ontologies that intensively imply nominals for anonymous individuals due to the "nominal dependency" encoded with I. In fact, if a nominal $\{a\}$ occurs in the label of an implied descendant of an individual b, then the referenced individual a must be integrated and b must be added to $I(a)$. If the nominal $\{a\}$ is often implied for blockable nodes, then the corresponding cache entry for a is regularly updated and, due to parallelisation, it is likely that the cache entry for a becomes inconsistent, e.g., due to non-matching update ids from accessing different (versions of) cache entries for a. As a consequence, it would be necessary to eventually reprocess the individual a in a completion graph and

due to Condition G3 it would be necessary to also integrate all individuals that are dependent on the nominal $\{a\}$, i.e., all individuals in $I(a)$, which could be quite many. This can be addressed by separately managing, for each cache entry, an "integration id" that encodes, based on the update id, the last required integration of indirectly connected individuals into a local completion graph. As long as only new indirectly connected individuals are added to a cache entry and the integration id stays the same (i.e., it is not required to integrate the indirectly connected individuals into local completion graphs), we can leave the cache entry in a consistent state even if the update ids do not match.

More precisely, if u is the update id that is associated with the cache entry $\mathcal{U}(a)$ in the update cache $\mathcal{U}$, then we annotate $\mathcal{U}(a)$ with an integration id i set to u if it was necessary to integrate the indirectly connected individuals in the constructed completion graph (e.g., due to Condition G3 for newly derived concepts for the individual a). If a is a newly handled individual or it was not necessary to integrate indirectly connected individuals, then the integration id i for $\mathcal{U}(a)$ is set to 0. When the update is integrated into a new cache 0version C_{j+1}, then the cache entry for a is interpreted inconsistent if the integration id changed, i.e., $i > 0$ and $i \neq i_j$ with i_j representing the integration id of the current entry in the cache, i.e., $C_j(a)$. If there was no entry for a in C_j, then the integration id is set to 1, i.e., $i_{j+1} = 1$ (analogous to the update id). If the integration of indirectly connected individuals was not necessary for a in the constructed completion graph that led to the update, i.e., $i = 0$, then the previous value of the integration id is kept, i.e., $i_{j+1} = i_j$. In all cases, the new indirectly connected individuals are added ($I_{j+1}(a) = I_j(a) \cup I_{\mathcal{U}}(a)$). Even if the update ids do not match (i.e., $u \neq u_j$), the cache entry can now be considered consistent if the update does not change the integration id and all parts, except the indirectly connected individuals, stay the same.

It is further possible to avoid interpreting a cache entry as inconsistent in some other scenarios. Most notably, as long as only deterministic consequences are derived, the concepts are the same, and there are no (possibly problematic) at-most restrictions, then we can simply merge the entries without marking them inconsistent. Note that typically also other language features are associated with $\mathcal{SROIQ}$ that may need more intensive checks whether the entries possibly have to be considered inconsistent, e.g., disjoint roles. Nonetheless, this is very handy for a "preprocessing step", where a simple but efficient procedure is used to derive (most) deterministic consequences. Of course, if such a procedure is possibly incomplete, then the corresponding cache entry for an individual has to be considered inconsistent, but it nevertheless allows us to avoid the processing of many "simple parts" of typical knowledge bases with fully-fledged procedures, such as tableau algorithms.

3.2 (Non-deterministic) Derivations Reuse

The presented influence criteria at the beginning of this section ensure that all individuals are expanded that may contribute some non-deterministic consequences to the currently handled part of the ABox. This is important when many or all non-deterministic alternatives have to be investigated to check whether there exists a satisfiable solution of the imposed restrictions or whether the knowledge base is inconsistent. At the same time, this can easily enforce the reprocessing of many individuals to get the required consequences for ensuring compatibility with the cache. As an example, let us assume

that we have the individuals $a_0, \dots, a_m$ in the knowledge base, which form a chain of instances w.r.t. the role r, i.e., we have $r(a_i, a_{i+1})$ for $0 \leq i < m$, and we have a concept assertion of the form $(A \sqcup B)(a_0)$ with $A \sqsubseteq \forall r.A$ in the knowledge base. Let us further assume that we first construct a local completion graph for the ABox part $\{(A \sqcup B)(a_0), r(a_0, a_1)\}$, where we (non-deterministically) derive the facts $A(v_{a_0})$ and $\forall r.A(v_{a_0})$ for the node v_{a_0} representing the individual a_0, which are then stored in the cache. If we now continue with the construction of a completion graph for $\{r(a_1, a_2)\}$, then a_0 must be expanded again (due to Condition G2) in order to rederive $\forall r.A$ for a_0. In the worst-case, we process the role assertions in the chain in such an order that, for each completion graph construction, all previously handled individuals must be expanded again to rederive $\forall r.A$ for these previous individuals. This inefficiency can be addressed by reusing non-deterministically derived consequences from cache entries, which is, in principle, quite straightforward. In fact, for a node v_a, we simply add $P^C(a)$ to $\mathcal{L}(v_a)$ and we add the role r to $\mathcal{L}(\langle v_a, v_b \rangle)$ if $b \in P^R(a)(r)$. If a new individual, say c, is later integrated into the local completion graph and it is a possible r-neighbour of the individual a (i.e., $c \in P^R(a)(r)$) for which consequences are being reused, then the label of the edge between these nodes (i.e., $\mathcal{L}(\langle v_c, v_a \rangle)$) must also be extended by the possible role instantiation (i.e., r) from the cache entry. It is worth noting that Conditions G1–G4 do not have to be checked for nodes that represent individuals for which consequences are being reused since the expansion of their potentially influencing individuals would not lead to new consequences (at least it would be possible to expand the potentially influencing individuals with the same consequences as stored in the cache). Hence, the reuse of non-deterministically derived consequences from the cache typically allows us to establish compatibility much faster and with less individuals that are to be integrated.

We have to be careful, however, how and when to reuse these consequences. By prioritising the reuse of non-deterministic consequences (e.g., by adding $P^C(a)$ as soon as we create a node for a), the thread would directly try to build a local completion graph that is compatible with the cache without expanding many nodes. This can be inefficient if the restrictions in the local completion graph require a lot of processing (e.g., due to a hard combinatorial problem encoded with these restrictions) and lead to clashes with the reused consequences. If the clashed facts are analysed and (dependency directed) backtracking is applied, i.e., the relevant non-deterministic decisions are identified, then it can be learned that reusing the consequences of corresponding cache entries should be avoided. As a consequence, the local completion graph would further be expanded and the thread may try to (non-deterministically) reuse consequences of other individuals. Due to the backtracking, it would be required to reprocess the restrictions of the local completion graph and this may have to be repeated many times until it has been learned for all possibly influencing individuals that their (non-deterministic) consequences are causing clashes for the newly handled part of the ABox. In contrast, if the reuse of (non-deterministic) consequences is delayed (i.e., the application of non-deterministic rules for already present concepts in the local completion graphs is prioritised), then it would be required to expand so far as the (non-deterministic) consequences of influencing individuals do not disturb the found solution for the already present restrictions in the local completion graph. Consequently, a lot of expansion to potentially influencing individuals could be required again. A compromise between both strategies clearly

makes sense, where we first try to find a local solution for the reused consequences (such that only a few expansion are required) and, if this fails, then our priority is to find a solution of the local restrictions and reuse only those consequences that do not contradict with the found solution.

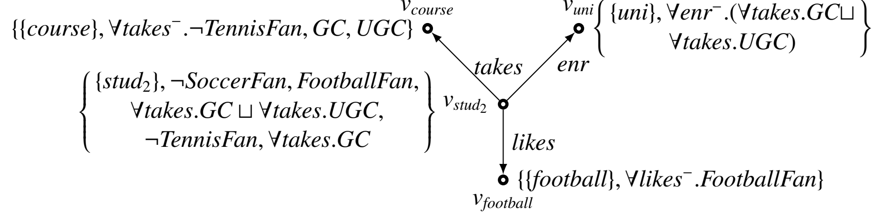
To implement the idea of the proposed compromise, we can create two branches of the completion graph (before the first reuse and the first non-deterministic decision), where the first branch corresponds to the case where the reuse of non-deterministic consequences is prioritised and which is processed first. If this first "fixed-reuse" branch fails (i.e., all non-deterministic decisions only results in clashes), then the second branch is attempted, where one tries to reuse non-deterministic consequences once and, if this leads to a clash, one simply further expands (and may try to reuse only consequences of other individuals). For this second "prioritised-reuse" branch, the expansion and reuse is delayed until all restrictions and facts in the existing completion graph are fully processed, i.e., the expansion and the reuse are processed with a low priority. Moreover, if the "fixed-reuse" branch only leads to clashes, we identify the involved individuals for which consequences have been reused and mark these individuals or their cache entries problematic. Problematic individuals or their problematic cache entries can then be used to handle them more carefully, e.g., by expanding to them first (since it is "more likely" that they cause problems) and/or avoiding the reuse of their consequences (since they "probably" lead to clashes). Note that the problematic individuals/cache entries should be reported to the cache such that it can reorder neighbours with the next update. For this, we can extend the cache to keep an order among the recorded neighbouring individuals (e.g., for K^R and P^R). This allows for prioritising problematic individuals such that they are expanded first. Also note that it is a good idea to identify all involved individuals of clashes (not only those for which consequences are reused) such that the causes of clashes can more easily be found. This can be realised with a precise dependency tracking between facts in completion graphs [32], where the causes of facts are stored, i.e., for each fact it is stored which other facts are its cause. If a clash occurs, then one simply tracks back the involved facts to their causes and identifies all individuals to which these facts belong.

For the update extraction, we have to consider that possible neighbour individuals are not identified by the neighb function if these individuals are not integrated into the local completion graph and that the cache usually just "replaces" the possible/non-deterministic information. Hence, if we reuse the possible consequences of an individual a, which has a possible r-neighbour b w.r.t. the cache that may not be integrated into the local completion graph due to the non-deterministic reuse, then we have to extend the possible neighbours extracted with neighb by b for the role r in order to get a correct cache update.

Example 2 (Example 1 continued). Let us assume that we have now, in addition to part 1 and 2 of Example 1, the ABox part $\mathcal{A}_3$ consisting of the axioms

$$\forall likes^{-}.FootballFan(football), \quad \neg SoccerFan(stud_2), \quad takes(stud_2, course), \quad enr(stud_2, uni), \quad \text{and} \quad likes(stud_2, football).$$

Some of the assertions (e.g., around the individual *football*) mainly become relevant in the continuation of this example (Example 3), whereas here, we first focus on the



uni	K^C	$\{uni\}, \forall enr^{-}.(\forall takes.GC \sqcup \forall takes.UGC)$	P^C	–	I	–
	K^R	$enr^{-} \mapsto \{stud, stud_2\}$	P^R	–	$\exists$	–
stud	K^C	$\{stud\}, SoccerFan, \forall takes.GC \sqcup \forall takes.UGC, \neg TennisFan$	P^C	$\forall takes.GC$	I	–
	K^R	$enr \mapsto \{uni\}, takes \mapsto \{course\}, likes \mapsto \{soccer\}$	P^R	–	$\exists$	–
course	K^C	$\{course\}, \forall takes^{-}.\neg TennisFan$	P^C	GC, UGC	I	–
	K^R	$takes^{-} \mapsto \{stud, stud_2\}, teaches^{-} \mapsto \{prof\}$	P^R	–	$\exists$	–
soccer	K^C	$\{soccer\}, \forall likes^{-}.SoccerFan$	P^C	–	I	*prof*
	K^R	$likes^{-} \mapsto \{stud, prof\}$	P^R	–	$\exists$	$likes^{-} \mapsto 1$
prof	K^C	$\{prof\}, \exists hc.\exists likes.\{soccer\}, \forall teaches.\forall takes^{-}.\neg TennisFan, SoccerFan$	P^C	–	I	–
	K^R	$teaches \mapsto \{course\}$	P^R	–	$\exists$	$hc \mapsto 1$
$stud_2$	K^C	$\{stud_2\}, \neg SoccerFan, FootballFan, \forall takes.GC \sqcup \forall takes.UGC, \neg TennisFan$	P^C	$\forall takes.UGC$	I	–
	K^R	$enr \mapsto \{uni\}, takes \mapsto \{course\}, likes \mapsto \{football\}$	P^R	–	$\exists$	–
football	K^C	$\{football\}, \forall likes^{-}.FootballFan$	P^C	–	I	–
	K^R	$likes^{-} \mapsto \{stud_2\}$	P^R	–	$\exists$	–

Fig. 3: Local completion graph with non-deterministically reused consequences (upper part) and the obtained individual derivations cache (lower part, changes in blue) for handling ABox $\mathcal{A}_3$ of Example 2

aspects of reusing cached non-deterministic derivations. As usual, we initialise a local completion graph by creating nodes for the individuals $stud_2, course, uni$, and *football* that occur in $\mathcal{A}_3$, for which then the concepts and roles from assertions and from deterministic/known elements of cache entries are added to node and edge labels (cf. upper part of Figure 3). Since the cache entry for *course* is consistent (i.e., $\bot \notin P^C(course)$), we can further reuse the non-deterministically derived/possible consequences of *course*, i.e., we create two branches and add $P^C(course)$ to $\mathcal{L}(v_{course})$ for the first "fixed-reuse" branch. If these possible consequences were not reused, then it would again be necessary to expand to the individual *stud* (due to Condition G2) such that the disjunction that caused the missing consequence can be processed. Since the student $stud_2$ is enrolled in the university *uni*, the disjunction $\forall takes.GC \sqcup \forall takes.UGC$ is also propagated to v_{stud_2} and must be processed there. Let us assume that the tableau algorithm adds here the second disjunct such that *UGC* is propagated to v_{course}, which is, however, not problematic since a course can be a graduate as well as an undergraduate course (in our model). Hence, the fixed-reuse branch is satisfiable and we can update the cache with the extracted data (cf. lower part of Figure 3).

