2

Labeling RDF Graphs for Linear Time and Space Querying

Tim Furche, Antonius Weinzierl, and François Bry

Institute for Informatics, University of Munich,
Oettingenstraße 67, D-80538 München, Germany
http://www.pms.ifi.lmu.de/

Abstract. Indices and data structures for web querying have mostly considered tree shaped data, reflecting the view of XML documents as tree-shaped. However, for RDF (and when querying ID/IDREF constraints in XML) data is indisputably graph-shaped. In this chapter, we first study existing indexing and labeling schemes for RDF and graph data in general with focus on support for efficient adjacency and reachability queries. For XML, labeling schemes are an important part of the widespread adoption of XML, in particular for mapping XML to existing (relational) database technology. However, the existing indexing and labeling schemes for RDF (and graph data in general) sacrifice one of the most attractive properties of XML labeling schemes, the constant time (and per-node space) test for adjacency (child) and reachability (descendant). In the second part, we introduce the first labeling scheme for RDF data that retains this property and thus achieves linear time and space processing of acyclic RDF queries on a significantly larger class of graphs than previous approaches (which are mostly limited to tree-shaped data). Finally, we show how this labeling scheme can be applied to (acyclic) SPARQL queries to obtain an evaluation algorithm with time and space complexity *linear in the number of resources* in the queried RDF graph.

2.1 Introduction

"Interesting data is *relationships*" (Tim Berners-Lee[1]). To support this view of data, initiatives such as RDF and Linked Data have been launched and are increasingly adopted beyond just an enthusiast and academic community (cf. RDFa use by Google).

What relationships exist between entities (resources) on the Web can not be regulated in advance. From a data management perspective, this renders storage and access schemes that rely on fixed, pre-established schemata mostly void in the Web context. Rather than storing and accessing relations of a particular schema, Web data management solutions have to deal with data in widely varying, constantly changing schemata.

Storage and access schemes specifically tailored to these properties of relationships on the Web are therefore needed. For XML, where data is mostly considered

[1] Talk on "The Next Web and Linked Data", at TED 2009, http://www.ted.com/index.php/talks/lang/eng/tim_berners_lee_on_the_next_web.html.

tree shaped, such schemes have been developed and implemented with great success, in particular in the form of labeling schemes.

Labeling schemes assign labels to nodes in a tree or graph in such a way that various relations between nodes can be decided given just the labels of two nodes. They have proved to be one of the most pertinent techniques for time and space efficient XML processing and are nowadays employed in most XML databases [35, 6]. Labeling schemes are particularly interesting if queries are mostly concerned with the relationships between entities in a query, e.g., the existence of a certain relation, the reachability between two entities, etc.

For RDF, however, previously proposed storage and access schemes have, with few exceptions, been adjacency- (storing triples directly and using standard joins for reconstructing the graph structure) or schema-based (exploiting regularities in the shape of RDF sub-graphs, e.g., that most resources of a give type have certain properties). The reason for this lack is that graph data poses a greater challenge for labeling schemes than tree data. Where labeling schemes for RDF and similar graph data have been developed [1, 44, 10, 43] these labeling schemes have failed to preserve essential properties that have made tree labeling schemes a success for all or at least a significant number of RDF graphs.

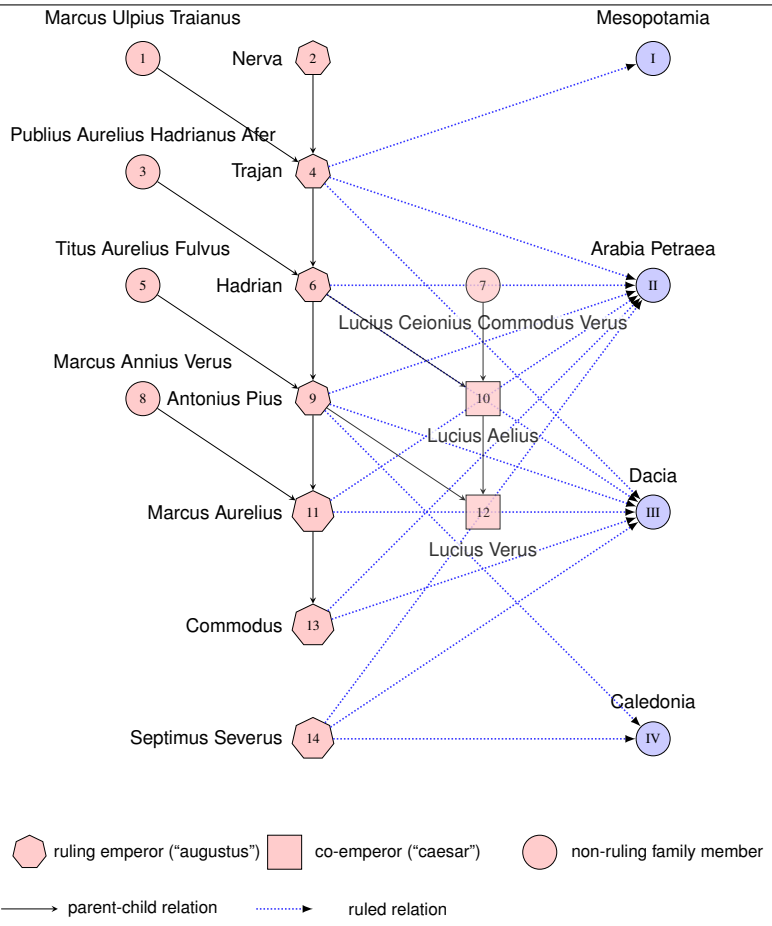
In this chapter, we describe the first labeling scheme for RDF that retains most of the following characteristics:

1. *constant* time *adjacency* test. In other words, the ability to test, in constant time, whether a given triple holds in the queried RDF graph.
2. *enumeration* of all adjacent resources for a given resource in time *linear* in the number of adjacent resources (but independent of the number of overall resources in the graph). In other words, the ability to enumerate all properties of a given resource in time linear in the number of these properties.
3. *constant* time *reachability* test. In other words, the ability to test, in constant time, whether there is a path between two resources in the RDF graph. Just like for adjacency, enumeration of reachable resources is linear in the number of reachable resources, rather than the overall graph.
4. *constant* size labels[2] and thus a *linear overall size* of the labeling (index) in the number of resources in the graph.
5. *polynomial* labeling algorithm that computes the labeling for a given graph.

All these characteristics hold on tree data, e.g., for the pre-/post-encoding used in [6]. However, none of the previously proposed labeling scheme for graph (or specifically RDF data) retains these properties. Though on arbitrary graphs our approach degenerates to quadratic space complexity (as previous approaches), we present the first characterisation of a class of graphs for which these properties are retained yet that is a non-trivial proper super-class of all structures (like XML trees) for which such a guarantee has been shown previously.

[2] In this chapter we adopt the common convention to disregard that the size of labels scales logarithmically with the size of the data. This is reasonable as such labels are usually stored in attributes of a fixed numerical type (such as SQL's `INT` or `BIGINT` type). If label size is of

Fig. 2.1 "The Five Good Emperors" (after Edward Gibbon).



Example. The principle idea of the proposed labeling builds on a property from XML query engines such as SPEX [33] and CAA [30]: to order the nodes of the XML tree in such a way, that we can describe adjacency between nodes through intervals over that order, rather than through explicit storage of the pairs of adjacent nodes.

Our labeling scheme, called CIG-labeling, generalizes this property to general graphs: In a CIG-labeling each node n is labeled with a pair (l_n, I_n) such that

1. l_n is the position of n in some order over the nodes of the graph and
2. I_n is a set of non-overlapping intervals that covers all adjacent nodes of n.

concern, a logarithmic, multiplicative factor in the size of the graph is to be added to all the complexities.

Figure 2.1 shows an RDF graph about roman emperors, their relations, and the provinces they ruled.[3] We only show provinces that changed ownership in the depicted time period, the remaining provinces are ruled by all the emperors shown here and can be added trivially.

Neither the relations between the emperors nor the `ruled` relation between emperors and provinces is tree shaped. In fact, some provinces are ruled by nearly all of the depicted emperors (e.g., Dacia) while others where only in roman hands for a short period of time (e.g., Mesopotamia). The relation between emperors is not tree shaped as the emperors of this time used legal adoption (rather than blood relation) to choose their heir and the depicted parent-child relation mixed legal and biological relations.

Despite the considerable complexity of the relations involved in this example, the adjacent nodes of each emperor can be represented as a single interval for each of the two relations. For instance, the sons of Hadrian form the interval $[9, 10]$, his ruled provinces the interval $[\text{II}, \text{III}]$. For Antonius Pius the intervals are $[11, 12]$ and $[\text{II}, \text{IV}]$.

Since a single interval suffices to cover the adjacent nodes under each relation, the CIG labels are constant for each node. Furthermore, testing whether two nodes (e.g., Hadrian and Dacia) are adjacent under one of the relations requires only two comparisons ($l_{\text{Dacia}} = \text{III} \in \{[\text{II}, \text{III}]\} = I_{\text{Hadrian}}$, i.e., testing that $\text{II} \leq \text{III} \leq \text{III}$).