If the knowledge base were to contain the axiom $UGC \sqsubseteq \neg GC$ and the assertions $takes(stud_2, course_2)$ as well as $\neg GC(course_2)$ were in $\mathcal{A}_3$, then the fixed-reuse

branch would not be satisfiable. In fact, if the tableau algorithm was to try the disjunct $\forall takes.GC$ for $stud_2$, then we would obtain a clash for $course_2$ (since there would be GC and $\neg GC$), and, for the disjunct $\forall takes.UGC$, we would obtain a clash for *course* (since we would have GC and UGC in the label of $course_2$, which is not permitted by the axiom $UGC \sqsubseteq \neg GC$). By tracing back the causes of the clashes, we would identify the individuals $stud_2$, $course_2$, and *course* as problematic, i.e., we would prioritise their expansion/processing in future completion graphs such that clashes could be discovered faster. Since the fixed-reuse branch would be unsatisfiable, we would try to build the completion graph with the prioritised-reuse branch, where first the local completion graph is entirely processed (i.e., the disjunct $\forall takes.UGC$ would be added) and then we would expand the processing to the individual *stud* (since GC would be missing for *course*). Again, the algorithm would try to also reuse possible consequences for *stud*, which would propagate GC to v_{course} and, thus, would again result in a clash. For the prioritised-reuse branch, we would then backtrack and avoid the non-deterministic reuse also for *stud* such that the disjunction could be satisfied by adding the disjunct $\forall takes.UGC$, which would result in a fully expanded and clash-free completion graph for $\mathcal{A}_3$ (extended by $takes(stud_2, course_2)$ and $\neg GC(course_2)$).

3.3 Restricting Processing

If nodes for individuals have the same consequences as the cache entries of these individuals (and the cache entries are consistent), then it can be possible to restrict the processing of these nodes and their descendants. In fact, it could already be clear that these nodes can be expanded as before such that they yield the same consequences as stored in the cache.

Definition 5 (Cached Expansion Blocked). *Let $G = (V, E, \mathcal{L}, \dot{\neq})$ be a completion graph for a knowledge base $\mathcal{K}$ and C the individual derivations cache. A node $v_a \in V$ is* cached expansion blocked *if the cache entry for a is consistent and*

- B1 $\mathcal{L}(v_a) = K^C(a) \cup P^C(a)$,
- B2 *there is no $v_b \in V$ such that $b \in P^R(a)(r)$ and $r \notin \mathcal{L}(\langle v_a, v_b \rangle)$*
- B3 *for each $\leqslant n\, r.C \in \mathcal{L}(v_a)$, $\exists(a)(r) + \#\mathsf{exrols}_r(v_a) > 0$, and $\#\{K^R(a)(r) \cup P^R(a)(r) \cup \mathsf{neighb}_r^P(v_a)\} + \exists(a)(r) + \#\mathsf{exrols}_r(v_a) \leq n$;*
- B4 *a is not potentially influenced due to Condition* D4 *and is not potentially influencing due to Condition* G3.

As long as a node is cached expansion blocked, then it is not necessary to apply generating tableau rules for the concepts in its label, i.e., we do not have to generate new successors with the $\exists$- or with the $\geqslant$-rule. The $\forall$-rule for universal restrictions must, however, still be applied to propagate the qualified concept to new neighbours. As long as the node is cached expansion blocked, the $\leqslant$-rule is not applicable for an at-most restriction of the form $\leqslant n\, r.C$ since Condition B3 ensures that the number of appropriate successors is less than n. Note that even nodes for individuals that are used as nominals can be cached expansion blocked as long as Conditions D4 and G3 are not satisfied, i.e., they only propagate the same consequences to new blockable predecessors.

The processing restriction is particularly useful in combination with a (non-deterministic) reuse of consequences from the cache. In fact, by reusing (non-deterministic) consequences, the nodes are often directly cache expansion blocked and the tableau algorithm can focus on applying rules for the newly handled part of the ABox. Analogously to the update extraction with reused consequences, we may have to integrate some data from the used cache entry for the update. In particular, the #exrols function may not return the correct number of potentially existing successors, but by adding the values from the mapping $\exists$ of the corresponding cache entry, we can ensure a correct upper bound.

3.4 Propagation Expansion Cut

The model construction work-flow with the individual derivations cache allows for splitting the ABox in many small parts that are well-suited for a parallelised processing. However, it is not guaranteed that each work package (consisting of a small part of the ABox or some individuals with inconsistent cache entries) requires a similar amount of work, which can be problematic for parallelisation. Clearly, it is generally difficult to avoid any imbalance since there could be a hard combinatorial problem encoded for a few individuals, whereas the remaining individuals only imply some trivial consequences. It is, nevertheless, desirable to have a limit on the number of individuals to which the local completion graph can be expanded such that it is less likely to have a work imbalance, where one thread has to do much more work than the others. For example, a universal restriction of the form $\forall t.C$ for a transitive role t is typically "unfolded" to $\forall t.F$ and $F \sqsubseteq C \sqcap \forall t.F$ (with F a fresh atomic concept). If the universal restriction is derived for an individual, then the concepts F and C must be propagated to all nodes in the transitive closure for this individual w.r.t. t, which can obviously be many or even all individuals of the knowledge base. Hence, if one thread derives such a universal restriction for an individual with a large transitive closure, then all individuals in the transitive closure are step-by-step identified as "influenced" (due to Condition D1) and are expanded in the same completion graph, which eventually leads to one large update. Other causes that could require an expansion to many individuals are, for example, propagations over highly connected individuals or merges with these. Besides the fact that a required expansion to many individuals could lead to an imbalanced work distribution, it can also make it much more problematic to process such ontologies with ordinary/commodity machines that only provide a limited amount of main memory. In fact, the construction of large completion graphs that involve many individuals usually requires a significant amount of memory since many processing details must be stored (e.g., processing queues, branching tags for dependency directed backtracking, caching status, etc.). In the worst case scenario, each thread could be required to expand the processing to all individuals at the same time, i.e., the parallelisation could even multiply the memory requirements by the number of parallel working threads. In contrast, the data of the individual derivations cache is significantly reduced and has a simpler structure such that it can more easily be compressed or even outsourced into a database. Hence, by introducing ways for limiting the required expansion, we not only ensure a more balanced parallelisation, but also reduce the memory requirements for the processing of (large) ontologies.

Since the (extended) individual derivations cache can already mark cache entries as inconsistent such that they have to be reprocessed later (see Section 3.1), we could simply stop the processing of individuals if we reach an expansion limit and mark the cache entries for these individuals in the update cache inconsistent. Unfortunately, this does only solve the issue to some extend. In fact, for highly connected individuals, i.e., individuals with many neighbours, it is either required to expand to all neighbours such that their cache entries can be marked inconsistent of we can directly mark the cache entry of the highly connected individual inconsistent, but then it would be required to expand to all neighbours when the individual is reprocessed to solve the inconsistent state. Due to nominals, we can easily get such highly connected individuals even if there are no explicit role assertions in the ontology that often involve the same individuals.

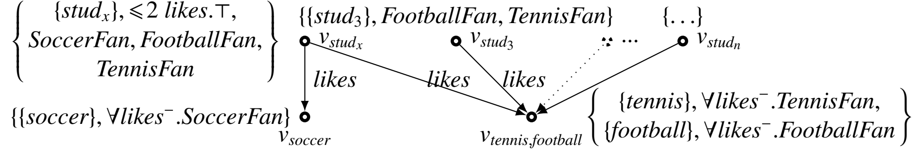
As a remedy, one can make the update process slightly more sophisticated such that the correct handling is ensured by the designated thread that integrates the update. Roughly speaking, if the expansion limit is reached, then the current iterator over the to be expanded neighbours for an individual is stored in the update. In addition, also other iterators for neighbouring individuals (to which an expansion is required) are added to the update even if they are still pointing to the first element, i.e., the iteration has not started. It must, however, be ensured that links between the integrated individuals are created in the local completion graph, which is, for example, possible by checking for all "propagation cut" individuals (i.e., individuals with unfinished iterators over to be expanded neighbours) whether a link has to be created for the individuals that are already in the completion graph. Now, while integrating the update, we use these iterators to find all individuals to which an expansion was required (but which were not integrated in the completion graph) and mark their cache entries as inconsistent. In addition, we add the propagation cut individuals as possible neighbour for an artificial/fresh role (e.g., $P^R(a)(s) = b$ if b is the propagation cut individual, a the expansion required individual, and s the artificial role), which can be used in the reprocessing for (quickly) determining from which individuals consequences still have to be propagated.

The propagation cuts allow, in combination with the proposed reuse of non-deterministically derived consequences (cf. Section 3.2), for limiting the number of individuals that may have to be integrated in local completion graphs while still being able to detect inconsistencies (as long as the number of individuals that are involved in the clashes are within this limit). This is illustrated with the following example.

Example 3 (Example 2 continued). Let us assume that we have now, in addition to part 1, 2, and 3 of Example 1 and Example 2, the ABox part $\mathcal{A}_4$ consisting of the axioms

$$\forall likes^-.TennisFan(tennis), \quad \leqslant 2\, likes.\top(stud_x), \quad likes(stud_x, football),$$
$$likes(stud_x, soccer), \quad likes(stud_x, tennis),$$

and several other students, say $stud_3, \ldots, stud_m$, that only like the activity football in order to keep the example simple. We further assume that $stud_3, \ldots, stud_m$ are already processed, i.e., we have cache entries of the form $K^R(stud_i)(likes) = \{football\}$ and $K^C(stud_i) = \{\{stud_i\}, FootballFan\}$ for $3 \leq i \leq m$ as well as $K^R(football)(likes) = \{stud_2, \ldots, stud_m\}$. Due to the statement that student $stud_x$ has at-most 2 hobbies (encoded with $\leqslant 2\, likes.\top(stud_x)$), the tableau algorithm has to merge two activities of



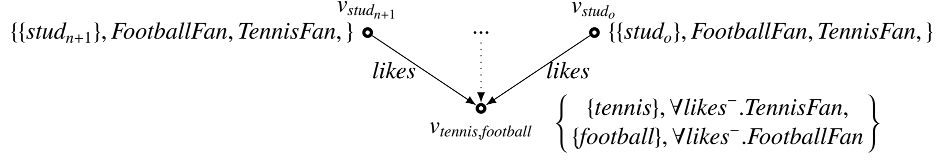
Node				
uni	K^C	$\{uni\}, \forall enr^-.(\forall takes.GC \sqcup \forall takes.UGC)$	P^C	–
	K^R	$enr^- \mapsto \{stud, stud_2\}$	P^R	–
	I	–	$\exists$	–
stud	K^C	$\{stud\}, SoccerFan, \forall takes.GC \sqcup \forall takes.UGC, \neg TennisFan$	P^C	$\forall takes.GC$
	K^R	$enr \mapsto \{uni\}, takes \mapsto \{course\}, likes \mapsto \{soccer\}$	P^R	–
	I	–	$\exists$	–
course	K^C	$\{course\}, \forall takes^-.\neg TennisFan$	P^C	GC, UGC
	K^R	$takes^- \mapsto \{stud, stud_2\}, teaches^- \mapsto \{prof\}$	P^R	–
	I	–	$\exists$	–
soccer	K^C	$\{soccer\}, \forall likes^-.SoccerFan$	P^C	–
	K^R	$likes^- \mapsto \{stud, prof, stud_x\}$	P^R	–
	I	$prof$	$\exists$	$likes^- \mapsto 1$
prof	K^C	$\{prof\}, \exists hc.\exists likes.\{soccer\}, \forall teaches.\forall takes^-.\neg TennisFan, SoccerFan$	P^C	–
	K^R	$teaches \mapsto \{course\}$	P^R	–
	I	–	$\exists$	$hc \mapsto 1$
$stud_2$	K^C	$\{stud_2\}, \neg SoccerFan, FootballFan, \forall takes.GC \sqcup \forall takes.UGC, \neg TennisFan$	P^C	$\forall takes.UGC, \bot$
	K^R	$enr \mapsto \{uni\}, takes \mapsto \{course\}, likes \mapsto \{football\}$	P^R	$t \mapsto \{football\}$
	I	–	$\exists$	–
football	K^C	$\{football\}, \forall likes^-.FootballFan$	P^C	$\{tennis\}, \forall likes^-.TennisFan$
	K^R	$likes^- \mapsto \{stud_3, \ldots, stud_m, stud_2, stud_x\}$	P^R	–
	I	–	$\exists$	–
$stud_3$	K^C	$\{stud_3\}, FootballFan$	P^C	$TennisFan$
	K^R	$likes \mapsto \{football\}$	P^R	$likes \mapsto \{tennis\}$
	I	–	$\exists$	–
...	...	...	...	...
$stud_n$	K^C	$\{stud_n\}, FootballFan$	P^C	$TennisFan$
	K^R	$likes \mapsto \{football\}$	P^R	$likes \mapsto \{tennis\}$
	I	–	$\exists$	–
$stud_{n+1}$	K^C	$\{stud_{n+1}\}, FootballFan$	P^C	$\bot$
	K^R	$likes \mapsto \{football\}$	P^R	$t \mapsto \{football\}$
	I	–	$\exists$	–
...	...	...	...	...
$stud_m$	K^C	$\{stud_m\}, FootballFan$	P^C	$\bot$
	K^R	$likes \mapsto \{football\}$	P^R	$t \mapsto \{football\}$
	I	–	$\exists$	–
tennis	K^C	$\{tennis\}, \forall likes^-.TennisFan$	P^C	$\{football\}, \forall likes^-.FootballFan$
	K^R	$likes^- \mapsto \{stud_x\}$	P^R	$likes^- \mapsto \{stud_3, \ldots, stud_n\}$
	I	–	$\exists$	–
$stud_x$	K^C	$\{stud_x\}, SoccerFan, TennisFan, FootballFan$	P^C	–
	K^R	$likes \mapsto \{soccer, tennis, football\}$	P^R	–
	I	–	$\exists$	–

Fig. 4: Local completion graph with a propagation cut (upper part) and the obtained individual derivations cache (lower part, changes in blue) for handling ABox $\mathcal{A}_4$ of Example 3

$stud_x$, i.e., two of the three nodes $v_{football}$, v_{soccer}, v_{tennis} must (non-deterministically) be

merged together. This would obviously result in clashes since we derived for $stud_2$ that he/she likes football, but is neither a soccer fan nor a tennis fan (preventing that $v_{football}$ is merged with v_{tennis} or v_{soccer}), and for *stud* that he/she is also not a tennis fan, but likes soccer (preventing that v_{tennis} is merged with v_{soccer}). Consequently, the activities cannot be merged and the knowledge base is inconsistent, which is, however, quite difficult to detect if we have many students (which may like different activities) and if we want to limit the expansion, i.e., by only processing a limited number of individuals in completion graphs. For example, let us assume that the tableau algorithm tries to merge the node $v_{football}$ into v_{tennis} (cf. upper part of Figure 4), which yields a node that has $\forall likes^{-}.TennisFan$ as well as $\forall likes^{-}.FootballFan$ in its label. As a consequence, we identify all $likes^{-}$-neighbours of *tennis* as influenced if it is not stored in the cache that *FootballFan* has been derived for them (due to Condition D1). Analogously, all $likes^{-}$-neighbours of *football* must be considered influenced if the cache does not contain *TennisFan* for them. (Due to Condition D5, we must also integrate all $likes^{-}$-neighbours of *tennis* and *football* if they are not already $likes^{-}$-neighbours of both activities such that the neighbour relation can be updated correctly. In our example, however, these are exactly those individuals that are already identified as influenced by Condition D1.) Since there can be a lot of students that do not have both activities as hobbies, this can easily exceed the limitations of the currently handled model construction process (e.g., due to memory constraints). With the propagation cut technique, we would use iterators over lists of individuals that require integration and would process as many of these individuals as possible until the expansion limit is reached. For example, let us assume that the $likes^{-}$-neighbour iterator for *football* first returns $stud_3$ for which we have not yet derived that he/she likes *tennis*. Consequently, it is required to integrate $stud_3$ such that also *TennisFan* can be propagated to it. The integration of influenced individuals from the iterator is continued until we reach the limit, which could, for example, be $stud_n$ with $n << m$, i.e., $stud_n$ is the last individual that is integrated and processed in the local completion graph. After that, the completion graph is considered processed and we send the update and the remaining iterators (or their positions) to the thread that manages the cache. The update is integrated as usual, but the updating thread also iterates over the remaining neighbours (or other lists of potentially influenced individuals) and makes their cache entries inconsistent, i.e., we add $\bot$ to $P^C(stud_i)$ for each $stud_i$ with $i > n$ referred to by the remaining iterators if $stud_i$ has not been integrated in the local completion graph. In addition, we add a possible neighbour relation for an artificial role, say t, to the individual from which the iterator stems (i.e., the individual for which the propagation has been cut), such that we know that consequences from the referenced individual are missing when we start the reprocessing for it. If $stud_i$ is an individual that is referred to by a remaining iterator (that has not been integrated in the local completion graph) and *football* is the individual for which the propagation has been cut, then we add *football* to $P^R(stud_i)(t)$. As a result, we obtain the cache entries that are depicted in the lower part of Figure 4.

For the next reprocessing phase, we retrieve some individuals with an inconsistent state from the cache, say $stud_{n+1}, \ldots, stud_o$ with $o < m$, and we try to build a fully expanded and clash-free completion graph for them (cf. upper part of Figure 5). Since these individuals have an inconsistent state, we cannot reuse possible consequences for



uni	K^C	$\{uni\}, \forall enr^{-}.(\forall takes.GC \sqcup \forall takes.UGC)$	P^C	–
	K^R	$enr^{-} \mapsto \{stud, stud_2\}$	P^R	–
	I	–	$\exists$	–
$stud$	K^C	$\{stud\}, SoccerFan, \forall takes.GC \sqcup \forall takes.UGC, \neg TennisFan$	P^C	$\forall takes.GC$
	K^R	$enr \mapsto \{uni\}, takes \mapsto \{course\}, likes \mapsto \{soccer\}$	P^R	–
	I	–	$\exists$	–
$course$	K^C	$\{course\}, \forall takes^{-}.\neg TennisFan$	P^C	GC, UGC
	K^R	$takes^{-} \mapsto \{stud, stud_2\}, teaches^{-} \mapsto \{prof\}$	P^R	–
	I	–	$\exists$	–
$soccer$	K^C	$\{soccer\}, \forall likes^{-}.SoccerFan$	P^C	–
	K^R	$likes^{-} \mapsto \{stud, prof, stud_x\}$	P^R	–
	I	$prof$	$\exists$	$likes^{-} \mapsto 1$
$prof$	K^C	$\{prof\}, \exists hc.\exists likes.\{soccer\}, \forall teaches.\forall takes^{-}.\neg TennisFan, SoccerFan$	P^C	–
	K^R	$teaches \mapsto \{course\}$	P^R	–
	I	–	$\exists$	$hc \mapsto 1$
$stud_2$	K^C	$\{stud_2\}, \neg SoccerFan, FootballFan, \forall takes.GC \sqcup \forall takes.UGC, \neg TennisFan$	P^C	$\forall takes.UGC, \bot$
	K^R	$enr \mapsto \{uni\}, takes \mapsto \{course\}, likes \mapsto \{football\}$	P^R	$t \mapsto \{football\}$
	I	–	$\exists$	–
$football$	K^C	$\{football\}, \forall likes^{-}.FootballFan$	P^C	$\{tennis\}, \forall likes^{-}.TennisFan$
	K^R	$likes^{-} \mapsto \{stud_3, \ldots, stud_m, stud_2, stud_x\}$	P^R	–
	I	–	$\exists$	–
$stud_3$	K^C	$\{stud_3\}, FootballFan$	P^C	$TennisFan$
	K^R	$likes \mapsto \{football\}$	P^R	$likes \mapsto \{tennis\}$
	I	–	$\exists$	–
...	...	...	...	...
$stud_n$	K^C	$\{stud_n\}, FootballFan$	P^C	$TennisFan$
	K^R	$likes \mapsto \{football\}$	P^R	$likes \mapsto \{tennis\}$
	I	–	$\exists$	–
$stud_{n+1}$	K^C	$\{stud_{n+1}\}, FootballFan$	P^C	$TennisFan$
	K^R	$likes \mapsto \{football\}$	P^R	$likes \mapsto \{tennis\}$
	I	–	$\exists$	–
...	...	...	...	...
$stud_o$	K^C	$\{stud_o\}, FootballFan$	P^C	$TennisFan$
	K^R	$likes \mapsto \{football\}$	P^R	$likes \mapsto \{tennis\}$
	I	–	$\exists$	–
$stud_{o+1}$	K^C	$\{stud_o\}, FootballFan$	P^C	$\bot$
	K^R	$likes \mapsto \{football\}$	P^R	$t \mapsto \{football\}$
	I	–	$\exists$	–
...	...	...	...	...
$stud_m$	K^C	$\{stud_m\}, FootballFan$	P^C	$\bot$
	K^R	$likes \mapsto \{football\}$	P^R	$t \mapsto \{football\}$
	I	–	$\exists$	–
$tennis$	K^C	$\{tennis\}, \forall likes^{-}.TennisFan$	P^C	$\{football\}, \forall likes^{-}.FootballFan$
	K^R	$likes^{-} \mapsto \{stud_x\}$	P^R	$likes^{-} \mapsto \{stud_3, \ldots, stud_n\}$
	I	–	$\exists$	–
$stud_x$	K^C	$\{stud_x\}, SoccerFan, TennisFan, FootballFan$	P^C	–
	K^R	$likes \mapsto \{soccer, tennis, football\}$	P^R	–
	I	–	$\exists$	–

Fig. 5: Local completion graph (upper part) and the obtained individual derivations cache (lower part, changes in blue) for the first reprocessing step from Example 3

them. Due to the possible neighbour relation to *football* for the artificial role t, we know that we have to expand to *football* to get all consequences. Since *football* and *tennis* have consistent states, we can reuse their consequences, i.e., we (non-deterministically) add the possible consequences from the cache in the fixed-reuse branch, for which these nodes are merged together. Now, we can basically continue to propagate *FootballFan* or *TennisFan* to those students that could not be handled in the previous parts due to the expansion limit. After processing the completion graph, we can update the cache, which results in the entries depicted in the lower part of Figure 5. Note that the cache entries for $stud_{n+1}, \ldots, stud_o$ have a consistent state now.

At some point, we also reprocess $stud_2$ and by trying to reuse the possible consequences for *tennis* and *football* in the fixed-reuse branch, we discover a clash (since $\neg$*TennisFan* has been derived for $stud_2$). By tracing back the causes of the clash, we can identify the involved individuals $stud_2$, *tennis*, and *football* and mark them "problematic". In the prioritised-reuse branch, we have to expand to *football* and *tennis* and even further since possible consequences are missing (which cannot be reused as already determined in the fixed-reuse branch). Again, we cannot integrate all (possible) neighbours of *football* and *tennis* due to the expansion limit and, due to the amount of students, it is unlikely that we integrate $stud_x$ that forces us to merge the activities. Nevertheless, we can integrate as many individuals as possible and then we can cut the propagation again, i.e., we send, in addition to the next update, iterators to the cache, where the cache entries for the remaining individuals are marked inconsistent. Let us assume, that $stud_x$ has indeed not been integrated, but its cache entry has been marked inconsistent due to the propagation cut. Hence, we have to reprocess $stud_x$ at some point, for which the activities have to be merged again. Let us assume that the tableau algorithms tries again to merge the node for *football* into the node for *tennis*, such that their neighbours must be checked and integrated. Since $stud_2$ is identified problematic and it is a neighbour of *football*, we prioritise the expansion to it. As a consequence, the previously discovered clash can immediately be detected and the tableau algorithm has to check whether a different merging can be satisfied. This may result in steps that are similar to the previous ones (with possibly several propagation cuts) until all clash causing individuals are identified problematic. Nevertheless, we can eventually detect the inconsistency of the knowledge base if we try to merge some of these activities and directly expand to all these problematic individuals. Note that the expansion to problematic individuals is prioritised and we even expand to them if the expansion limit is already reached in order to ensure that consistency can correctly be checked. In the worst case, we have to reprocess all individuals for each non-deterministic alternative that results in a clash with new involved individuals until all of them are identified as problematic. Another worst-case scenario is that the reuse of non-deterministic consequences often leads to clashes such that most individuals are identified problematic. Consequently, it can become necessary to expand to all these individuals, which bypasses the expansion limit.

It is worth pointing out that some propagation cuts can be realised more efficiently. For example, if we cut the propagation for a deterministically added universal restriction, then we can directly add the propagated concept to all remaining neighbours in the cache while installing the update and marking their entries incomplete. If some of these

neighbours are reprocessed, then it may not be necessary to expand to the individual for which the propagation was cut in order to achieve compatibility.

3.5 Termination and Correctness

The repeated (possibly parallel) reprocessing of individuals from the cache threatens the termination of the consistency checking procedure. In particular, the tableau algorithm may repeatedly derive different consequences for the same individuals in different completion graphs, which could be incompatible with each other, such that we get oscillating cache entries. This effect can even be amplified with techniques for limiting the expansion (e.g., the propagation cut technique presented in Section 3.4), where only parts of the (possibly) affected individuals are handled as well as their cache entries updated and the possible consequences may or may not be reused in subsequent completion graph construction processes. However, as already discussed, termination can easily be ensured by stepwise increasing the (limit of the) number of individuals that are reprocessed together in one completion graph. Consequently, we would, in the worst case, eventually build a completion graph that includes all individuals and their assertions, which would yield an update that leaves all entries in a consistent state. In case of parallelisation, one must further ensure that the last completion graph is based on the last version of the cache and that there are no other parallel completion graph construction processes (which may result in updates that make entries inconsistent again). This can easily be achieved by prohibiting parallelisation and by synchronising with the cache updating thread if the number of potential inconsistent entries is lower than the current limit, which is increased with each scheduled model construction task.

It is further clear that the consistency checking procedure with the cache is sound, i.e., if there is a model of the knowledge base, then it will eventually guarantee the existence of a fully expanded and clash-free completion graph. In fact, it is well-known that the tableau algorithm for $\mathcal{SROIQ}$ is sound [16] and since we apply it for parts of the ABox and only reuse deterministically derived facts in subsequent model construction processes, we cannot derive invalid consequences. Note that we propose to also reuse non-deterministically derived consequences as an extension of our approach (cf. Section 3.2), but since they are also reused non-deterministically, it would be necessary to rederive them directly with the tableau algorithm if they had an actual impact on the result.