It should be obvious from the example, however, that the order in which the nodes are labeled has a significant effect on the quality (label size and speed of adjacency test) of the CIG-labeling. We call a CIG-labeling optimal, if the overall size of the I_n is minimal, i.e., if the minimal number of intervals is needed to cover the adjacent nodes of all nodes.

Fortunately, it turns out that this simple generalization of labeling schemes for tree data is already surprisingly useful due to two properties: It is possible to test (and label) in polynomial time whether the nodes in a graph can be ordered in such a way that the adjacent nodes of each node form a single interval. Such a graph is then called a CIG graph and it can be stored and queried *as efficiently as a tree* using the CIG-labeling scheme.

Unfortunately, if a given graph is not a CIG, finding the optimal CIG labeling is NP-complete. However, a polynomial 1.5-approximation algorithm exists that allows the CIG-labeling to outperform adjacency-based approaches even in that case.

2.1.1 Contributions

To summarize, we present a labeling scheme for RDF data that

– generalizes interval based tree labeling schemes such as the pre-/post-encoding [6] or the labeling schemes used in SPEX [33, 31] and CAA [30] to arbitrary graphs. It combines a fast adjacency test (at worst logarithmic in the number of nodes) with often significantly lower storage than adjacency-based schemes (including adjacency lists), see Section 2.4.

[3] For clarity of presentation, we omit technical details of RDF not relevant for this discussion. For instance, which URLs (if any) are chosen to identify the emperors and provinces is of no concern here. We also depict `rdf:label` by adjacent labels, `rdf:type` triples through different node shapes, `ex:father-of` edges black, and `ex:ruled` edges dotted blue.

data	single, ground triple pattern	acyclic basic graph pattern	full graph pattern
tree	$O(\log n), O(1)$	$O(q \cdot n)$	$O(n^{q_g} + q \cdot n)$
CIG	$O(\log n), O(1)$	$O(q \cdot n)$	$O(n^{q_g} + q \cdot n)$
graph	$O(\log n), O(n^2)$	$O(q \cdot n^2)$	$O(n^{q_g} + q \cdot n^2)$

Table 2.1. Complexity of query evaluation with the CIG-labeling (time and space complexity are the same in all cases except for single, ground triple pattern on graphs, where the space complexity is given after the time complexity); q query size, n data size, q_g: number of "graph" variables, i.e., variables with multiple incoming query edges)

- retains the most important properties of tree labeling schemes (see Section 2.3.1), yet extends them to a significantly larger class of graphs, so-called CIGS, than any previous labeling scheme (see Section 2.3.2 on previous graph labeling schemes):
  1. constant-time adjacency (and reachability) test,
  2. constant label size, thus linear overall size of the labeling,
  3. linear time enumeration of adjacent (and reachable) nodes,
  4. polynomial time labeling algorithm.

  The formal definition of CIGS and their properties are given in Section 2.5.
- performs on tree data as good as the best known tree labeling schemes and thus is ideally suited when both tree, CIG, and general graph data is to be processed, see Section 2.6.
- allows the linear time and space evaluation of acyclic SPARQL queries and provides an efficient basis for full SPARQL implementation, see Section 2.6.

Table 2.1 summarizes the complexity results for evaluating three subsets of SPARQL based on the CIG-labeling: for single, ground triple patterns (i.e., testing whether a graph contains a given triple of only named resources), which corresponds to testing adjacency between the two resources in the triple; for acyclic (as in [21]) SPARQL graph patterns with variables and filters; for full SPARQL graph patterns. Note, that there is no penalty at all to going from tree data to CIG data, but arbitrary graph data does incur a logarithmic penalty in time (and a linear penalty in space).

The rest of this chapter is organized as follows: First we briefly introduce or revisit a few notions on RDF and SPARQL needed in the rest of the paper (Section 2.2). Then we investigate existing labeling schemes for RDF and graph data in general, with a brief perspective also on XML labeling schemes (Section 2.3), as these form the basis for many RDF labeling schemes. Section 2.4 finally introduces the general CIG-labeling scheme and its properties on arbitrary graphs. For trees and the novel class of CIGS, it is shown in Section 2.5 that a CIG-labeling with constant label size can be found in polynomial time. In Section 2.6 we briefly discuss how the CIG-labeling is used to evaluate acyclic and full SPARQL queries and even some nSPARQL [38] path expressions.

2.2 Preliminaries—RDF as Graphs

In this section, we briefly revisit basic notions about RDF and RDF queries, in particular SPARQL queries.

RDF graphs contain simple statements about *resources* (which, in other contexts, are be called "entities", "objects", etc., i.e., elements of the domain that may partake in relations). Statements are triples consisting of subject, predicate, and object, all of which are resources. If we want to refer to a specific resource, we use (supposedly globally unique) URIs, if we want to refer to a resource for which we know that it exists and maybe some of its properties, we use *blank nodes* which play the role of existential quantifiers in logic. However, blank nodes may not occur in predicate position. Finally, for convenience, we can directly use *literal values* as objects.

RDF may be serialized in many formats (for a recent survey see [5]), such as RDF/XML [4], an XML dialect for representing RDF, or Turtle [3] which is also used in SPARQL. The following Turtle data, e.g., describes a number of (fictitious) articles, their titles, creators, and relations to conferences.

```
1 @prefix dc: <http://purl.org/dc/elements/1.1/> .
  @prefix dct: <http://purl.org/dc/terms/> .
3 @prefix vcard: <http://www.w3.org/2001/vcard-rdf/3.0#> .
  @prefix bib: <http://www.edutella.org/bibtex#> .
5 @prefix ulp: <http://example.org/roman/libraries/ulpia#> .
  ulp:cicero-46-wt a bib:Article ; dc:title "Wax Tablets" ;
7      dc:creator [ a rdf:Seq ;
           rdf:_1 ulp:cicero ; rdf:_2 ulp:tiro ] ;
9      ulp:cites ulp:hirtius-47-bc ;
       dct:isPartOf ulp:conf-46-mutina .
11 ulp:cicero a bib:Person ; vcard:FN "M. T. Cicero" .
  ulp:tiro a bib:Person ; vcard:FN "M. T. Tiro" .
13 ulp:hirtius-47-bc a bib:Article ;
       ulp:cites ulp:cicero-46-wt ;
15     dct:isPartOf ulp:conf-46-mutina .
  ulp:conf-46-mutina a bib:InProceedings ;
17     rdfs:label "Storage Media" .
```

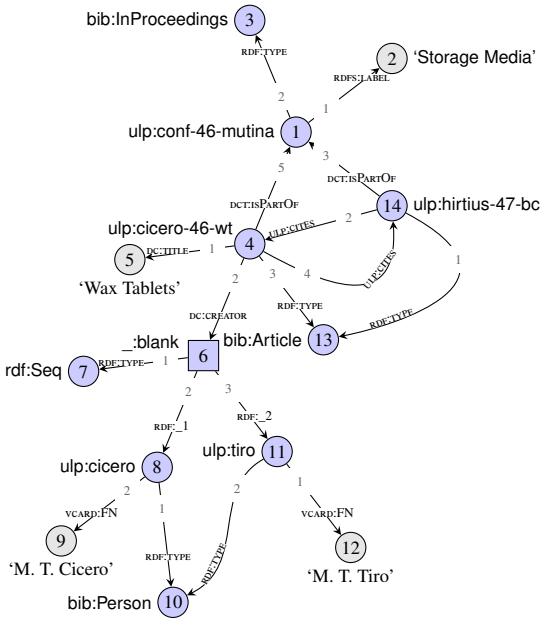
Following the definition of namespace prefixes used in the remainder of the Turtle document (omitting common RDF namespaces), each line contains one or more statements separated by colon or semi-colon. If separated by semi-colon, the subject of the previous statement is carried over. E.g., line 6 reads as ulp:cicero-46-wt is a bib:Article and has dc:title "Wax Tablets". Lines 7–9 show a blank node: the creator of the article is neither Cicero nor Tiro, but some unnamed resource that is a sequence of those two authors.

For the rest of this chapter, we consider RDF as plain data without much consideration for concepts of interpretations, models, entailment, etc. However, as holds for SPARQL, the discussed labeling could as well be applied on an entailment graph containing inferred (rather than explicitly stated) triples.

It is also worth pointing out that all graphs we consider in this chapter are directed and we use "adjacent" and "reachable" accordingly. That is, a node m' is adjacent to m

if there is an edge from m to m' (the other direction does not matter), m' is reachable from m if there is a directed path from m to m'.

Fig. 2.2 Exemplary RDF Graph: RDF Conference Data



This RDF data is mapped to a graph as shown in Figure 2.2. There is a single node in the graph for each named resource that occurs in the RDF data. The same literal may occur multiple times. Each blank node is depicted as a rectangular node (e.g., 6). As in Turtle [3] and SPARQL, blank nodes are labeled with local identifiers prefixed by `_:`. There is one node for each blank node in the RDF data, though the graph does not need to be lean.

2.2.1 Queries on RDF Graphs

We consider three types of queries on RDF graphs, all sub-languages of SPARQL. The first and simplest type, called **1-SPARQL queries**, corresponds to single, ground SPARQL triple patterns. For instance, the triple pattern

```
1 ulp:cicero vcard:FN 'M. T. Cicero'
```

matches a given graph if the given triple occurs in the graph, i.e., if there is a `vcard:FN` labeled edge from `ulp:cicero` to a literal node with label `M. T. Cicero`.