It remains to show completeness, i.e., the resulting cache entries guarantee the existence of a fully expanded and clash-free completion graph (considering all individuals), which can be unravelled to a model of the knowledge base. We show this in the following for the base version of the proposed procedure since the discussed extensions rely on it and fall back to the base version if they are not applicable or do not optimise the work-flow. For this, we use the following notation: Let C be the individual derivations cache and $G^1 = (V^1, E^1, \mathcal{L}^1, \dot{\neq}^1), \ldots, G^n = (V^n, E^n, \mathcal{L}^n, \dot{\neq}^n)$ the local completion graphs that are constructed for (re)processing the ABox assertions and the individuals with inconsistent cache entries. We assume that the completion graph number reflects the order in which the updates are integrated into the cache. Moreover, let b be the function that returns the number on which a completion graph cache entry is based, i.e., $\mathsf{b}(a)$ returns k if $v_a \in V^k$ and there is no $l > k$ with $v_a \in V^l$ such that the extracted update may or

may not change the cache entry for a. We write v_a^k for node $v_a \in V^k$, i.e., $v_a^{\mathsf{b}(a)}$ is the node from the completion graph with the last integrated update for the cache entry for a. Now we can build a fully expanded and clash-free completion graph $G = (V, E, \mathcal{L}, \dot{\neq})$ w.r.t. the entire knowledge base $\mathcal{K}$ by borrowing the appropriate nodes and edges from the local completion graphs on which the cache entries are based as follows:

Definition 6 (Full Completion Graph). *Let $\mathcal{K}$ be a knowledge base and C the individual derivations cache that is obtained with updates from the local completion graphs $G^1 = (V^1, E^1, \mathcal{L}^1, \dot{\neq}^1), \ldots, G^n = (V^n, E^n, \mathcal{L}^n, \dot{\neq}^n)$, which are fully expanded and clash-free w.r.t. the processed individuals/assertions. W.l.o.g. we assume that the new nominals are differently named for all local completion graphs. The* full completion graph *$G = (V, E, \mathcal{L}, \dot{\neq})$ is obtained by setting $V, E, \mathcal{L}$, and $\dot{\neq}$ as follows:*

- $V = \bigcup_{1 \leq k \leq n} (W^k_{\mathsf{inds}(\mathcal{K})} \cup W^k_{\mathsf{impl}})$, *(V1)*

  with $W^k_{\mathsf{inds}(\mathcal{K})} = \{v_a^k \mid a \in \mathsf{inds}(\mathcal{K})$ *and* $\mathsf{b}(a) = k\}$ *are the* individual nodes *and* $W^k_{\mathsf{impl}} = \{v^k \mid v^k$ *is an implied descendant of* $W^k_{\mathsf{inds}(\mathcal{K})}\}$ *the* implied nodes *of* G^k;

- $E = \bigcup_{1 \leq k \leq n} \{\langle v_1, v_2 \rangle \in E^k \mid v_1, v_2 \in V \cap V^k\} \cup$ *(E1)*

  $\bigcup_{1 \leq k \leq n} \{\langle v^k, v_a^{\mathsf{b}(a)} \rangle \mid a \in \mathsf{inds}(\mathcal{K}), \mathsf{b}(a) \neq k, v^k \in V \cap W^k_{\mathsf{impl}}, \langle v^k, v_a^k \rangle \in E^k\} \cup$ *(E2)*

  $\{\langle v_a^{\mathsf{b}(a)}, v_c^{\mathsf{b}(c)} \rangle \mid \mathsf{b}(a) \neq \mathsf{b}(c), c \in K^R(a)(r) \cup P^R(a)(r)$ *for some* $r \in \mathsf{rols}(\mathcal{K})\}$; *(E3)*

- $\mathcal{L} = \bigcup_{1 \leq k \leq n} \{v \mapsto \mathcal{L}^k(v) \mid v \in V \cap V^k\} \cup$ *(LV1)*

  $\bigcup_{1 \leq k \leq n} \{\langle v_1, v_2 \rangle \mapsto \mathcal{L}^k(\langle v_1, v_2 \rangle) \mid v_1, v_2 \in V \cap V^k$ *and* $\langle v_1, v_2 \rangle \in E \cap E^k\} \cup$ *(LE1)*

  $\bigcup_{1 \leq k \leq n} \{\langle v^k, v_a^{\mathsf{b}(a)} \rangle \mapsto \mathcal{L}^k(\langle v^k, v_a^k \rangle) \mid a \in \mathsf{inds}(\mathcal{K}), \mathsf{b}(a) \neq k, v^k \in V \cap W^k_{\mathsf{impl}}$,
  and $\langle v^k, v_a^k \rangle \in E^k\} \cup$ *(LE2)*

  $\{\langle v_a^{\mathsf{b}(a)}, v_c^{\mathsf{b}(c)} \rangle \mapsto \{s \in \mathsf{rols}(\mathcal{K}) \mid c \in K^R(a)(s) \cup P^R(a)(s)\} \mid \mathsf{b}(a) \neq \mathsf{b}(c)\}$; *(LE3)*

- $\dot{\neq} = \bigcup_{1 \leq k \leq n} \{\langle v_1, v_2 \rangle \in \dot{\neq}^k \mid v_1, v_2 \in V \cap V^k\} \cup$ *(IE1)*

  $\bigcup_{1 \leq k \leq n} \{\langle v^k, v_a^{\mathsf{b}(a)} \rangle \mid a \in \mathsf{inds}(\mathcal{K}), \mathsf{b}(a) \neq k, v^k \in V \cap W^k_{\mathsf{impl}}, \langle v^k, v_a^k \rangle \in \dot{\neq}^k\} \cup$ *(IE2)*

  $\{\langle v_a^{\mathsf{b}(a)}, v_c^{\mathsf{b}(c)} \rangle \mid \mathsf{b}(a) \neq \mathsf{b}(c)$ *and* $\neg\{c\} \in K^C(a) \cup P^C(a)(r)\}$, *(IE3)*

  where the $\dot{\neq}$ *relation is symmetric, i.e.,* $\langle v_1, v_2 \rangle \in \dot{\neq}$ *if* $\langle v_2, v_1 \rangle \in \dot{\neq}$.

We can now prove completeness of the approach by showing that none of the tableau expansion rules can be applied to the full completion graph and that it is clash-free.

Lemma 1 (Completeness). *Let $\mathcal{K}$ be a knowledge base and C the individual derivations cache that is obtained via updates from the constructed local completion graphs $G^1 = (V^1, E^1, \mathcal{L}^1, \dot{\neq}^1), \ldots, G^n = (V^n, E^n, \mathcal{L}^n, \dot{\neq}^n)$ for certain individuals/assertions. If $G^1, \ldots, G^n$ are clash-free and fully expanded w.r.t. the integrated individuals/assertions under the compatibility/expansion criteria of Definition 2, then there exists a model of the knowledge base $\mathcal{K}$.*

Proof. Let $G = (V, E, \mathcal{L}, \dot{\neq})$ be the full completion graph as defined in Definition 6. We first show that the clash conditions (cf. [16]) are not satisfied for G:

- Clash Condition 1 and 2 are trivially not satisfied since each node label is taken from exactly one local completion graph, i.e., if Clash Condition 1 and 2 were satisfied for G, then a completion graph of $G^1, \ldots, G^n$ would already contain a clash, which contradicts our assumption.
- Clash Condition 3 is not satisfied since only Part E1 of the definition of E can lead to loops/self-edges of a node, which are labelled through LE1. Hence, the clash condition cannot be satisfied since these edges (with their labels) stem from the same local completion graph as the associated nodes (with their labels), i.e., the clash would already be in the local completion graph, which contradicts our assumption.
- Clash Condition 4 can also not be satisfied for the full completion graph, as revealed by the following case-by-case analysis of the definition of E and $\mathcal{L}$. The edges and edge labels from E1 and LE1 stem from one completion graph, i.e., the clash would already be in a local completion graph, which contradicts our assumption. For E2 and LE2, we can only obtain an edge that is labelled with roles r and s for clash-free local completion graphs with $\mathsf{Disj}(r, s) \in \mathcal{K}$ if $v_a^{\mathsf{b}(a)} = v_c^{\mathsf{b}(c)}$ for some $a, c \in \mathsf{inds}(\mathcal{K})$. However, due to Condition $\mathsf{D4}$, all indirect connected individuals would be integrated if there is a (later) local completion graph with $v_a^l = v_c^l$ and, therefore, we would have $k = l$, which is, however, excluded for E2 and LE2 (cf. $\mathsf{b}(a) \neq k$). Analogously for E3 and LE3, we can only obtain an edge labelled with r and s for clash-free local completion graphs with $\mathsf{Disj}(r, s) \in \mathcal{K}$ if there are $a, c, d \in \mathsf{inds}(\mathcal{K})$ such that $c \in K^R(a)(r) \cup P^R(a)(r)$, $d \in K^R(a)(s) \cup P^R(a)(s)$, and $v_c^{\mathsf{b}(c)} = v_d^{\mathsf{b}(d)}$ with $\mathsf{b}(c) \neq \mathsf{b}(a)$ (otherwise they would stem from the same completion graph). However, if $v_c^l = v_d^l$ for a local completion graph, then Condition $\mathsf{D5}$ ensures that a would be integrated if a is not yet a (possible) r^-- as well as s^--neighbour, i.e., we would have $\mathsf{b}(a) = \mathsf{b}(c) = \mathsf{b}(d)$, which is, however, excluded by LE3.
- For Clash Condition 5, we observe that the "trees" of blockable nodes (including new nominal nodes) are taken as a whole from local completion graphs (cf. V1). Moreover, the edges and $\dot{\neq}$-relations between these blockable as well as new nominal nodes (and the associated node and edge labels) are analogously taken over from the local completion graphs. Hence, Clash Condition 5 can only be satisfied if nodes that represent individuals are involved. If there is an at-most restriction of the form $\leqslant m\, r.C$ in a blockable or new nominal node $v \in V$, then we can observe from E2, LE2, and IE2 that there can be at-most the same number of connected/related individual nodes as in the local completion graph from which the blockable node has been taken. In fact, we could have $v_a^{\mathsf{b}(a)} = v_c^{\mathsf{b}(c)}$ such that less neighbours exists in G (which certainly does not introduce a clash) or that we have an entry for the $\dot{\neq}$-relation that is not present in the local completion graph. The latter is, however, also not problematic since the local completion graph would not be fully expanded if there there are more r-neighbours than allowed by $\leqslant m\, r.C \in \mathcal{L}(v)$, which contradicts our assumption. If we have $\leqslant m\, r.C \in \mathcal{L}(v_a^{\mathsf{b}(a)})$, then we know, due to Condition $\mathsf{D4}$, that all blockable/new nominal nodes that are r-predecessors of $v_a^{\mathsf{b}(a)}$ stem from the same local completion graph as $v_a^{\mathsf{b}(a)}$. Consequently, Clash Condition 5 can only be satisfied if there are new corresponding r-neighbours that represent individuals for $v_a^{\mathsf{b}(a)}$ in G or if some of these neighbours are now related w.r.t. $\dot{\neq}$ in G. A clash for the latter can easily be excluded since this would mean that

there are more r-neighbours than allowed and the $\leqslant$-rule would be applicable for the local completion graph, which contradicts our assumption that the local completion graphs are fully expanded. If there were new corresponding r-neighbours that represent individuals for $v_a^{\mathsf{b}(a)}$, then they would have been added through E3 and LE3, i.e., there would be some $c \in \mathsf{inds}(\mathcal{K})$ with $c \in K^R(a)(r) \cup P^R(a)(r)$. However, we can only add c to $K^R(a)(r)$ or $P^R(a)(r)$ if a and c are integrated in a local completion graph. Since E3 and LE3 only adds edges and labels that stem from different local completion graphs and, due to Condition D3, we know that the (number or) r-neighbours has been considered in the local completion graph $G^{\mathsf{b}(a)}$. Consequently, if the number of (new) corresponding r-neighbours were greater than n in G, then there would already be a clash in the local completion graph $G^{\mathsf{b}(a)}$, which contradicts our assumption.

- For Clash Condition 6, we observe that an occurrence of a nominal $\{a\}$ in the label of a node forces us to integrate the individual a (from the cache) and the o-rule of the tableau algorithm ensures that there exists only one node for $\{a\}$. Moreover, we have to integrate all individuals which are (already) possible instances of $\{a\}$ w.r.t. the cache (due to Condition G4) and, hence, all individuals that are (possible) instances of $\{a\}$ w.r.t. the cache are based on the same node from one local completion graph. Since $\langle v, v\rangle$ is only added to $\dot{\neq}$ for nodes, where the relation already holds in the local completion graph, the local completion graph would have a clash or would not be fully expanded, which contradicts our assumption.

Next, we show that the tableau expansion rules (cf. [16]) are not applicable to G:

- The $\sqcap$- and the $\sqcup$-rule are trivially not applicable since all node labels are taken from the local completion graphs, which are fully expanded.
- For the $\exists$-rule w.r.t. a concept $\exists r.C \in \mathcal{L}(v^k)$, we have two cases: If there exists an implied r-successor w^k of v^k with $C \in \mathcal{L}^k(w^k)$ in G^k, then, through V1, LV1, E1, and LE1, we must have $w^k \in V$ with $C \in \mathcal{L}(w^k)$, $\langle v^k, w^k\rangle \in E$, and $r \in \mathcal{L}(\langle v^k, w^k\rangle)$. In contrast, if $\exists r.C \in \mathcal{L}^k(v^k)$ is satisfied by an individual node in G^k, say v_a^k, then we have $\langle v^k, v_a^{\mathsf{b}(a)}\rangle \in E, r \in \mathcal{L}(\langle v^k, v_a^{\mathsf{b}(a)}\rangle)$, and $C \in \mathcal{L}(v_a^{\mathsf{b}(a)})$. In fact, the former holds by Condition G1 (which leads to the integration of all neighbours if non-deterministically derived links are missing) and E3 as well as LE3 if v^k is an individual node. If v^k is an implied node, then the same is ensured by Part E2 and LE2 of the full completion graph definition as well as the fact that we only remove indirectly connected individuals from the same update (i.e., if they no longer have the nominal as "dependency" in the local completion graph). $C \in \mathcal{L}(v_a^{\mathsf{b}(a)})$ holds by Condition G2 (which leads to the integration of all neighbours if a non-deterministically derived concept is missing) and E3 as well as LE3 if v^k is an individual node. If v^k is an implied node, then this is analogously ensured by Condition G3 (which leads to the integration of all indirectly connected individuals if a non-deterministically derived concept is missing) and Part E2 as well as LE2 of the full completion graph definition. We further observe that nodes in G are blocked by the same nodes as in the local completion graph from which they stem and, thus, we can observe that the $\exists$-rule is not applicable for G.
- The Self-rule is not applicable for a concept $\exists r.\mathsf{Self} \in \mathcal{L}(v^k)$ since loops/self-edges (and their labels) are copied with the nodes (see E1 and LV1), i.e., if the Self-rule

was applicable for v^k, then it would have been applicable in G^k, which contradicts our assumption that G^k is fully expanded. Note that the handling of a reflexive role is typically "absorbed" via an axiom $\exists r.\top \sqsubseteq \exists r.\mathsf{Self}$.

- The $\forall$-rule is similar to the $\exists$-rule. We consider only cases that involve individual nodes since the "trees" of implied nodes are taken as a whole from local completion graphs, which are fully expanded by assumption. If we have $\forall r.C \in \mathcal{L}(v^k)$ with v^k an implied node and $\langle v^k, v_a^{\mathsf{b}(a)}\rangle \in E$ as well as $r \in \mathcal{L}(\langle v^k, v_a^{\mathsf{b}(a)}\rangle)$, then it must hold that $C \in \mathcal{L}(v_a^{\mathsf{b}(a)})$. This is the case since G^k is fully expanded, i.e., $C \in \mathcal{L}^k(v_a^k)$, and, due to Condition G3, which would lead to the integration of all indirectly connected individuals for a in a later completion graph if a (non-deterministically) derived concept was missing. In contrast, if $\forall r.C \in \mathcal{L}(v_a^k)$ with v_a^k an individual node, then we have two cases: If there is an implied r-predecessor $w^l \in V$ of v_a^k (from Part E2 and LE2), then we have $\forall r.C \in \mathcal{L}^l(v^l)$ (otherwise Condition D4 would have forced us to integrate all indirectly connected individuals for v_a^k such that we would have $l = k$, which is, however, excluded by E2 and LE2) and, thus, $C \in \mathcal{L}^l(w^l)$, which implies $C \in \mathcal{L}(w^l)$ from the definition of the full completion graph. If there is an r-neighbour $w_c^l \in V$ of v_a^k (from Part E3 and LE3) with w_c^l an individual node, then $C \in \mathcal{L}^l(w_c^l)$ must hold due to Condition D1 (if $l < k$, i.e., the later processing of v_a^k as an r^--neighbour ensures the integration of w_c^l if the concept C has not already been derived for it) and due to fact that v_a^k must be integrated if v_c^l is later established as an r-neighbour (which would mean $l = k$, but this is excluded by E3 and LE3). As a consequence, we have $C \in \mathcal{L}(w_c^l)$ and, thus, the $\forall$-rule is not applicable.
- The choose-rule is, analogously to the $\forall$-rule, not applicable (by using Condition D2 instead of Condition D1).
- The applicability of the $\geqslant$-rule is similar to the case of the $\exists$-rule, but needs additional consideration of the defined $\dot{\neq}$ relation. Again, we only consider cases that involve individual nodes and stem from different completion graphs since the "trees" of implied nodes are taken as a whole from the local completion graphs. For two individual nodes, say v_a^k and v_c^k, with $v_a^k \dot{\neq}^k v_c^k$, we observe that $v_a^{\mathsf{b}(a)} \dot{\neq} v_c^{\mathsf{b}(c)}$ must hold due to IE3 and because $v_a^k \dot{\neq}^k v_c^k$ is encoded as $\neg\{c\}$ for a as well as $\neg\{a\}$ for c and Condition G2 as well as Condition G3 ensure that the neighbours and/or indirectly connected individuals are integrated if such a concept is missing in a later completion graph such that the $\dot{\neq}$ causing $\geqslant$-rule is reapplied. In contrast, if we have an implied node, say $v_k \in V$ with $v_k \dot{\neq}^k v_a^k$, then $v_k \dot{\neq} v_a^{\mathsf{b}(a)}$ must also hold due to IE2. As a consequence, $\dot{\neq}$-relations for nodes that are taken from local completion graphs can also be found in the full completion graph and, therefore, the $\geqslant$-rule is not applicable.
- The argumentation of the non-applicability of the $\leqslant$-rule is analogous to case of Clash Condition 5.
- As already argued for Clash Condition 6, there is only one node for each nominal such that the o-rule is not applicable for the full completion graph.
- For the NN-rule, we can observe that, due to Condition D4, the new nominal nodes stem from the same local completion graph as the node for which the NN-rule has been applied to create these new nominal nodes. Note that new nominals are added to new nominal nodes such that, due to Condition D1, all neighbours must be integrated if an individual node is (re)processed that has been merged with a new

nominal node and, thus, the (individual) node that requires the application of the NN-rule must also be integrated and reprocessed. As a consequence, the NN-rule is not applicable for the full completion graph.

Since the full completion graph is fully expanded and clash-free, it can be unravelled to a model of the knowledge base.

4 Query Answering Support

Compared to other more sophisticated reasoning tasks, conjunctive query answering is typically more challenging since an efficient reduction to consistency checking is not easy. However, a new approach for answering (conjunctive) queries has recently been introduced, where the query atoms are "absorbed" into several simple DL-axioms [28]. These "query axioms" are of the form $C \sqsubseteq {\downarrow}x.S^x$, $S^x \sqsubseteq \forall r.S^x_r$, $S^x \sqcap A \sqsubseteq S^x_A$, and $S^x \sqcap S^y \sqsubseteq S^{xy}$, where ${\downarrow}x.S^x$ is a binder concept that triggers the creation of variable mappings in the extended tableau algorithm and S (possibly with sub- and/or superscripts) are so-called query state concepts that are associated with variable mappings, as defined in the following, in order to keep track of partial matches of the query in a completion graph.

Definition 7 (Variable Mappings). *A* variable mapping μ *is a (partial) function from variable names to nodes. Let $G = (V, E, \mathcal{L}, \dot{\neq}, \mathcal{M})$ be an (extended) completion graph, where $\mathcal{M}(C, v)$ denotes the sets of variable mappings that are associated with a concept C in $\mathcal{L}(v)$. A variable mapping $\mu_1 \cup \mu_2$ is defined by setting $(\mu_1 \cup \mu_2)(x) = \mu_1(x)$ if x is in the domain of μ_1, and $(\mu_1 \cup \mu_2)(x) = \mu_2(x)$ otherwise. Two variable mappings μ_1 and μ_2 are* compatible *if $\mu_1(x) = \mu_2(x)$ for all x in the domain of μ_1 as well as μ_2. The* join *$\mathcal{M}_1 \bowtie \mathcal{M}_2$ between the sets of variable mappings $\mathcal{M}_1$ and $\mathcal{M}_2$ is defined as:*

$$\mathcal{M}_1 \bowtie \mathcal{M}_2 = \{\mu_1 \cup \mu_2 \mid \mu_1 \in \mathcal{M}_1, \mu_2 \in \mathcal{M}_2 \text{ and } \mu_1 \text{ is compatible with } \mu_2\}.$$

Rules of the extended tableau algorithm are shown in Table 2 (without considering blocking), which handle the new axioms and concepts by correspondingly creating and propagating variable mappings. For example, a binder concept ${\downarrow}x.S^x \in \mathcal{L}(v)$ is handled by adding S^x to $\mathcal{L}(v)$ and by creating a mapping $\{x \mapsto v\}$ that is associated with S^x for v, i.e., $\{x \mapsto v\} \in \mathcal{M}(S^x, v)$. In contrast, variable mappings associated with a concept $\forall r.S^x_r$ for a node v are propagated (in addition to S^x_r) to all r-neighbours, i.e., $S^x_r \in \mathcal{L}(w)$ with $\mathcal{M}(S^x_r, w) \supseteq \mathcal{M}(\forall r.S^x_r, v)$ if w is an r-neighbour of v. For (binary) inclusion axioms with query state concepts, the variable mappings are propagated to the implied concept if the left hand-side is satisfied for a node. For example, $S^x_A \in \mathcal{L}(v)$ with $\mu \in \mathcal{M}(S^x_A, v)$ if $S^x, A \in \mathcal{L}(v)$ and $\mu \in \mathcal{M}(S^x, v)$ for an axiom $S^x \sqcap A \sqsubseteq S^x_A$. Note that only compatible variable mappings of S^x and S^y are propagated to S^{xy} for $S^x \sqcap S^y \sqsubseteq S^{xy}$. Although conjunctive query answering with arbitrary existential variables is still open for $\mathcal{SROIQ}$, the approach works for knowledge bases where only a limited number of new nominal nodes is enforced (by using an extended analogous propagation blocking technique) [28], which is generally the case in practice (also see [28] for more details w.r.t. correctness and termination of this approach).

As an example, a simple query with only the atoms $r(x, y)$ and $s(y, x)$ (with x, y both answer variables) can systematically be absorbed into the axioms $\top \sqsubseteq {\downarrow}x.S^x$, $S^x \sqsubseteq$

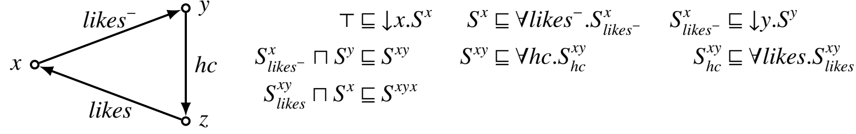


Fig. 6: Illustration of query $Q(x, y) = \{likes^-(x, y), hc(y, z), likes(z, x)\}$ (left-hand side) and the axioms generated by "absorbing" $Q(x, y)$ (right-hand side) of Example 4

$\forall r.S^x_r$, $S^x_r \sqsubseteq {\downarrow}y.S^y$, $S^x_r \sqcap S^y \sqsubseteq S^{xy}$, $S^{xy} \sqsubseteq \forall s.S^{xy}_s$, and $S^{xy}_s \sqcap S^x \sqsubseteq S^{xyx}$. The query state concept S^x_r, for example, represents the state where bindings for x are propagated to r-successors, i.e., $r(x, y)$ is satisfied. For bindings that are propagated back over s-edges via $\forall s.S^{xy}_s$, the final binary inclusion axiom checks whether the cycle is closed. If it is, the joined variable mappings are associated with S^{xyx} from which answer candidates can be extracted once a fully expanded and clash-free completion graph is found.

Note that with sophisticated absorption techniques, variable mappings can often be derived deterministically, i.e., they directly constitute query answers. Non-deterministically obtained variable mappings do, however, require a separate entailment check to verify that there exist no counter example with the query variables equally bound as in the non-deterministically derived variable mapping. This can be realised by restricting the generated binder concepts of the absorption process to only create corresponding bindings and by triggering a clash with the additional axiom $S^{xyx} \sqsubseteq \bot$.

While query answering by absorption is able to process queries for many (expressive) real-world ontologies [28], especially queries with existential variables can require a substantial amount of computation. A significant bottleneck is often the (variable mappings) *propagation task*, i.e., the creation and propagation of the variable mappings to get all potential answers from a completion graph. Building and using completion graphs for partial ABoxes (possibly in parallel) is difficult since it is unclear which joins of bindings can occur in answers and, hence, how the ABox can suitably be partitioned.

Example 4. The query $Q(x, y) = \{likes^-(x, y), hc(y, z), likes(z, x)\}$ (cf. left-hand side of Figure 6) retrieves all persons (y) with their hobbies (x) that they have in common with their children (z) from the knowledge base. For the answer variables x and y, the reasoner has to return those pairs of individuals that constitute an instance of the role $likes^-$ and for which a possibly implied individual exist such that the restrictions for the existential variable z are satisfied.

The query can be absorbed by choosing a starting variable, say x, for which a binder concept is implied (see first axiom of right-hand side of Figure 6). All role atoms are now successively "absorbed" by generating a universal restriction that propagates the next query sate concept over the corresponding role. For the role atom $likes^-(x, y)$, we generate $S^x \sqsubseteq \forall likes^-.S^x_{likes^-}$ such that $S^x_{likes^-}$ in a node label (associated with some variable mappings) indicates that the state of the query is reached, where this role atom is satisfied. Since y is an answer variable, we also have to collect bindings for y which is realised through the axiom $S^x_{likes^-} \sqsubseteq {\downarrow}y.S^y$. In order to join/concatenate the new bind-

ings for y with the existing propagating (with bindings for x), we generate a binary inclusion axiom of the form $S^x_{likes^-} \sqcap S^y \sqsubseteq S^{xy}$. The role atoms $hc(y, z)$ and $likes(z, x)$ are analogously absorbed via propagation axioms with universal restrictions over the corresponding roles. Note that z is only an existential variable and it is not required to join different propagations at the position of z and, therefore, bindings for z are not required, i.e., we do not have to imply a binder concept for z. In order to ensure that is correctly checked whether the cycle is closed, we use the the binary inclusion axiom $S^{xy}_{likes} \sqcap S^x \sqsubseteq S^{xyx}$ for which the join of compatible variable mappings is propagated to S^{xyx}, where compatibility means that bindings for common variables map to the same nodes of the completion graph. By building a completion graph for a knowledge base that is extended by the absorbed query axioms, we can extract the answer candidates from the variable mappings that are associated with S^{xyx}.

Interestingly, the work-flow with the individual derivations cache is quite handy to improve different aspects of the discussed query answering approach. Although the techniques may also be applicable with other completion graph caching techniques, they lead to a particularly nice combination with our caching technique since only small parts of the ABox are simultaneously considered (i.e., we do not maintain a completion graph for the entire knowledge base, but only have a few cached consequences for the remaining parts). This allows for handling much larger knowledge bases for which parallelisation is particularly useful (since queries for small knowledge bases can often trivially be answered within a few milliseconds).

The techniques in the following focus on splitting the propagation work such that each thread can completely determine a few answer candidates while considering only a small part of the ABox. As an alternative, one could propagate (possibly the same) variable mappings with several threads over different parts of the ABox, which would, again, require some kind of synchronisation. Although we could extend the individual derivations cache to also handle this kind of synchronisation (it is already used to synchronise ordinary consequences such as implied concepts), variable mappings are much more problematic. In fact, variable mappings have a more complex structure and can refer arbitrary individuals. Moreover, it is difficult to store variable mappings with existential variables bound to anonymous individuals (i.e., blockable or new nominal nodes in the completion graph), for which it is unclear how to represent these bindings such that variable mappings can correctly be reused.

It is clear that the (creation of) bindings for the first (answer) variable can be restricted,[1] i.e., the first binder concept is restricted such that it can only bind to a few predefined individuals, which is already utilised by the original approach to restrict the creation of variable mappings with results from the realisation reasoning task. This can also be used for parallelisation (by partitioning the individuals for the first answer variable such that, for each propagation task, only bindings to a few individuals are allowed), but this can easily lead to a work imbalance if all or almost all answers consist of the same individual for the first variable. Using binding restrictions for the second

[1] For simplicity, we assume that the first answer variable is also the first one for which the absorption generates the first binder concept, but in practise the variables are typically reordered based on statistics for the concept and role atoms of the query such that the propagation effort through the completion graph is minimised or at least reduced.