The second type of queries, called **A-SPARQL queries**, are essentially acyclic conjunctive queries on RDF and correspond to acyclic SPARQL basic graph patterns. For instance, the basic graph pattern[4]

[4] We use the variant syntax for SPARQL discussed in [37] to ease the presentation.

```
?a rdf:type bib:Article
  AND ?a dc:creator ?p
  AND ?p vcard:FN 'M. T. Cicero'
```

selects from a given graph all articles created by someone with the full-name "M. T. Cicero" and binds the article to `?a` and the creator to `?p`. We only allow acyclic basic graph patterns:

Definition 1 (Acyclic Basic Graph Pattern). *Let P be a basic SPARQL graph pattern. Then P is acyclic, if no variable in P depends directly or indirectly on itself.*

We say a variable ?x depends (directly) on a variable ?y (a resource R) if there is a triple pattern ?x P ?y (?x P R) for some property P. We say a variable ?x depends indirectly on a variable ?y if there are variables or resources $z_1, \ldots, z_l$ such ?x depends on z_1, z_1 on $\ldots$, and z_l depends on ?y.

The most common case of acyclic graph patterns are tree patterns, where the variables of the query form a proper hierarchy.

The third type of queries are full SPARQL graph patterns. They may be cyclic and include **`UNION`**, **`MINUS`**, **`OPTIONAL`**, **`FILTER`**. We use the variant syntax for SPARQL discussed in [37] to ease the definition of syntax and semantics of the language. We omit named graphs and assume that all queries are on the single input graph. An extension of the discussion to named graphs is easy (and partially demonstrated in [39]) but only distracts from the salient points of the discussion.

The full grammar of SPARQL graph patterns considered here is as follows:

⟨*pattern*⟩ ::= ⟨*triple*⟩ | '{' ⟨*pattern*⟩ '}'
| ⟨*pattern*⟩ 'FILTER' '(' ⟨*condition*⟩ ')' |
| ⟨*pattern*⟩ 'AND' ⟨*pattern*⟩ | ⟨*pattern*⟩ 'UNION' ⟨*pattern*⟩
| ⟨*pattern*⟩ 'MINUS' ⟨*pattern*⟩ | ⟨*pattern*⟩ 'OPT' ⟨*pattern*⟩

⟨*triple*⟩ ::= ⟨*resource*⟩',' ⟨*predicate*⟩',' ⟨*resource*⟩

⟨*resource*⟩ ::= ⟨*iri*⟩ | ⟨*variable*⟩ | ⟨*literal*⟩ | ⟨*blank*⟩

⟨*predicate*⟩ ::= ⟨*iri*⟩ | ⟨*variable*⟩

⟨*variable*⟩ ::= '?' ⟨*identifier*⟩

⟨*condition*⟩ ::= ⟨*variable*⟩ '=' ⟨*variable*⟩ | ⟨*variable*⟩ '=' (⟨*literal*⟩|⟨*iri*⟩)
| 'BOUND(' ⟨*variable*⟩ ')' | 'isBLANK(' ⟨*variable*⟩ ')'
| 'isLITERAL(' ⟨*variable*⟩ ')' | 'isIRI(' ⟨*variable*⟩ ')'
| ⟨*negation*⟩ | ⟨*conjunction*⟩ | ⟨*disjunction*⟩

⟨*negation*⟩ ::= '¬'⟨*condition*⟩

⟨*conjunction*⟩ ::= ⟨*condition*⟩ '∧' ⟨*condition*⟩

⟨*disjunction*⟩ ::= ⟨*condition*⟩ '∨' ⟨*condition*⟩

We pose some additional syntactic restrictions: SPARQL graph patterns must be *error-free* SPARQL expressions, i.e., for each **`FILTER`** expression all variables occurring in the (right-hand) condition must also occur in the (left-hand) pattern. This can easily be ensured a-priori and queries violating this condition rewritten to the canonical false **`FILTER`** expression (as **`FILTER`** expressions with unbound variables raise errors which, in turn, are treated as a false filter, see "effective boolean value" in [40]).

Following [39], we define the semantics of SPARQL graph patterns based on *substitutions*. A substitution $\theta = \langle v_1, n_1, \ldots, v_k : n_k \rangle$ with $v_i \in \mathsf{Vars}(Q) \wedge n_i \in \mathit{nodes}(D)\}$ for

a query Q over an RDF graph D maps some variables from Q to nodes in D. For a substitution θ we denote with $dom(\theta)$ the variables mapped by θ. Given a triple pattern $t = (s, p, o)$, we denote with $t\theta$ the application of θ to t replacing all occurrences of variables mapped in θ by their mapping in t. For a triple (s, p, o) containing no variables, we say $(s, p, o) \in D$ if there is a p labeled edge between s and o labeled nodes in D.

On sets of substitutions the usual relational operations $\bowtie$, $\cup$, and $\setminus$ apply. We define the (left) semi-join $R \ltimes S = (R \bowtie S) \cup (R \setminus S)$.

$$
\begin{array}{ll}
[\![(s,p,o)]\!]^D_{\mathsf{Subst}} & = \{\theta : dom(\theta) = \mathsf{Vars}((s,p,o)) \wedge t\theta \in D\} \\
[\![pattern_1 \ \texttt{AND} \ pattern_2]\!]^D_{\mathsf{Subst}} & = [\![pattern_1]\!]^D_{\mathsf{Subst}} \bowtie [\![pattern_2]\!]^D_{\mathsf{Subst}} \\
[\![pattern_1 \ \texttt{UNION} \ pattern_2]\!]^D_{\mathsf{Subst}} & = [\![pattern_1]\!]^D_{\mathsf{Subst}} \cup [\![pattern_2]\!]^D_{\mathsf{Subst}} \\
[\![pattern_1 \ \texttt{MINUS} \ pattern_2]\!]^D_{\mathsf{Subst}} & = [\![pattern_1]\!]^D_{\mathsf{Subst}} \setminus [\![pattern_2]\!]^D_{\mathsf{Subst}} \\
[\![pattern_1 \ \texttt{OPT} \ pattern_2]\!]^D_{\mathsf{Subst}} & = [\![pattern_1]\!]^D_{\mathsf{Subst}} \ltimes [\![pattern_2]\!]^D_{\mathsf{Subst}} \\
[\![pattern \ \texttt{FILTER} \ condition]\!]^D_{\mathsf{Subst}} & = \{\theta \in [\![pattern]\!]^D_{\mathsf{Subst}} : \mathsf{Vars}(condition) \subset dom(\theta) \\
 & \qquad \wedge [\![condition]\!]^D_{\mathsf{Bool}}(\theta)\}
\end{array}
$$

$$
\begin{array}{ll}
[\![condition_1 \wedge condition_2]\!]^D_{\mathsf{Bool}}(\theta) & = [\![condition_1]\!]^D_{\mathsf{Bool}}(\theta) \wedge [\![condition_2]\!]^D_{\mathsf{Bool}}(\theta) \\
[\![condition_1 \vee condition_2]\!]^D_{\mathsf{Bool}}(\theta) & = [\![condition_1]\!]^D_{\mathsf{Bool}}(\theta) \vee [\![condition_2]\!]^D_{\mathsf{Bool}}(\theta) \\
[\![\neg condition]\!]^D_{\mathsf{Bool}}(\theta) & = \neg [\![condition]\!]^D_{\mathsf{Bool}}(\theta) \\
[\![\texttt{BOUND}(?v)]\!]^D_{\mathsf{Bool}}(\theta) & = v\theta \neq \texttt{nil} \\
[\![\texttt{isLITERAL}(?v)]\!]^D_{\mathsf{Bool}}(\theta) & = v\theta \in L \\
[\![\texttt{isIRI}(?v)]\!]^D_{\mathsf{Bool}}(\theta) & = v\theta \in I \\
[\![\texttt{isBLANK}(?v)]\!]^D_{\mathsf{Bool}}(\theta) & = v\theta \in B \\
[\![?v = literal]\!]^D_{\mathsf{Bool}}(\theta) & = v\theta = literal \\
[\![?u = ?v]\!]^D_{\mathsf{Bool}}(\theta) & = u\theta = v\theta \wedge u\theta \neq \texttt{nil}
\end{array}
$$

Table 2.2. Semantics for SPARQL graph patterns

Using these definitions, Table 2.2 gives the semantics of SPARQL graph patterns by means of $[\![\ ]\!]^D_{\mathsf{Subst}}$. It produces a set of substitutions (or bindings) for variables in P. Triple patterns t (case 1) are evaluated to the set of substitutions θ such that the $t\theta$ contains no more variables and falls in D. Pattern compositions **AND**, **UNION**, **MINUS**, and **OPT** are reduced to the appropriate operations on sets of substitutions (cases 2–4). **FILTER** expressions (case 5) are again evaluated straightforwardly, as restrictions on the substitutions returned by the (left-hand) pattern with the boolean formula that is provided by $[\![\ ]\!]^D_{\mathsf{Bool}}$ for the condition of the filter expression. $\mathsf{Vars}(condition) \subset dom(\theta)$ is not strictly necessary as it merely restates that we only consider error-free SPARQL queries.