Table 2: Tableau rule extensions

	$\downarrow$-rule:
if	$\downarrow x.C \in \mathcal{L}(v)$, and $C \notin \mathcal{L}(v)$ or $\{x \mapsto v\} \notin \mathcal{M}(C, v)$
then	$\mathcal{L}(v) = \mathcal{L}(v) \cup \{C\}$, $\mathcal{M}(C, v) = \mathcal{M}(C, v) \cup \{\{x \mapsto v\}\}$
	$\forall$-rule:
if	$\forall r.C \in \mathcal{L}(v)$, there is an r-neighbour w of v with $C \notin \mathcal{L}(w)$ or $\mathcal{M}(\forall r.C, v) \nsubseteq \mathcal{M}(C, w)$
then	$\mathcal{L}(w) = \mathcal{L}(w) \cup \{C\}$, $\mathcal{M}(C, w) = \mathcal{M}(C, w) \cup \mathcal{M}(\forall r.C, v)$
	$\sqsubseteq_1$-rule:
if	$S^{x_1 \ldots x_n} \sqsubseteq C \in \mathcal{K}$, $S^{x_1 \ldots x_n} \in \mathcal{L}(v)$, and $C \notin \mathcal{L}(v)$ or $\mathcal{M}(S^{x_1 \ldots x_n}, v) \nsubseteq \mathcal{M}(C, v)$
then	$\mathcal{L}(v) = \mathcal{L}(v) \cup \{C\}$, $\mathcal{M}(C, v) = \mathcal{M}(C, v) \cup \mathcal{M}(S^{x_1 \ldots x_n}, v)$
	$\sqsubseteq_2$-rule:
if	$S^{x_1 \ldots x_n} \sqcap A \sqsubseteq C \in \mathcal{K}$, $\{S^{x_1 \ldots x_n}, A\} \subseteq \mathcal{L}(v)$, and $C \notin \mathcal{L}(v)$ or $\mathcal{M}(S^{x_1 \ldots x_n}, v) \nsubseteq \mathcal{M}(C, v)$
then	$\mathcal{L}(v) = \mathcal{L}(v) \cup \{C\}$, $\mathcal{M}(C, v) = \mathcal{M}(C, v) \cup \mathcal{M}(S^{x_1 \ldots x_n}, v)$
	$\sqsubseteq_3$-rule:
if	$S_1^{x_1 \ldots x_n} \sqcap S_2^{y_1 \ldots y_m} \sqsubseteq C \in \mathcal{K}$, $\{S_1^{x_1 \ldots x_n}, S_2^{y_1 \ldots y_m}\} \subseteq \mathcal{L}(v)$, and $(\mathcal{M}(S_1^{x_1 \ldots x_n}, v) \bowtie \mathcal{M}(S_2^{y_1 \ldots y_m}, v)) \nsubseteq \mathcal{M}(C, v)$
then	$\mathcal{L}(v) = \mathcal{L}(v) \cup \{C\}$, $\mathcal{M}(C, v) = \mathcal{M}(C, v) \cup (\mathcal{M}(S_1^{x_1 \ldots x_n}, v) \bowtie \mathcal{M}(S_2^{y_1 \ldots y_m}, v))$

Algorithm 1 recPropTask(R, i)

Input: Variable binding restrictions R and the index of the next to be handled variable

```
 1: if i ≤ n then
 2:     B ← recPropTask(R, i + 1)
 3:     for each x_j with 1 ≤ j < i do
 4:         R(x_j) ← B(x_j)
 5:     end for
 6:     while |B(x_i)| ≥ l do
 7:         R(x_i) ← R(x_i) \ B(x_i)
 8:         B_t ← recPropTask(R, i + 1)
 9:         B(x_i) ← B_t(x_i)
10:     end while
11:     B(x_i) ← ∅
12: else
13:     G ← buildComplGraph(R, l)
14:     C ← C ∪ answerCands(G)
15:     B ← extractBoundIndis(G)
16: end if
17: return B                  ▷ Returning the bound individuals from the last constructed completion graph
```

variable (or for even more variables) is not easily possible since we do not directly know upfront, to which individuals the second variable can be bound if we have certain bindings for the first variable. Of course, one could generate and compute sub-queries where we ask for bindings of the second variable given one or several bindings for the first, but this would result in many additional queries and, hence, is likely that this would cause a significant overhead.

We can, however, use a dynamic approach, where we limit the number of individuals to which a variable can be bound. Each individual bound to such a "binding-limited" variable is recorded and in the next propagation task we exclude bindings to already tested individuals. This can, for example, be realised with the recursive function recPropTask shown in Algorithm 1, which takes as input a mapping R from variables to (still) allowed bindings for individuals and the index i of the current variable (assuming that the variables are sorted in the order in which they are absorbed, i.e., x_1 denotes the variable that is absorbed first). The function accesses and modifies some variables via side effects, namely l, denoting the limit for the number of allowed bindings for each variable, n, standing for the number of variables in the query, and C, denoting the set of answer candidates. The function is initially called with $R(x) = \mathsf{inds}(\mathcal{K})$ for each variable x in the query and with $i = 1$ such that the restrictions for the first variable are managed first. As long as there are more variables to handle, the function calls itself recursively for the next variable (cf. Line 2) and checks for the returned sets of bound

individuals, denoted with B, from the last generated completion graph whether the limit l has been reached for the current variable. If this is the case, then the bindings for previous variables are "frozen", i.e., they are interpreted as the only allowed bindings (cf. Line 3–5), and the used bindings for the current variable are excluded for the next propagation task (cf. Line 7). This ensures that all combinations are tested step-by-step and that each propagation task only creates and propagates a limited amount of variable mappings. In fact, if the restrictions for all variables are set, then they are used for constructing the next completion graph (Line 13), where R and l are checked by an adapted $\downarrow$-rule. Subsequently, we can extract the additionally found answer candidates (Line 14) and the individuals that have been used for bindings (Line 15), which may or may not be part of answer candidates. For the completion graph construction, we only pass the binding restrictions R and the binding limit l in the algorithm, but it obviously also relies on the axioms of the knowledge base and the query absorption. Although the presented variant for splitting and limiting the propagation task is sequential, it can easily be parallelised by partitioning the allowed bindings for the next variable beforehand.

Note that, in addition to a separate limit for each variable, one can further impose an overall limit for bindings (independently from a variable), but it has to be ensured that at least for all variables some bindings are possible. If it is unclear how to determine these limits, one can start with small ones and can increase them step-by-step as long as the propagation tasks can still easily be managed (e.g., w.r.t. the memory consumption and the computation time).

Note that the cached non-deterministic consequences from the consistency check can directly be reused for the propagation tasks since the new concepts from the query absorption do not lead to (new) clashes. This is different, however, for the separate query entailment checks, where the query absorption implies $\bot$ at the end to trigger the investigation of different (non-deterministic) alternatives. On the other hand, the concepts from the query absorption typically cause an expansion of the local completion graph due to influence criteria, e.g., if $\forall r.S^x_r$ is in the label of some node, then Condition G1 identifies all r-neighbours from the cache as (potentially) influenced and the corresponding nodes need to be integrated into the completion graph to propagate the associated variable mappings to them. This can result in significant propagation work, in particular, for complex roles and individuals with many neighbours. Moreover, exhausted binding restrictions for the next variable might prevent us from actually using the mappings. To address this, one can impose propagation restrictions for universal restrictions of the form $\forall r.S^x_r$ such that the local completion graph is only expanded to nodes for which bindings are possible for the next variable. This can easily be implemented by adapting the query absorption to annotate universal restrictions with the variable of the role atom to which the propagation occurs. For example, a concept of the form $\forall r.S^x_r$ resulting from the query atom $r(x, y)$ is annotated with y, denoted as $\forall r_{\to y}.S^x_r$. Condition D1 is then adapted to only identify individuals as influenced that are allowed as bindings for the labelled variable of the universal restriction.

Definition 8 (Query Propagation Influence). *Let $G = (V, E, \mathcal{L}, \dot{\neq}, \mathcal{M})$ be an (extended) completion graph for a knowledge base $\mathcal{K}$, C an individual derivations cache (c.f. Def. 1), $v_a \in V$ a node representing the individual a, and y a query variable. A* restriction set $R(y) \subseteq \mathsf{inds}(\mathcal{K})$ for *y restricts the individuals to which y can be bound,*

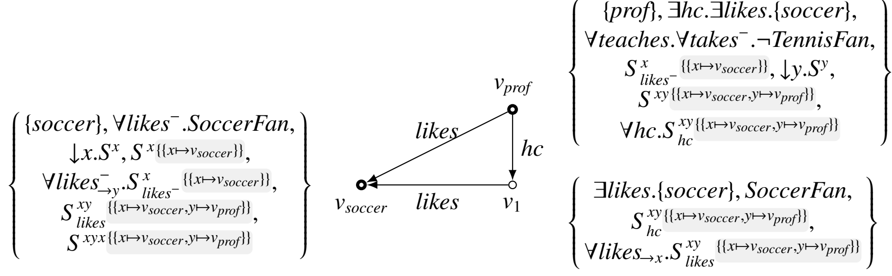


Fig. 7: Local completion graph for a query propagation task of Example 5

i.e., only to a node v_a *if* $a \in R(y)$. *An individual* $b \in \mathsf{C}$ *such that no node in* V *contains* $\{b\}$ *in its label is* (query propagation) influenced *if*

Q1 $\forall r_{\to y}.S_r^{x_1,\ldots,x_n} \in \mathcal{L}(v_a)$, $b \in K^R(a)(r) \cup P^R(a)(r)$, *and* $b \in R(y)$.

As mentioned, if the restrictions for a variable are not known upfront, then one can collect them dynamically by only imposing a limit for the number of individuals for the restriction set. While we check whether an individual b is query propagation influenced w.r.t. a variable y and the amount of individuals in the restriction set $R(y)$ is less than the limit, we simply add b to $R(y)$ such that Condition Q1 is satisfied. Analogously, we add b to $R(y)$ when we test whether we can bind y to b for $\downarrow$-concepts and the limit is not yet reached. When the limit is reached, no more individuals are added to the restriction set and, therefore, no other (combinations of) variable mappings are created and the completion graph is not further expanded to other individuals. The collected restrictions are then used in the next propagation task to enforce the exploration of other (combinations of) variable mappings. Note, however, that the expansion with the query propagation influence does not work for roles with recursive role inclusion axioms (such as transitive roles) due to the unfolding process and due to the fact that it would be too restrictive. In principle, it would be possible to limit the expansion also for complex/transitive roles to some extent by checking query propagation influence in a depth-first based manner, but this would require some non-trivial adaptations to the tableau algorithm, where the propagation depth is managed, and also a more sophisticated management of binding restrictions. For simplicity, we, therefore, assume that only universal restrictions with simple roles are labelled in the absorption process such that influenced individuals for universal restrictions with complex roles are identified with Condition D1. Consequently, complex/transitive roles can lead to an unnecessary expansion to individuals for which the propagation of the query is not further continued by the $\downarrow$-rule (since it adheres the binding restrictions for the next variable).

Example 5 (Example 1 and 4 continued). In order to restrict the expansions for propagation tasks for query $Q(x, y)$ of Example 4, we replace the universal restrictions $\forall likes^{-}.S^{x}_{likes^{-}}$ and $\forall likes.S^{xy}_{likes}$ with $\forall likes^{-}_{\to y}.S^{x}_{likes^{-}}$ and $\forall likes_{\to x}.S^{xy}_{likes}$, respectively. We do not replace $\forall hc.S^{xy}_{hc}$ since z is an existential variable that only binds to blockable nodes in our example for which we do not have to restrict the expansion. In order to

show the approach with the small example, let us assume that the expansion is restricted to only one individual for each variable. By analysing/indexing the cache entries of Example 1, we already know that only the activities (i.e., *soccer*, *tennis*, *football*) have outgoing links for the role $likes^-$. Hence, we can partition the propagation tasks w.r.t. the activities by asserting $\downarrow x.S^x$ only to one of them (instead of using the axiom $\top \sqsubseteq \downarrow x.S^x$). For *soccer*, the rule for $\downarrow x.S^x$ creates the variable mapping $\{x \mapsto v_{soccer}\}$, where x is mapped to *soccer*, and associates it with S^x, which is denoted with $S^{x\,\{\{x \mapsto v_{soccer}\}\}}$ in the label of node v_{soccer} of Figure 7. After unfolding S^x to the universal restriction $\forall likes^-_{\to y}.S^x_{likes^-} \in \mathcal{L}(v_{soccer})$, for which we keep the associated variable mappings, we have to check for influenced individuals from the cache. In principle, the individuals *stud* and *prof* of $K^R(soccer)(likes^-)$ are influenced, but due to our expansion restriction, we cannot integrate both. In fact, while checking Condition Q1 for the first individual, say *prof*, we add it to $R(y)$ and prohibit the addition of other individuals since we reached the limit of one individual per variable. Since the Condition Q1 then identifies *prof* influenced, we integrate it into the completion graph. Now, the application of the $\forall$-rule for $\forall likes^-_{\to y}.S^x_{likes^-} \in \mathcal{L}(v_{soccer})$ adds $S^{x\,\{\{x \mapsto v_{soccer}\}\}}_{likes^-}$ to v_{prof}, where the query state concept is unfolded to $\downarrow y.S^y$. After creating the variable mapping $\{y \mapsto v_{prof}\}$ for the binder concept $\downarrow y.S^y$, we join it with the one associated with $S^x_{likes^-}$ and propagate the joined variable mapping $\{x \mapsto v_{soccer}, y \mapsto v_{prof}\}$ to S^{xy} in order to satisfy $S^x_{likes^-} \sqcap S^y \sqsubseteq S^{xy}$. After unfolding the query state concept to the next universal restriction, we propagate the associated variable mapping to $S^{xy}_{hc} \in \mathcal{L}(v_1)$ and, after one more unfolding, it is propagated back to v_{soccer}. Since the join of $\{\{x \mapsto v_{soccer}, y \mapsto v_{prof}\}\}$ and $\{\{x \mapsto v_{soccer}\}\}$ is $\{\{x \mapsto v_{soccer}, y \mapsto v_{prof}\}\}$ (and obviously non-empty), we propagate it to S^{xyx}. Since the variable mapping has deterministically been propagated to S^{xyx}, we can directly identify $\langle soccer, prof\rangle$ as an answer for $Q(x, y)$. For *soccer*, it remains to repeat the propagation tasks, where *prof* is prohibited for $R(y)$, which results in the integration of *stud*. However, there are no children for *stud* and, hence, the query propagation cannot be completed. Note that *prof* must also be integrated due to Condition D4, but we only create a variable mapping for the binder concept for y if *prof* is in $R(y)$ (or if *prof* can still be added to $R(y)$). The propagation tasks for the other activities are analogous but do not lead to additional answers. Nevertheless, all these propagation tasks only require the integration of a limited number of individuals, which makes them (more) suitable for parallelisation.

It is worth pointing out that the presented mechanisms can also be used for existential variables. As mentioned, existential variables can bind to blockable nodes in completion graphs, which cannot easily be restricted, but if their binder concepts occur in the label of a node that represents a known individual of the knowledge base, then we can use the dynamic binding restrictions. Additionally, we can also use the query propagation influence condition for universal restrictions towards existential variables. If answer candidates are extracted, then we have to validate the candidates with query entailment checks, where bindings for (answer) variables are restricted to the individuals of the answer candidates, and, therefore, the query propagation influence condition can further be used for these entailment checks to ensure that only few individuals have to be integrated into the local completion graph.

In principle, these mechanisms can even be used for "pure" query entailment checking with only existential variables in the query. For this, we assume that the generating rules (i.e., the $\exists$-, the $\geqslant$-, and the NN-rule) of the tableau algorithm add an artificial, symmetric super role $u^{\exists}$ to the edge labels towards newly generated nodes. Now, the absorption can generate axioms of the form $F \sqsubseteq \forall u^{\exists}.F$ and $F \sqsubseteq \downarrow x.S^{x}$ (instead of $\top \sqsubseteq \downarrow x.S^{x}$) to force the addition of the first binder concept. By partitioning the individuals of the knowledge base, we can then create many entailment checking tasks, where the concept F is only asserted to the individuals of the corresponding partition. Again, we can label universal restrictions with variables such that the query propagation influence condition usually expands only a small part of the entire ABox via the dynamic binding restrictions/limits. Note, however, that the $u^{\exists}$ role may be added to the labels of edges between nodes that represent individuals (e.g., if nominals are used or blockable nodes are merged), which could result in a redundant evaluation of the query for some individuals. Nevertheless, this helps in splitting the propagation work such that query entailment checking (and also the answer candidate propagations) can be realised with several small batches instead one batch (which may not be reasonably processable if the ontology is very large and also not easily parallelisable).

5 Implementation and Experiments

We implemented the individual derivations cache with the presented extensions in the tableau-based reasoning system Konclude with minor adaptations to fit the architecture and the optimisations of Konclude. In particular, we use Konclude's efficient, but incomplete saturation procedure [29] to initialise the cache entries for all individuals. If completeness of the saturation cannot be guaranteed for an individual, we mark the corresponding cache entry as inconsistent such that it is reprocessed with the tableau algorithm. Parallel processing (via small batches) is straightforward for the saturation as individuals with their assertions are handled separately. This automatically leads to a very efficient handling of the "simple parts" of an ABox and it only remains to implement the (repeated) reprocessing of individuals with inconsistent cache entries.

For ontologies that intensively state many individuals as identical or for which many and large same individual clusters must be derived, a naive implementation of the cache can lead to an (exponential) blow up in the number of neighbour relations that have to be stored, which can be quite problematic in practise. To address this, we choose a representative individual among the individuals that are deterministically derived as the same and establish a data sharing mechanism such that only the representative individual can be updated and the associated cache entries also appear for the other individuals. To make this more efficient, we use special entries for (possibly) same individuals instead of encoding them as nominals within K^{C} and P^{C}. In case of certain language features (such as functional roles), the same individual clusters may frequently be extended from different completion graphs, which may also change the representative individual (e.g., if it is determined by the smallest/largest individual id/hash). As a consequence, parallelisation may not work since the entries of individuals are often identified as inconsistent, which also prevents the establishment of the data sharing mechanism and could make the reprocessing phase even slower than without parallelisation. As a remedy, it

Table 3: Evaluated ontologies, number of queries (from the PAGOdA+VLog evaluation), and parsing times with different number of threads in seconds (speedup factor in parentheses)

Ontology	DL	#Axioms	#Assertions	#Q	K-1	K-2	K-4	K-8
ChEMBL	$\mathcal{SRIQ}(\mathcal{D})$	3,171	$255.8 \cdot 10^6$	6 + 3	1830	935 (2.0)	497 (3.7)	268 (6.8)
$LUBM_{800}$	$\mathcal{ALEHI}^+(\mathcal{D})$	93	$110.5 \cdot 10^6$	35 + 3	363	184 (2.0)	102 (3.6)	56 (6.5)
Reactome	$\mathcal{SHIN}(\mathcal{D})$	600	$87.6 \cdot 10^6$	7 + 3	66	34 (1.9)	19 (3.6)	11 (6.2)
$Uniprot_{100}$	$\mathcal{ALCHOIQ}(\mathcal{D})$	608	$109.5 \cdot 10^6$	–[3]	409	229 (1.8)	119 (3.5)	63 (6.7)
$Uniprot_{40}$	$\mathcal{ALCHOIQ}(\mathcal{D})$	608	$42.8 \cdot 10^6$	13 + 3	215	113 (1.9)	59 (3.6)	33 (6.5)
$UOBM_{500}$	$\mathcal{SHIN}(\mathcal{D})$	246	$127.4 \cdot 10^6$	20 + 0	431	227 (1.9)	121 (3.5)	66 (6.5)

seems a good idea to introduce an additional phase, where all inconsistent individuals from the saturation phase are reprocessed based on their initial cache entries.

Since tableau algorithms are usually quite memory intensive, scalability of the parallelisation not only depends on the CPUs but also on the memory bandwidth and access. Hence, the memory allocator must scale well and the data must be organised in a way that allows for effectively using the CPU caches (e.g., by writing the data of entries with one thread in cohesive memory areas). We investigated different memory allocators (hoard, tcmalloc, jemalloc) and integrated jemalloc [10] since it seems to work best in our scenario. The worker threads for constructing completion graphs only extract the data for cache updates. A designated thread then integrates the cache updates, based on the update ids introduced on page 14, which reduces blocking, improves memory management, and allows for more sophisticated update mechanisms.

To further improve the utilisation of multi-processor systems and to avoid bottlenecks, we also parallelised some other processing steps, e.g., parsing of large RDF triple files, some preprocessing aspects (i.e., extracting internal representations from RDF triples), and indexing of the cache entries for retrieving candidates for query answering. Also note that some higher-level reasoning tasks of Konclude are already (naively) parallelised by creating and processing several consistency checking problems in parallel.

For evaluating the approach,[2] we used the large ontologies and the appertaining queries from the PAGOdA evaluation [43], which includes the well-known LUBM and UOBM benchmarks as well as the real-world ontologies ChEMBL, Reactome, and Uniprot[3] from the European Bioinformatics Institute. To improve the evaluation w.r.t. the computation of large amounts of answers, we further include the queries from tests for the datalog engine VLog [6], but we use them w.r.t. the original TBoxes. We run the evaluations on a Dell PowerEdge R730 server with two Intel Xeon E5-2660V3 CPUs at 2.4 GHz and 512 GB RAM under a 64bit Ubuntu 18.04.3 LTS. For security reasons and due to multi user restrictions, we could, however, only utilise 480 GB RAM and 8 CPU cores of the server in a containerised environment (via LXD).

[2] Source code, evaluation data, all results, and a Docker image (kocludeeval/parqa) are available at online, e.g., at `https://zenodo.org/record/4606566`.

[3] We evaluated query answering on a sample (denoted with $Uniprot_{40}$) since the full Uniprot ontology ($Uniprot_{100}$) is inconsistent and, hence, not interesting for evaluating query answering.

Metrics of the evaluated ontologies are depicted on the left-hand side of Table 3, whereas the right-hand side shows the (concurrent) parsing times for the ontologies in seconds, where K-1, K-2, K-4, and K-8 stand for the versions of Konclude, where 1, 2, 4, and 8 threads are used, respectively. Since the parallel parsing hardly requires any synchronisation and only accesses the memory in a very restricted way, it can be seen as a baseline for the achievable scalability (there are minor differences based on how often the different types of assertions occur).

The left-hand side of Table 4 shows the (concurrent) pre-computation times, i.e., the time that is required to get ready for query answering after parsing the ontology, which includes the creation of the internal representation, preprocessing the ontology (e.g., absorption), saturating the concepts and individuals, repeatedly reprocessing the individuals with inconsistent cache entries, classifying the ontology, and preparing data structures for an on-demand/lazy realization. Consistency checking clearly dominates the (pre-)computation time such that the other steps can mostly be neglected for the evaluation (e.g., classification takes only a few milliseconds for these ontologies). As shown in Table 4, our parallelisation approach with the individual derivations cache is able to significantly reduce the time required for consistency checking, but the scalability w.r.t. the number of threads depends on the ontology. For LUBM and ChEMBL, the approach scales almost as well as the parsing process, whereas the scalability w.r.t. Reactome seems limited. The Reactome ontology intensively relies on (inverse) functional roles such that many and large clusters of same individuals are derived in the reasoning process. With a naive implementation of the cache, we would store, for each individual in a cluster, all derived neighbour relations, which easily becomes infeasible if large clusters of same individuals are linked. For our implementation of the cache, we identify and utilise representative individuals to store the neighbour relations more effectively, but we require consistent cache entries for this. If the clusters of same individuals are updated in parallel, which often leads to inconsistent cache entries, more neighbour relations must be managed and, thus, the parallelisation of ontologies such as Reactome only works to a limited extent with the current implementation. Also note that the enormous amounts of individuals in these ontologies make it impossible for the previous version of Konclude to build full completion graphs covering the entire ABox, i.e., the version of Konclude without the cache quickly runs out of memory for these ontologies. Also note that the individuals from the cache are mostly picked in the order in which they are indexed, i.e., more or less randomly due to hashing of pointers. Nominals, however, are indexed first and, hence, are prioritised in the (re-)processing. Clearly, the processing order can have a significant influence on how much (re-)processing is required, but the runs for the evaluated real-world ontologies showed hardly any variance since most consequences could be derived locally.

The right-hand side of Table 4 reveals the query answering times (and scalability), accumulated for each ontology. Since not all steps are parallelised and the version of Konclude with only one thread uses specialised and more efficient implementations in some cases (e.g., an optimised join algorithm for results from several sub-queries, whereas the parallelised version is based on several in-memory map-reduce steps), query answering scalability leaves still room for improvement. Nevertheless, without splitting the propagation tasks, several queries cannot be computed, i.e., the version of

Table 4: (Pre-)computation and accumulated query answering times for the evaluated ontologies with different numbers of threads in seconds (speedup factor in parentheses)

Ontology	(Pre-)computing				Query answering			
	K-1	K-2	K-4	K-8	K-1	K-2	K-4	K-8
ChEMBL	2421	1244 (1.9)	663 (3.7)	397 (6.1)	12767	8927 (1.4)	4507 (2.8)	3231 (4.0)
LUBM_{800}	2793	1658 (1.7)	831 (3.4)	437 (6.4)	2777	1829 (1.5)	1026 (2.7)	569 (4.8)
Reactome	1408	687 (2.0)	427 (3.3)	361 (3.9)	935	524 (1.8)	333 (2.8)	232 (4.0)
Uniprot_{100}	1343	742 (1.8)	429 (3.1)	302 (4.4)	N/A[3]	N/A[3]	N/A[3]	N/A[3]
Uniprot_{40}	1090	532 (2.0)	289 (3.7)	198 (5.5)	28	21 (1.3)	16 (1.8)	14 (2.0)
UOBM_{500}	1317	735 (1.8)	394 (3.3)	245 (5.4)	3774	1799 (2.1)	947 (4.0)	554 (6.8)

Konclude without the presented (query answering) splitting techniques cannot answer all of the queries within the given memory and time limits. Moreover, the parallelisation significantly improves the query answering times and the improvements are larger, the more computation is required. For ChEMBL, there is one very challenging query that results in around 2.4 billion answers, which requires the majority of the time and almost all of the available memory (447 GB). Note that the query answers must be serialised and sent to the evaluation client, which communicates with the reasoner via the loopback device and simply discards all received parts of the query results if they exceed a certain size. For large results (e.g., the 2.4 billion answers require roughly 500 GByte in the SPARQL XML result serialization), Konclude streams the answers to the client via the HTTP chunked transfer encoding. Due to the relatively small batches in which tasks are processed, the parallelisation does not require significantly more memory.

To the best of our knowledge, the only other system that is able to handle such enormous and expressive ontologies is PAGOdA, which delegates most of the workload to an efficient datalog engine (such as RDFox) by utilising different (lower and upper bound) optimisations as well as rewriting techniques and only falls back to a fully-fledged reasoner if required by the query. Note that we used PAGOdA with RDFox as the underlying datalog engine, which, in principle, supports parallelisation, but it seem that this is hardly utilised by PAGOdA. We verified the correctness of the implementation by comparing the results with PAGOdA (as long as the serialised results do not exceed 1 GByte) and created a comparison of the required processing times, which is shown in Table 5. Since it is not easily possible to separate the parsing of the ontologies from other steps for PAGOdA, we summarize all the time that is required until the system is ready for query answering as preprocessing time.