The semantics of A-SPARQL and 1-SPARQL patterns are the obvious specializations of the semantics for full SPARQL patterns. For 1-SPARQL patterns this yields a

boolean semantics, with {} as false and {∅} as true (the empty substitution is a solution, if $t\emptyset = t \in D$).

2.2.2 Triple Patterns and Adjacency

Triple patterns as in 1-SPARQL are a form of adjacency test between the two named resources (or the named resource and the literal) in the given triple pattern. In this chapter, we often prefer to speak of adjacency test to emphasize the general nature of CIG-labelings. In particular, whether the edges are labeled or not has no impact on the labeling: If the number of edge labels is fixed and small, we can choose different CIG-labelings for each of the induced sub-graphs. Otherwise, we can, e.g., choose to represent edges as special nodes (connected to its source and sink) with a label property for storing the edge label. We can also choose a mixture of both approaches, e.g., to use separate CIG-labelings for each of the k most used edge labels and the second approach for all remaining edge labels.

It is worth noting that this holds, more or less, for most labeling schemes and thus allows us to consider also labeling schemes intended for un-labeled graph data for RDF, see Section 2.3.2.

Though not part of SPARQL, reachability is also a property that is often useful in graph queries. There are already proposals for extending SPARQL with reachability or even full path expressions [38, 9]. Here, we only mention in passing that CIG-labelings can handle reachability testing just as well as adjacency testing. For more details see Section 2.6.4.

2.3 State-of-the-Art—Labeling Schemes for RDF Graphs

Though RDF and, in particular, SPARQL are fairly new technologies, there is significant relevant related work on labeling schemes for graph data in general. Though little of that has been specifically considered for RDF, most is easily adapted to RDF.

Before turning to graph labeling schemes, we briefly revisit labeling schemes for *tree data*, as most graph labeling schemes are extensions of one of the tree labeling schemes. Moreover, tree labeling schemes have seen significant attention from academia and implementation in most XML-enabled relational databases.

2.3.1 Foundation: Tree Labeling

Labeling schemes of XML data fall into one of three categories: interval-based, prefix-based, and arithmetic. *Interval-based labeling* schemes order the nodes of the tree in one or more orders and describe the structural relations (such as child, descendant, etc.) as windows over the assigned orders. For instance, the pre-/post-encoding (first proposed in [15] and adapted for use in XML querying in [22, 6]) assigns each node its position in pre- and post-order traversal. The descendants of a node are then all nodes with higher pre- and lower post-order number. With an additional label for the nesting level, all 13 XPath axes can be expressed analogously. *Prefix-based labeling* schemes

such as the Dewey ID-based ORDPATH [35] (used in Microsoft SQL server for XML storage) assign each node a label, such that the label of a parent is the prefix of the labels of all its children. For instance, if the parent is labeled 1.7.2, its children are labeled 1.7.2.x for some (positive or negative integer x). Notable is that ORDPATH leaves gaps in the assignment of the child portions of a label to allow updates without relabeling the entire tree. However, prefix-based labeling schemes are, in general, best suited for rather regular, flat trees. If the tree becomes too deep or fan-outs vary significantly the label sizes quickly degenerate (up to $O(n)$ where n is the number of nodes in the tree). Finally *arithmetic labeling* schemes (such as BIRD [45]) use some kind of arithmetic relation between labels to express child-parent, ancestor-descendant relationships. For example, prime numbers are assigned to the leaf nodes in the tree and labels of parent nodes become the product of the labels of their child nodes. Thus testing whether a node is an ancestor of another one subsumes to testing whether the label of the first node is a multiple of the label of the second node. Here we focus primarily on interval-based labeling schemes, as these have been extended to graph data and are particularly well suited for large data sets (where prefix-based labeling schemes often fail due to the large label size and arithmetic labeling schemes often fail due to the high computational cost for computing a labeling). For a more detailed comparison of labeling schemes for XML see [42].

Aside of labeling schemes as separate indices (often for relational storage and access of XML), it is also worth to briefly consider different approaches for XML query evaluation as these often exploit properties of tree data similar to those exploited by interval-based labelings:

There are five approaches to XML query evaluation that are particularly relevant for a comparison with interval-based labeling schemes. The time and space complexity for evaluating various classes of queries (path, tree, and graph queries) on various classes of data (tree, DAG, graph) are given in Table 2.3. Note, that in all cases we consider pointers of constant size (as in most related work [8], [31], and [30]). In fact, all approaches need an additional $\log n$ multiplicative factor if pointer size is taken into consideration. The assumption of constant pointer (or label) size is reasonable in practice, as these are usually stored in attributes of a reasonably sized, constant length numerical type such as SQL's `INT` or `BIGINT`.

The first approach, dubbed *structural joins* [2], is predominantly used in relational storage of XML. It uses some form of labeling scheme, here pre-/post-encoding as in [6], to find nodes that fulfill each of the structural conditions of the query and joins the result (either using standard relational joins or using a tree-aware join such as the staircase join [23]). This results in a very flexible evaluation technique (it can, e.g., deal with all 13 XPath axes), but is not quite as space efficient as more specialized approaches. Though structural join approaches can be easily extended to graph data, they do not perform well on graph data. CIG-labelings can be easily integrated into structural join approaches, but as we illustrate in Section 2.6.2 a specialized interval join yields even better results (linear time and *space* processing).

In contrast to structural join approaches, twig (or stack) join approaches [8, 28, 11, 12] use a single ("holistic") operator for matching tree patterns. They are limited to tree and DAG data and only consider child and descendant relations. The basic idea of twig

approach	query	data	time	space
Structural Joins [2, 6]	path	tree	$O(n^{q_a} + q \cdot n \cdot \log n)$	$O(n^{q_a} + q \cdot n^2)$
	tree	tree/DAG	$O(n^{q_a} + q \cdot n \cdot \log n)$	$O(n^{q_a} + q \cdot n^2)$
	graph	tree	$O(n^q)$	$O(n^q)$
	graph	graph	$O(n^q)$	$O(n^q)$
Twig or Stack Joins [8]	tree (c/d)	tree	$O(q \cdot n)$	$O(q \cdot n + n \cdot d)$
	tree (c/d)	DAG	$O(q \cdot n^2 + e)$	$O(q \cdot n + e)$
SPEX (streaming) [32, 31]	tree	tree	$O(q \cdot n^2)$	$O(q \cdot n \cdot d)$
CAA [30]	tree	tree	$O(q \cdot n \cdot \log n \cdot d)$	$O(q \cdot n)$
CIG-labeling & Interval Join	tree (c/d)	tree	$O(q \cdot n)$	$O(q \cdot d)$
	tree	tree, CIG	$O(q \cdot n)$	$O(q \cdot n)$
	tree	graph	$O(q \cdot n^2)$	$O(q \cdot n^2)$
	graph	tree, CIG	$O(n^{q_g} + q \cdot n)$	$O(n^{q_g} + q \cdot n)$
	graph	graph	$O(n^{q_g} + q \cdot n^2)$	$O(n^{q_g} + q \cdot n^2)$

Table 2.3. Comparison of Related Approaches. n: number of nodes in the data, d: depth, resp. diameter of data; e: number of edges; q: size of query, q_a: number of result or answer variables; q_g: number of "graph" variables, i.e., variables with multiple incoming query edges

join approaches is the use of one stack per query variable (or XPath step) containing nodes that may be matches for that variable. Relations between such potential matches are established in form of explicit parent pointers. However, since queries may only contain child and descendant relations, at most d such parent pointers may exist per node. The essential observation from twig joins is that, we need to limit the space (and management overhead) for describe the relations between potential matches of different variables. In twig joins, the combination of tree data and tree queries with only child and descendant relations together with an efficient stack management ensures this property. Similarly, the evaluation of A-SPARQL using a specialized interval join as discussed in Section 2.6.2 uses CIG-labelings to represent relations and ensures that, if the data is a tree or CIG, the relations between potential matches can always be represented by a linear space CIG labeling. Thus, on tree and CIG data CIG-labelings with the interval join are actually more space efficient and as fast as twig joins on tree data, yet are not limited to only child and descendant relations.

Twig joins can be considered a hybrid of in-memory and streaming evaluation approach. Specialized streaming engines for XML query evaluation, such as SPEX [33], also often use stacks for collection potential matches. In contrast to twig joins, SPEX buffers nodes centrally and manages only annotations (or conditions) that represent relations among potential matches in the stacks of its transducers (roughly, one for each query variable). SPEX compacts these annotations into intervals where possible, but (partially due to its streaming nature) can not always achieve this compaction as the CIG-labeling scheme can on tree and CIG data. SPEX can process all forward [34] XPath axes, but is limited to tree data.

The approach closest in spirit to ours are the complete answer aggregates (CAA) [30]. Like twig joins and other polynomial approaches to XPath tree query processing (e.g., [20]), the (potential) answers for each query variable are stored separately. Unlike the other approaches (and like CIG-labelings) the relations between these potential answers may be described using intervals (rather than lists of pointers or keys). CAAs are unique among the discussed approaches in that they always store answers for all query variables (therefore *complete* answer aggregates), which is helpful in the context of query refinement and exploration of the database, but otherwise often undesirable.

2.3.2 Reachability in Graphs

When we turn from tree data to graph data, the use of labeling schemes becomes less predominant. However, there have been a number of approaches for exploiting labeling schemes to provide an efficient test for *reachability* in graph data. For RDF, as discussed in [13] and [43], this is particularly relevant as these approaches allow subsumption in RDF. Though SPARQL does not provide specific means for reachability queries, several extensions for SPARQL, most notably nSPARQL [38], have been proposed that include such support. Furthermore, SPARQL can be used to query also, e.g., the RDFS entailment (rather than the raw) graph where the subsumption is expanded. For either case labeling schemes can be exploited: In the latter case, we can use the CIG-labeling scheme as for adjacency testing (on the expanded relation). However, it is often preferable not to expand the reachability relation a-priori. We call this case *ad-hoc reachability test.*

Considerable research on indexing arbitrary graph data for ad-hoc reachability testing has basically fallen into two classes. Table 2.4 summarizes the most relevant approaches for comparison with the CIG-labeling. As baselines, we also include the naive storage of the full reachability matrix as well as the online (shortest path) computation with no index at all. For large graphs, neither of these baseline approaches is feasible. Therefore, two classes of approaches have been developed that allow with significantly lower space to obtain sub-linear time for membership test:

The first class is based on the idea of a *2-hop cover* [14]: Instead of storing a full reachability matrix, we allow that reachable nodes are reached via at most one other node (i.e., in two "hops"). More precisely, each node n is labeled with two connection sets, $in(n)$ and $out(n)$. $in(n)$ contains a set of nodes that can reach n, $out(n)$ a set of nodes that are reachable from n. Both sets are assigned in such a way, that a node m is reachable from n iff $out(n) \cap in(m) \neq \emptyset$. Unfortunately, computing the optimal 2-hop cover is NP-hard. Approximation algorithms [41] that can provide very good guarantees have been developed, but still require rather significant time for index computation.

A different approach [1, 10, 44, 43] is to use *interval encoding* for labeling a spanning tree of the graph and treating the remaining non-tree edges separately. This allows for sublinear or even constant membership test, though constant membership test incurs lower but still considerable indexing cost, e.g., in Dual Labeling [44] where a full transitive closure over the non-tree edges is build. GRIPP [43] and SSPI [10] use a different trade-off by attaching additional interval labels to non-tree edges. This leads to linear index size and time at the cost of increased query time.

In comparison, the CIG-labeling combines many of the best characteristics of these approaches: It is even better suited to sparse, tree-like graphs than interval encoding

approach	reachability test time	index time	index size
No index [36]	$O(n+e)$	$O(n+e)$	$O(n+e)$
Full Reachability Matrix	$O(1)$	$O(n^3)$	$O(n^2)$
2-Hop [14]	$O(\sqrt{e}) \leq O(n)$	$O(n^4)$	$O(n \cdot \sqrt{e})$
HOPI [41]	$O(\sqrt{e}) \leq O(n)$	$O(n^3)$	$O(n \cdot \sqrt{e})$
Graph labeling [1]	$O(n)$	$O(n^3)$	$O(n^2)$
SSPI [10]	$O(e-n)$	$O(n+e)$	$O(n+e)$
Dual labeling [44]	$O(1)$	$O(n+e+e_g^3)$	$O(n+e_g^2)$
GRIPP [43]	$O(e-n)$	$O(n+e)$	$O(n+e)$
CIG labeling on trees,CIGS	$O(1)$	$O(e)$	$O(n)$
—- on arbitrary graphs	$O(\log(1.5 \cdot c_{\mathsf{opt}})) < O(\log n)$	$O(n^3)$	$O(1.5 \cdot i_{\mathsf{opt}}) < O(n^2)$

Table 2.4. Cost of Reachability Test in Graph Data. n, e: number of nodes, edges in the data, e_g: number of non-tree edges, c_{opt} the number of intervals needed to cover the children of a single node in an optimal CIG-labeling and i_{opt} the total sum of these intervals.

approaches (as it provides guaranteed constant reachability test not only for trees but also for CIGS). At the same time, we can give very strong guarantees for maximum time (and space) for reachability testing on general graphs (see Section 2.4.3). These guarantees come with an increased indexing time ($O(n^3)$), yet we can also choose to spend less time indexing and choose a rougher heuristics.

2.4 CIG-Labeling Scheme

The CIG-labeling scheme is designed around a generalization of interval labels, called here CIG-labeling. CIG-labelings provide flexible, yet simple description of arbitrary relations with attractive properties: In essence, we map each node to a single (integer) label and describe the children of each node as a set of intervals over the above mapping.

Definition 2 (CIG-Labeling). *Let $G = (N, E)$ be an (arbitrary) graph. Then a* CIG*-labeling $\mathcal{L}$ of G, is a pair $(l, \mathcal{I})$ such that*

1. *$l : N \to \mathbb{N}_{|N|}$ is a bijective labeling function that assigns to each node in G an (integer) label from $\{1, \ldots, |N|\} = \mathbb{N}_{|V|}$.*
2. *$\mathcal{I} : N \to 2^{\mathbb{N}_{|N|} \times \mathbb{N}_{|N|}}$ is a mapping from nodes in G to sets of closed, non-empty, non-overlapping, non-adjacent intervals over $\mathbb{N}_{|N|}$.*
3. *$\mathcal{I}(n)$ covers all adjacent nodes of n for each $n \in N$, i.e.,*

$$\{n' \in N : (n, n') \in N\} = \{n' \in N : \exists [s, e] \in \mathcal{I}(n) : s \leq l(n') \leq e\}.$$

For an interval $[a, b]$ we say that $[a, b]$ "covers" the nodes with $a \leq l(n) \leq b$. Recall, that an interval $[a, b]$ is closed, if their start and end point is part of the nodes covered by the interval, and non-empty, if $a \leq b$ and thus at least one node is covered by the interval. A pair of intervals $[a, b]$ and $[c, d]$ are called non-overlapping, if no node is

covered by both intervals, and non-adjacent if neither $b = c$ nor $d = a$. Thus in a set of closed, non-empty, non-overlapping, and non-adjacent intervals each index occurs in no more than one interval (there is no start- or endpoint that is start- or endpoint of another interval).

Compared to other interval labelings, it is worth pointing out that a CIG-labeling uses only a single integer label per node, but at the cost of requiring explicit descriptions of the children intervals. The often used pre-/post-encoding [22] for trees, e.g., uses three such labels (the pre- and the post-numer as well as the level of the node), but does not need to explicitly store the interval boundaries (they follow from the labels) and can cover multiple relations with these three labels.

However, the main difference is that most labeling schemes assign labels in a predetermined way (e.g., by means of a pre- and post-order traversal of the tree). The CIG-labeling, on the other hand, allows us to choose rather flexibly among possible orderings (and thus labels) of the node. Thus, the CIG-labeling scheme consists in the CIG-labeling together with an algorithm for generating that labeling. We will suggest such algorithms for general graphs (Section 2.4.3), as well as trees and CIGS (Section 2.5).

To emphasize this flexibility, it is worth explicitly stating that for each graph can be labeled by some CIG-labeling:

Theorem 1. *For any graph G there exists a* CIG*-labeling.*

Proof. Let $G = (N, E)$. Then we assign labels from $\mathbb{N}_{|N|}$ to the nodes in N arbitrarily. Given the resulting labeling function l, we set $\mathcal{L}(n)$ to the set of all intervals $[l(n'), l(n')]$ where $(n, n') \in E$. The resulting intervals are trivially closed, non-overlapping and non-empty. As long as there are still adjacent intervals in any of the interval sets, we merge those adjacent intervals into a single interval. This terminates after at most n merge steps per node resulting in interval sets that are also non-adjacent.

The challenge is finding an optimal labeling for our purpose. For querying a single relation, we call a CIG-labeling *optimal*, if the total size of the interval labels is minimal (the integer labels are always the same size).

Definition 3 (Optimal CIG-Labeling). *Let $G = (N, E)$ be a graph and $\mathcal{L}$ a* CIG*-labeling for G. We call $\sum_{n \in N} |\mathcal{I}(n)|$ the size of $\mathcal{L}$, denoted $|\mathcal{L}|$. $\mathcal{L}$ is optimal for G iff there is no other* CIG*-labeling $\mathcal{L}'$ for G with $|\mathcal{L}'| < |\mathcal{L}|$.*

It turns out that for trees we can define several labeling strategies that yield a CIG-labeling with similar properties as a pre-/post-encoding, most notably with linear size for the entire labeling (and constant per node size). With a slight extension it is also possible to define a CIG-labeling that can cover all forward XPath axes in one labeling.

In contrast to tree labelings such as pre-/post-encoding, the CIG-labeling is, however, flexible enough to be used also for many non-tree graphs. It turns out that it can even provide constant reachability test for a significantly larger class of graphs than trees (at linear size), see Section 2.5. Even on arbitrary graphs, we can still profit from a CIG-labeling compared to, e.g., a full (quadratic) reachability matrix or previous graph labeling schemes (see Section 2.3.2 for an overview). Though finding the optimal CIG-labeling is NP-complete for general graphs, a polynomial 1.5 approximation exists that gives very compact representations in most practical cases.

Before we turn to the question *how* to compute an optimal CIG-labeling, we first summarize the main properties of a given CIG-labeling (whether optimal or not), namely its label size (Section 2.4.1 and the complexity of its adjacency test (Section 2.4.2).