As one can observe, PAGOdA and Konclude have different strengths and weaknesses w.r.t. the preprocessing of these ontologies. As mentioned, Reactome leads to many and large clusters of same individuals, which are difficult to handle with the individual derivations cache. In addition, PAGOdA ignores several data range axioms with disjunctions, which lead to a significant amount of non-determinism for Konclude. The type of inconsistency in Uniprot_{100} (multiple values for functional data properties) seems to be checked much earlier with PAGOdA than for Konclude. In principle, one could extend the saturation in Konclude to check (values for) data properties more precisely such that the inconsistency is detected much faster, but this does not seem very

Table 5: Comparison between Konclude and PAGOdA w.r.t. the preprocessing and accumulated query answering times for the evaluated ontologies in seconds

Ontology	Preprocessing			Query answering		
	K-1	K-8	PAGOdA	K-1	K-8	PAGOdA
ChEMBL	4250	665	6553	12767	3231	≥ 17424
LUBM$_{800}$	3156	493	1552	2777	569	≥ 45291
Âŕ Reactome	1473	371	1032	935	232	2649
Uniprot$_{100}$	1769	366	550	–	–	–
Uniprot$_{40}$	1305	231	2206	28	14	28
UOBM$_{500}$	1747	311	7773	3774	554	≥ 36425

relevant given the fact that most ontologies are indeed consistent. For LUBM, it can be assumed that the more direct handling with the datalog engine under the relative trivial TBox is more efficient than the handling with the individual derivations cache in Konclude. In contrast, the preprocessing of Konclude with one thread is more efficient for ChEMBL, Uniprot$_{40}$, and UOBM$_{500}$ than the one of PAGOdA and the parallelised version of Konclude is significantly faster in most cases. Moreover, query answering seems faster and more reliable with Konclude than with PAGOdA. In fact, PAGOdA reached the time limit of 10 hours for two queries, which may be caused by the fully-fledged reasoner (HermiT) in cases where the query cannot fully be answered with the datalog engine (under the use of rewriting techniques). Although PAGOdA usually require significantly less memory, it reached the memory limit for the ChEMBL query that results in around 2.4 billion answers. Regarding the memory consumption, it also has to be noted that the parsed axioms and assertions are kept in memory for Konclude even after the internal representation is created, which is more than 100 GByte for the larger ontologies and is not really required for the evaluated application scenarios.

6 Conclusions

We presented an approach for splitting the reasoning work w.r.t. the assertional knowledge of an ontology, which includes reasoning tasks such as consistency checking and (conjunctive) query answering. This is particularly useful to enable parallelisation, which benefits from uniformly sized work packages. The foundation of the presented approach is a cache that stores specifically chosen consequences derived for individuals. By reusing cached consequences and by employing appropriate expansion strategies, it is ensured that the partial models constructed in parallel for small parts of the knowledge base are compatible with each other such that the existence of a model considering all individuals and assertions of the knowledge base is guaranteed. We also presented techniques to support and improve (conjunctive) query answering, which utilise and restrict the expansion of these partial models to gather a limited amount of (possible) answers in order to enable a uniform parallelisation. As a result, the reasoning work can be organised in several small batches instead of doing all in one pass, which cannot directly be parallelised and can easily become problematic for large ontologies. We integrated the

presented techniques in the reasoning system Konclude and our experiments with large and expressive ontologies show some promising performance improvements, especially by utilising parallelisation. Given the fact that multi-core processors are ubiquitous now, this are some important advancements to speed up reasoning with expressive Description Logics. The approach may even be a suitable basis for distributed reasoning in a compute cluster, which may require to split the cache entries onto different machines, which should, however, be straightforward by using an appropriate partitioning based on hash values of individuals. Moreover, the approach also seems interesting for realising incremental/stream reasoning, where a few assertions are (frequently) added or removed, which remains to be investigated in future works.

7 Acknowledgements

This research was funded by the German Research Foundation (Deutsche Forschungsgemeinschaft, DFG), project number 330492673.

References

1. Allocca, C., Calimeri, F., Civili, C., Costabile, R., Cuteri, B., Fiorentino, A., Fuscà, D., Germano, S., Laboccetta, G., Manna, M., Perri, S., Reale, K., Ricca, F., Veltri, P., Zangari, J.: Large-scale reasoning on expressive horn ontologies. In: Proc. 3rd Int. Workshop on the Resurgence of Datalog in Academia and Industry (Datalog 2.0'19). CEUR WS Proceedings, vol. 2368, pp. 10–21. CEUR (2019)
2. Baader, F., Calvanese, D., McGuinness, D., Nardi, D., Patel-Schneider, P. (eds.): The Description Logic Handbook: Theory, Implementation, and Applications. Cambridge University Press, second edn. (2007)
3. Bajraktari, L., Ortiz, M., Simkus, M.: Compiling model representations for querying large ABoxes in expressive DLs. In: Proc. 27nd Int. Joint Conf. on Artificial Intelligence (IJCAI'18). pp. 1691–1698 (2018)
4. Bate, A., Motik, B., Grau, B.C., Cucala, D.T., Simancik, F., Horrocks, I.: Consequence-based reasoning for description logics with disjunctions and number restrictions. J. of Artificial Intelligence Research **63**, 625–690 (2018)
5. Calvanese, D., De Giacomo, G., Lembo, D., Lenzerini, M., Rosati, R.: Tractable reasoning and efficient query answering in description logics: The DL-Lite family. J. of Automated Reasoning **39**(3), 385–429 (2007)
6. Carral, D., Dragoste, I., González, L., Jacobs, C.J.H., Krötzsch, M., Urbani, J.: VLog: A rule engine for knowledge graphs. In: Proc. 18th Int. Semantic Web Conf. (ISWC'19). pp. 19–35. Springer (2019)
7. Carral, D., González, L., Koopmann, P.: From Horn-SRIQ to datalog: A data-independent transformation that preserves assertion entailment. In: Proc. 33rd AAAI Conf. on Artificial Intelligence (AAAI'19). pp. 2736–2743. AAAI Press (2019)
8. Cucala, D.T., Grau, B.C., Horrocks, I.: Sequoia: A consequence based reasoner for $\mathcal{SROIQ}$. In: Proc. 32nd Int. Workshop on Description Logics (DL'19). CEUR WS Proceedings, CEUR (2019)
9. Dolby, J., Fokoue, A., Kalyanpur, A., Schonberg, E., Srinivas, K.: Scalable highly expressive reasoner (SHER). J. of Web Semantics **7**(4), 357–361 (2009)

10. Evans, J.: A scalable concurrent malloc(3) implementation for FreeBSD. In: Proc. 3rd Technical BSD Conf. (BSDCan'06) (2006)
11. Faddoul, J., MacCaull, W.: Handling non-determinism with description logics using a fork/join approach. Int. J. of Network Computing **5**(1), 61–85 (2015)
12. Glimm, B., Horrocks, I., Motik, B., Stoilos, G., Wang, Z.: HermiT: An OWL 2 reasoner. J. of Automated Reasoning **53**(3), 1–25 (2014)
13. Glimm, B., Kazakov, Y., Tran, T.: Ontology materialization by abstraction refinement in horn $\mathcal{SHOIF}$. In: Proc. 31st AAAI Conf. on Artificial Intelligence (AAAI'17). pp. 1114–1120. AAAI Press (2017)
14. Haarslev, V., Möller, R.: On the scalability of description logic instance retrieval. J. of Automated Reasoning **41**(2), 99–142 (2008)
15. Haarslev, V., Möller, R., Turhan, A.Y.: Exploiting pseudo models for TBox and ABox reasoning in expressive description logics. In: Proc. 1st Int. Joint Conf. on Automated Reasoning (IJCAR'01). LNCS, vol. 2083, pp. 61–75. Springer (2001)
16. Horrocks, I., Kutz, O., Sattler, U.: The even more irresistible $\mathcal{SROIQ}$. In: Proc. 10th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR'06). pp. 57–67. AAAI Press (2006)
17. Hudek, A.K., Weddell, G.E.: Binary absorption in tableaux-based reasoning for description logics. In: Proc. 19th Int. Workshop on Description Logics (DL'06). vol. 189. CEUR (2006)
18. Kazakov, Y.: $\mathcal{RIQ}$ and $\mathcal{SROIQ}$ are harder than $\mathcal{SHOIQ}$. In: Proc. 11th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR'08). pp. 274–284. AAAI Press (2008)
19. Kikot, S., Kontchakov, R., Podolskii, V.V., Zakharyaschev, M.: Exponential lower bounds and separation for query rewriting. In: Proc. 39th Int. Colloquium on Automata, Languages, and Programming (ICALP 2012). vol. 7392, pp. 263–274. Springer (2012)
20. Kiryakov, A., Ognyanov, D., Manov, D.: OWLIM - a pragmatic semantic repository for OWL. In: Proc. Web Information Systems Engineering Workshops (WISE 2005). LNCS, vol. 3807, pp. 182–192. Springer (2005)
21. Kolovski, V., Wu, Z., Eadon, G.: Optimizing enterprise-scale OWL 2 RL reasoning in a relational database system. In: Proc. 9th Int. Semantic Web Conf. (ISWC 2010). pp. 436–452. LNCS, Springer (2010)
22. Kontchakov, R., Lutz, C., Toman, D., Wolter, F., Zakharyaschev, M.: The combined approach to query answering in DL-Lite. In: Proc. 12th Int. Conf. on the Principles of Knowledge Representation and Reasoning (KR 2010). AAAI Press/The MIT Press (2010)
23. Motik, B., Nenov, Y., Piro, R., Horrocks, I., Olteanu, D.: Parallel materialisation of datalog programs in centralised, main-memory RDF systems. In: Proc. 28th AAAI Conf. on Artificial Intelligence (AAAI'14). pp. 129–137. AAAI Press (2014)
24. Motik, B., Shearer, R., Horrocks, I.: Hypertableau reasoning for description logics. J. of Artificial Intelligence Research **36**, 165–228 (2009)
25. Quan, Z., Haarslev, V.: A parallel computing architecture for high-performance OWL reasoning. J. of Parallel Computing **83**, 34–46 (2019)
26. Sirin, E., Cuenca Grau, B., Parsia, B.: From wine to water: Optimizing description logic reasoning for nominals. In: Proc. 10th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR'06). pp. 90–99. AAAI Press (2006)
27. Sirin, E., Parsia, B., Cuenca Grau, B., Kalyanpur, A., Katz, Y.: Pellet: A practical OWL-DL reasoner. J. of Web Semantics **5**(2), 51–53 (2007)
28. Steigmiller, A., Glimm, B.: Absorption-based query answering for expressive description logics. In: Proc. 18th Int. Semantic Web Conf. (ISWC'19). pp. 593–611. LNCS, Springer (2019)
29. Steigmiller, A., Glimm, B., Liebig, T.: Coupling tableau algorithms for expressive description logics with completion-based saturation procedures. In: Proc. 7th Int. Joint Conf. on Automated Reasoning (IJCAR'14). LNCS, Springer (2014), accepted

30. Steigmiller, A., Glimm, B., Liebig, T.: Optimised absorption for expressive description logics. In: Proc. 27th Int. Workshop on Description Logics (DL'14). vol. 1193. CEUR (2014)
31. Steigmiller, A., Glimm, B., Liebig, T.: Completion graph caching for expressive description logics. In: Proc. 28th Int. Workshop on Description Logics (DL'15) (2015)
32. Steigmiller, A., Liebig, T., Glimm, B.: Extended caching, backjumping and merging for expressive description logics. In: Proc. 6th Int. Joint Conf. on Automated Reasoning (IJCAR'12). LNCS, vol. 7364, pp. 514–529. Springer (2012)
33. Steigmiller, A., Liebig, T., Glimm, B.: Konclude: system description. J. of Web Semantics **27**(1) (2014)
34. Tsarkov, D., Horrocks, I.: FaCT++ description logic reasoner: System description. In: Proc. 3rd Int. Joint Conf. on Automated Reasoning (IJCAR'06). LNCS, vol. 4130, pp. 292–297. Springer (2006)
35. Tsarkov, D., Horrocks, I., Patel-Schneider, P.F.: Optimizing terminological reasoning for expressive description logics. J. of Automated Reasoning **39**, 277–316 (2007)
36. Urbani, J., Kotoulas, S., Maassen, J., van Harmelen, F., Bal, H.E.: Webpie: A web-scale parallel inference engine using mapreduce. J. of Web Semantics **10**, 59–75 (2012)
37. Wandelt, S., Möller, R.: Distributed island-based query answering for expressive ontologies. In: Proc. 5th Int. Conf. on Advances in Grid and Pervasive Computing (GPC'10). pp. 461–470. Springer (2010)
38. Wandelt, S., Möller, R.: Towards ABox modularization of semi-expressive description logics. J. of Applied Ontology **7**(2), 133–167 (2012)
39. Wu, J., Hudek, A.K., Toman, D., Weddell, G.E.: Absorption for ABoxes. J. of Automated Reasoning **53**(3), 215–243 (2014)
40. Wu, K., Haarslev, V.: A parallel reasoner for the description logic ALC. In: Kazakov, Y., Lembo, D., Wolter, F. (eds.) Proc. 25th Int. Workshop on Description Logics (DL'12). CEUR Workshop Proceedings, CEUR (2012)
41. Wu, K., Haarslev, V.: Exploring parallelization of conjunctive branches in tableau-based description logic reasoning. In: Proc. 26th Int. Workshop on Description Logics (DL'13). pp. 1011–1023. CEUR (2013)
42. Wu, K., Haarslev, V.: Parallel OWL reasoning: Merge classification. In: Proc. 3rd Joint Int. Semantic Technology Conf. (JIST'13). LNCS, vol. 8388, pp. 211–227. Springer (2013)
43. Zhou, Y., Cuenca Grau, B., Nenov, Y., Kaminski, M., Horrocks, I.: PAGOdA: Pay-as-you-go ontology query answering using a datalog reasoner. J. of Artificial Intelligence Research **54**, 309–367 (2015)