2.4.1 Label Size

The first property to be investigated is the size of a CIG-labeling. It is easy to give an upper bound for arbitrary graphs:

Theorem 2. *For any given graph G with n nodes and e edges, a* CIG*-labeling uses exactly n integer labels and $O(e)$ intervals.*

Proof. From the definition it follows directly that each node is associated with an integer label. Furthermore, in the worst case each child of every node is covered by a single interval (i.e., no two children are adjacent in the order of the nodes). Thus $O(e)$ total intervals are used. This is indeed the worst-case as intervals are closed non-empty, non-overlapping, and non-adjacent.

We can also characterize the space complexity in terms of space used per node:

Proposition 1. *For any given graph G with n nodes and e edges, each node m requires a single integer label and an interval set label containing at most* $\max(\mathsf{out\text{-}degree}(m), \frac{1}{2}n)$ *intervals.*

Proof. Again for the integer label this follows directly from the definition. For the interval set label observe that m has $\mathsf{out\text{-}degree}(m)$ children that need to be covered by an interval. Since all intervals are empty and non-overlapping, each does cover a node that is not covered by any of the others and thus we can cover all $\mathsf{out\text{-}degree}(m)$ children with at most $\mathsf{out\text{-}degree}(m)$ intervals. Furthermore, $\frac{1}{2}n$ is an upper bound for the number of intervals per node, as if $\mathsf{out\text{-}degree}(m) > \frac{1}{2}n$ $\mathsf{out\text{-}degree}(m) - \frac{1}{2}n$ of the children of m must have an integer label consecutive with one or more of the other children in any order over the nodes. Thus no additional intervals are needed for these nodes.

The above are the label sizes for the adjacency relation on a given graph G. If we consider the reachability relation on G the same considerations apply but wrt. the reachability graph G' of G. Thus the number of nodes remains the same, but the number of edges increases to the number of pairs of reachable nodes which is, even for sparse graphs, often close to $O(n^2)$.

Fortunately, this is in practice partially offset by the observation (see Section 2.5.1) that both graphs with sparse and almost complete graphs are more likely CIGS than graphs that are neither.

2.4.2 Adjacency & Reachability Test

Given a CIG-labeling (for the original graph or for the reachability graph), we can test adjacency and reachability fairly efficiently:

Theorem 3. *Let G be a graph with n nodes. Given any* CIG*-labeling for G and two nodes m, m' we can test whether there is an edge from m to m' in $O(\log i_m)$ where i_m is the number of intervals in the interval set label of m ($i_m \leq \max(\mathit{out\text{-}degree}(m), \frac{1}{2}n)$ by Proposition 1).*

Proof. When constructing a CIG-labeling, we can easily assure that the intervals in each interval set are stored ordered by start position (and thus by end position since they are non-overlapping), e.g., by sorting them. Then testing whether m' is a child of m subsumes to looking up $l(m')$ in constant time, looking up $\mathcal{I}(m)$, performing a binary search on the sorted intervals in $\mathcal{I}(m)$, see Figure 7. Obviously, the latter runs in $\log i_m$.

Algorithm 1: Binary Search in Interval Set

input : nodes m, m', CIG-labeling $(l, \mathcal{I})$
output: true if $(m, m') \in E$, false otherwise

```
1 first ← 1; last ← |I(m)| ;
2 while first ≤ last do
3 |   mid ← (first+last)/2; [s, e] ← I(m)[mid] ;
4 |   if l(m') < s then last ← mid − 1 ;
5 |   else if l(m') > e then first ← mid + 1 ;
6 |   else return true;
7 return false;
```

Again the same applies for reachability testing but on the reachability graph G' for G. Note, that the out-degree of a node is affected by the step from G to G', but not the number of nodes. Thus in both cases $\log \frac{n}{2}$ is an upper bound.

In addition to testing adjacency and reachability, the iteration over the adjacent or reachable nodes of a given node is an important operation for evaluating SPARQL. In the optimal case, this iteration depends only on the number of these nodes and not on the (size, shape, etc.) of the remaining graph. With a CIG-labeling we obtain this optimal case, simply by iterating over the ordered intervals in the interval set of each node:

Corollary 1. *Let G be a graph with n nodes. Given any* CIG*-labeling for G and a node m we can iterate over the c_m adjacent (reachable nodes) of m in $O(c_m)$.*

2.4.3 Optimal CIG-Labeling

So far we have only investigated the properties of CIG-labelings in general, without considering how to compute a "good" or even optimal CIG-labeling. For an arbitrary graph, it turns out that computing an optimal CIG-labeling is hard:

Theorem 4. *Let G be an arbitrary graph. Then computing an* optimal CIG*-labeling for G is* NP*-complete.*

Proof. In particular, it is NP-complete to determine whether there is a CIG-labeling $\mathcal{L}$ for G with $|\mathcal{L}| \leq k$.

Obviously we can find such a CIG-labeling by guessing a suitable order and verifying that $\mathcal{L} \leq k$ in linear time.

NP-hardness is established by reduction from the consecutive block minimization (or CBM) problem for binary matrices, first introduced in [29]. Following [18], the consecutive block minimization problem is the problem of computing a permutation of the columns of a binary matrix such that results in a matrix B with at most k consecutive blocks of 1's. In other words, B must not have more than k entries b_{ij} such that $b_{ij} = 1$ and either $b_{ij} = 0$ or $j = n$. The problem remains NP-hard for quadratic matrices as well as for sparse matrices.

A block of 1's in row i in CBM corresponds to an interval in the interval set of node $l^{-1}(i)$ in a CIG-labeling: Each of the nodes covered by the interval is a child of $l^{-1}(i)$ and thus the corresponding entry in the adjacency matrix of G is 1. As in CBM, we aim to find a permutation of the nodes that minimize the number of such intervals where the next node in the permutation is not a child of $l^{-1}(i)$, i.e., the corresponding entry in the adjacency matrix is 0.

Thus the reduction is very easy: We just take the quadratic matrix (for which the problem is still NP-hard) and consider it as adjacency matrix of the graph for which to find an optimal CIG-labeling. Given that labeling we can compute the matrix B as the adjacency matrix of the graph where the columns are ordered by l.

Though finding an *optimal* CIG-labeling is thus infeasible even for small graphs, [25] proposes an 1.5 approximation algorithm for CBM based on a (fairly straightforward) transformation to the traveling sales problem.

Corollary 2. *Let G be an arbitrary graph. Then we can compute, in polynomial time, a* CIG*-labeling with a size that does not differ more than* 50% *from the optimal one (i.e., it size is $1.5i_{opt}$ where i_{opt} is the optimal number of intervals for any* CIG*-labeling).*

More precisely, the computation is in $O(n^3)$ (it is dominated by the time for finding a maximum matching in a graph).

2.5 CIG-Labeling on Trees and CIGS

We have established that CIG-labelings on arbitrary graphs can give us logarithmic adjacency test and linear adjacency iteration, yet with optimal or near optimal CIG-labelings can have significantly smaller size than full adjacency lists or matrices.

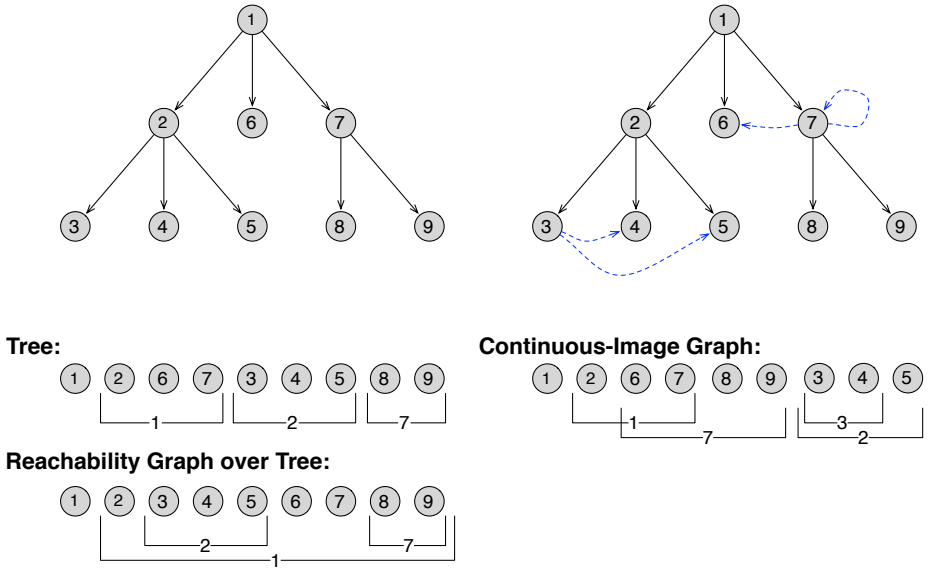
However, CIG-labelings have another desirable property: On certain graphs, including all trees and forest, but extending significantly beyond, we can test adjacency in constant time and require only constant labels (thus, linear total space).

In this section, we first characterize and illustrate the class of graphs where this property holds and then show how to compute a CIG-labeling with constant labels for this class.

2.5.1 CIGS: Sharing-Limited Graphs

In a sense, a graph with a CIG-labeling with constant labels (called a CIG) exhibits a certain regularity: It limits the way child nodes may be shared among parents such that two parents may share a node, but only if there is still a way to order the nodes such that the shared nodes of both parents are adjacent to its non-shared nodes. This can be seen as a generalization of similar restrictions on trees and the reachability graphs of trees: In trees, each child has at most one parent and thus there is no sharing of child nodes. In reachability graphs for trees (i.e., the graph for the transitive closure of the edges of a tree), if two nodes share a child, then one is a descendant of the other and thus all its children are shared with the other node. These differences are mirrored in the way intervals for covering the children of nodes in these three structures behave: Figure 2.3 illustrates how in a tree these intervals can be made not to overlap at all. In the reachability graph for a tree they may be contained in each other, but no other form of overlapping is allowed. In a CIG these interval may overlap arbitrarily, but they still must be single, continuous intervals. In an arbitrary graph the latter restriction is also lifted, as seen above.

Fig. 2.3 Sharing of children in trees, reachability graphs over trees, and CIGS



More formally, CIGS are a proper superset of trees defined as follows:

Definition 4 (CIG). *Let G be a graph. Then G is called a* continuous-image graph *(or* CIG*) if there is a* CIG*-labeling for G where all the interval set labels contain a single interval only.*

Thus, a graph $G = (N, E)$ is a CIG if there is

1. a bijective labeling function $l : N \to \mathbb{N}_{|N|}$ that assigns to each node in G an (integer) label from $\{1, \ldots, |N|\} = \mathbb{N}_{|V|}$,
2. a partial mapping from nodes in G to closed, non-empty intervals $\mathcal{I} : N \to \mathbb{N}_{|N|} \times \mathbb{N}_{|N|}$ and
3. $\mathcal{I}(n)$ covers all adjacent nodes of n for each $n \in N$, i.e.,

$$\{n' \in N : (n, n') \in N\} = \{n' \in N : \exists [s, e] = \mathcal{I}(n) : s \leq l(n') \leq e\}.$$

Essentially this is a slightly simplification of the conditions for a CIG-labeling where each node is assigned a single interval (or none).

In practice, we belief that CIGs are fairly common, in particular where hierarchical or mostly hierarchical ontologies as well as time-related (e.g., event) data is considered. If relations, e.g., between Germany and kings, are time-related, it is quite likely that there will be some overlapping, e.g., for periods where two persons were king of Germany at the same time. Similarly, hierarchical data often has some limited anomalies that make a modelling as strict tree data impossible. Both points are illustrated in Figure 2.1 where we show the relationships (both by law and by blood) between emperors of the "Five Good Emperors" (Edward Gibbon) period (roughly 2nd century A.D.) as well which provinces they ruled (provinces not shown have been ruled by all of these emperors).[5] Despite the rather complicated shape of the relations (they are obviously not tree-shaped and there is considerable overlapping, in particular w.r.t. province rulership) the whole graph is a CIG and can be stored with constant size labels and its adjacency can be queried in constant time.

2.5.2 Labeling CIGs and Trees

The crucial remaining questions are whether we can distinguish CIGs from graphs efficiently and how to compute a CIG-labeling for CIGs.

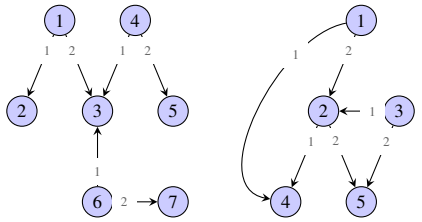
For answering both questions, we have to observe once more that there is a related problem from binary matrix theory (like the CBM for finding optimal CIG-labelings of arbitrary graphs): The consecutive ones property of binary matrix [16]. A binary matrix is said to carry the consecutive ones property if there is a permutation of the columns of that matrix such that the 1's in each row are consecutive. Again the block of 1's corresponds to the CIG-interval for the children of that row's node.

Theorem 5. *Let G be an arbitrary graph with e edges. Then determining whether G is a* CIG *and, if so, computing a* CIG*-labeling with constant label size for G is possible in $O(e)$.*

Proof. For the consecutive-ones problem [7] gives the first linear time (in the number of nodes) algorithm based on so called PQ-trees, a compact representation for permutations of rows in a matrix. More recent refinements in [24] and [27] show that simpler

[5] The name and status of the province between the wall of Hadrian and the wall of Antonius Pius in northern Britain is controversial. For simplicity, we refer to it as "Caledonia", though that actually denotes all land north of Hadrian's wall.

Fig. 2.4 Limits of Continuous-image Graphs



algorithms, based on the PC-tree [26], can be achieved. To decide whether a given graph is a CIG we use the PC-tree algorithm on that graph's adjacency matrix which runs in $O(e)$. If the matrix carries the consecutive ones property the graph is a CIG and any of the consecutive ones order in the PC-tree can be used to label the nodes of the graph. The intervals are then computed accordingly and we obtain a single interval per node (thus the node label is constant, one integer label and one interval). Both is possible in $O(e)$.

2.5.3 Properties of CIG-labelings on CIGS and Trees

The previous section establishes that CIGS can be characterized and labeled in linear time. However, we have yet to note the precise time and space complexities for adjacency/reachability testing with a CIG-labeling. The same complexities as for CIGS hold also for trees (which are, after all, a special case of CIGS) and are as good as those of the best known complexities for tree-only labeling schemes such as the pre-/post-encoding.

Theorem 6. *Let G be a* CIG *(or tree, forest, reachability graph over a tree) with n nodes and $\mathcal{L}$ a* CIG*-labeling computed as in Theorem 5. Then the size of each node label is constant and thus the overall space complexity of the* CIG*-labeling $O(n)$. Furthermore, testing the adjacency of two nodes in G is possible in constant time.*

Proof. The labels are constant by the construction in Theorem 5: A single interval is needed to represent all children, as all children are consecutive in the computed order. For testing the adjacency of two nodes m, m' in G it suffices to test if $\{[s,e]\} = \mathcal{I}(m) \wedge s \leq l(m) \leq e$.

2.5.4 Limitations and Extensions

Though CIGS already cover many practical graphs, it is still worth considering the limits of this class. There are even (fairly) simple graphs that are no CIGS, see Figure 2.4. The figure shows two graphs that are *no* CIGS. Incidentally, both graphs are acyclic and, if we take away any one edge in either graph, the resulting graph becomes a CIG. The first illustrates an easy to grasp sufficient but not necessary condition for being a non-CIG: if a node has at least three parents and each of the parents has at least one (other) child not shared by the others then the graph can not be a CIG.

To overcome these limitations, CIG-labelings can be extended in at least three aspects while retaining the constant label size and constant adjacency test (for more details see [46]):

1. We can allow more allow more than one, but still a constant number of k intervals per node. It turns out, however, that already for $k = 2$ computing an optimal labeling is NP-hard [19].
2. Another promising extension is to allow multiple integer labels per node, i.e., to consider intervals over multiple orders. In some sense, this extensions generalizes intervals over one dimension to those over multiple dimensions. However it has been shown in [46] recently (by reduction from k-colorability) that finding an optimal labeling for this extension also becomes NP-hard quickly (in fact, already for 2 orders).
3. Localizing maximum CIG subgraphs also turns out to be an intractable problem (cf. consecutive ones submatrix [18]).

Though these extensions cover additional graphs, we would, once more, have to rely on heuristics or approximations for finding optimal labelings. This strongly indicates that CIGS are very much a "sweet spot" wrt. size of the class and complexity of the labeling algorithm.

2.6 Evaluating SPARQL with CIG-Labelings

The previous sections introduce the CIG-labeling, its properties on arbitrary graphs, on CIGS, the novel class of graphs with constant time adjacency test, and on trees. In this section, we use these results to sketch how to evaluate queries in the three SPARQL dialects introduced in Section 2.2 efficiently.

For all three dialects, the first issue to address is how to label an RDF graph. Node labels and typed nodes (the distinction between literals, URIs, and blank nodes) are easily represented in a CIG-labeling. However, we have to consider what to do with edge labels which are not present in the plain directed graphs used so far for defining the CIG-labeling. There are two possibilities to dealing with edge labels:

1. The RDF graph can be transformed by introducing a new node of type edge-node n_l for each node n and edge label l if n has outgoing edges labeled with l. We also add edges (n, n_l) and one outgoing edge to each n' with (n, l, n') in the original graph. Finally, we delete the edge (n, l, n'). The resulting graph has only unlabeled edges and we can rewrite each query (s, p, o) to (s, p), (p, o), and edge-node(p). This representation is obviously linear in the number of triples in the graph.
2. The graph induced by the edges of each edge label can be considered separately and queries adapted to query the appropriate graph.

The first approach has advantages if the number of edge labels (i.e., properties) in the RDF graph is very large and most properties occur infrequently. The second approach requires slightly more space (at least one integer label per node and edge label), but allows the use of different node orders for different edge labels. Thus graphs

that are, if we consider edges with any label, no CIG, may turn out to be CIGS if we consider each edge label separately, which provides a significant advantage.

In the outlook (Section 2.7), we briefly sketch an extension of the CIG-labeling scheme where we aim to find compatible orders for multiple relations (edge sets) on the same nodes. Such an extension allows us to combine the advantages of both approaches: Where necessary separate CIG-labelings are created, but where possible the integer labels are shared.

2.6.1 Evaluating 1-SPARQL

In the first dialect, 1-SPARQL, queries are single, ground triple patterns. Thus we only need to look-up the nodes for the given subject and object (whether they are URIs, literals, or blank nodes) and test adjacency wrt. the given property URI between the two nodes.

Theorem 7. *Evaluating a 1-SPARQL query on a* CIG*-labeled graph G with n nodes takes $O(\log n)$ time (regardless of the shape of G).*

Proof. Finding the nodes and their labels for the given subject and object is in $\log|\Sigma|$ (e.g., if data URIs are held in an appropriate dictionary) where Σ is the set of unique URIs and literals that occur in the data. Obviously, $|\Sigma| \leq n$. In both of the above representations the adjacency test given the node labels is constant, if the RDF graph can be represented as a CIG. Otherwise the test is in $\log n$. The overall complexity is in both cases $O(\log n)$

2.6.2 Evaluating A-SPARQL

For evaluating A-SPARQL a basic algorithm can be expressed as follows: Evaluate each triple pattern separately and then join the resulting substitutions. The challenge lies in representing the result of the join compactly. For acyclic queries we can achieve a compact representation of the join as follows:

- For each variable v, store candidate nodes in form of their integer label and interval sets for each dependent variable. Order these candidate nodes in the order of the relation between v and its parent in the query (if there are multiple parents clone v, but not its dependent variables). Initialize these candidate stores with all nodes and the interval sets $\{[1,|N|]\}$. This can be done in $O(n)$ as the nodes are stored in order in the CIG-labeling.
- For each triple pattern (v,p,v') involving two variables v and v', we look at each of the candidates for v and intersect its current interval set over v' with the one retrieved from the CIG-labeling for p. The intersection of ordered interval sets can be performed in time linear in the total number of intervals over v' in candidates for v.
- For each conditions (such as URI labels, type restrictions) on a variable, we mark all candidates for that variable that do not match that condition.

– Finally, we compute the actual answers by iterating (bottom-up) over each candidate store and computing for each index in the store its new index, the index of the next and the previous actual answer: For leave variables that is the first variable that is not marked (including itself). For inner variables that is the first variable that is not marked and for which the interval sets for each dependent variable are non empty. Analog for the previous actual answer. Whenever we have computed the actual answers of a variable v we adjust the indices of the parent variable(s) p as follows: The start index of each interval in the interval set of a candidate for p is advanced to the index of the next actual answer for p, the end index is reduced to the index of the previous actual answer. If the interval is now empty, it is dropped. This algorithm runs in time linear to the total number of candidates and intervals in candidates of any variable.

Theorem 8. *Evaluating an A-SPARQL query of size q on a* CIG*-labeled graph G with n nodes takes*

1. *$O(q \cdot n)$ time and space if G is a tree or* CIG*.*
2. *$O(q \cdot n^2)$ time and space otherwise.*

Proof. In both cases, step 1 is performed $O(q)$ times, thus takes in total $O(q \cdot n)$ time.

For step 2 and 3 we distinguish between CIGS and arbitrary graphs:

If G is a tree or CIG then there is a single interval per node and relation. Step 2 is performed once for triple pattern and at most n intervals are intersected (as each of the at most n candidates for v only contains one interval over v'). Thus it takes in total at most $O(q \cdot n)$ time. Step 3 is performed once and takes at most $O(q \cdot n)$ time as that is the upper bound for the total number of intervals in all candidate answers.

If G is an arbitrary graph there are up to $\frac{1}{2}n$ intervals per node and relation. Step 2 thus intersects at most n^2 intervals and takes $O(q \cdot n^2)$ time. Step 3 is performed once and takes at most $O(q \cdot n^2)$ time as that is the upper bound for the total number of intervals in all candidate answers.

For the actual implementation, we adapt a number of aspects of the algorithm, in particular the handling of marked nodes and the initialization that improve space usage in most cases, but do not affect the complexity and are therefore omitted here.

2.6.3 Towards Full SPARQL

For a given SPARQL query, we essentially compute an A-SPARQL core of the query, evaluate that core and then compute the rest of the query as in the relational algebra, accessing the results of the A-SPARQL core when needed.

It is worth pointing out that we can add many of SPARQL features also to the A-SPARQL core (without sacrificing the linear time evaluation). E.g., many forms of `UNION`, most `FILTER` expressions, and some cases of `OPTIONAL`. Though `UNION` and `OPTIONAL` somewhat complicate the evaluation algorithm sketched above, it is well-worth in practice. None of these changes, however, changes the overall complexity significantly and they are omitted here. For more details see [17].

Corollary 3. *Evaluating a SPARQL query of size q with q_g triples not covered by a chosen A-SPARQL core of the query on a* CIG*-labeled graph G with n nodes takes $O(n^{q_g} + q \cdot n)$ time and space on* CIGS *and $O(n^{q_g} + q \cdot n^2)$ time and space on other graphs.*

2.6.4 Towards Path Expressions

As discussed in Section 2.3.2, the CIG-labeling is particularly well suited for reachability graphs. Though SPARQL does not explicitly support reachability queries, it may be evaluated, e.g., against the RDFS-entailment graph of a given RDF graph which includes reachability semantics for, e.g., rdfs:subClassOf. Furthermore, CIG-labelings can be used to support efficient iteration and constant time reachability test for extensions of SPARQL with reachability queries. For instance, nSPARQL [38] proposes such and more powerful operators (full path expressions). The latter, unfortunately, can no longer be iterated in time liner to the matches (rather than to the entire graph) or tested in constant time, not even with CIG-labelings.

2.7 Conclusion

In this chapter, we introduce the CIG-labeling, a novel, easy and efficiently implementable, yet surprisingly powerful labeling schemes for trees and graphs. We show that CIG-labelings allow for constant time, constant per-node space adjacency (and reachability) testing not only on trees, but also on many graphs. We precisely characterize the class of graphs with this property.

CIG-labelings can be easily applied to RDF and give us linear time evaluation for large classes of SPARQL queries on many RDF graphs. They also have the potential to significantly speed-up the processing of general SPARQL queries, however current indexing algorithms may prove to be too expensive for very large RDF graphs.

The development of more efficient heuristics and approximations for very large RDF graphs is therefore clearly called for. On the other hand, it also worth investigating how to increase the class of graphs which can still be queried in constant time, even if we increase the indexing time (this is relevant, e.g., when indexing is done infrequently and en-bloc as in search engines). In Section 2.5.4 we discussed several such extensions, though optimal labeling algorithms turn out to be NP-complete even for small extensions.

Another candidate is the exploitation of results on zero-partitionable property of adjacency matrix. This is, in some sense, a generalization of the consecutive ones property that CIGS are based on: Intuitively in a zero-partitionable matrix we can push all 0s either to the right or to the bottom: A binary matrix is zero-partitionable if every 0 can be labeled either by R, in which case every position to its right must be an R labeled 0; or by C, in which case very position below must be a C labeled 0. On a graph with a zero-partitonable adjacency matrix we can still provide constant time and constant per-node space adjacency testing though the class is a proper superclass of CIGS. The best known labeling algorithm is polynomial but with a very large constant in the exponent. It is an open question whether the labeling is effective for realistic RDF graphs.

Acknowledgements

The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement n^{o} 211932 (cf. `http://www.kiwi-project.eu/`).

References

1. Agrawal, R., Borgida, A., Jagadish, H.V.: Efficient management of transitive relationships in large data and knowledge bases. In: Proc. ACM Symp. on Management of Data (SIGMOD), New York, NY, USA, ACM (1989) 253–262
2. Al-Khalifa, S., Jagadish, H.V., Koudas, N., Patel, J.M., Srivastava, D., Wu, Y.: Structural Joins: A Primitive for Efficient XML Query Pattern Matching. In: Proc. Int. Conf. on Data Engineering, Washington, DC, USA, IEEE Computer Society (2002) 141
3. Backett, D.: Turtle—Terse RDF Triple Language. Technical report, Institute for Learning and Research Technology, University of Bristol (2007)
4. Beckett, D., McBride, B.: RDF/XML Syntax Specification (Revised). Recommendation, W3C (2004)
5. Bolzer, O.: Towards Data-Integration on the Semantic Web: Querying RDF with Xcerpt. Diplomarbeit/diploma thesis, University of Munich (2005)
6. Boncz, P., Grust, T., van Keulen, M., Manegold, S., Rittinger, J., Teubner, J.: MonetDB/X-Query: a fast XQuery Processor powered by a Relational Engine. In: Proc. ACM Symp. on Management of Data (SIGMOD), New York, NY, USA, ACM Press (2006) 479–490
7. Booth, K.S., Lueker, G.S.: Linear Algorithms to Recognize Interval Graphs and Test for the Consecutive Ones Property. In: Proc. of ACM Symposium on Theory of Computing, New York, NY, USA, ACM Press (1975) 255–265
8. Bruno, N., Koudas, N., Srivastava, D.: Holistic Twig Joins: Optimal XML Pattern Matching. In: Proc. ACM SIGMOD Int. Conf. on Management of Data, New York, NY, USA, ACM Press (2002) 310–321
9. Bry, F., Furche, T., Linse, B., Pohl, A.: Xcerptrdf: A pattern-based answer to the versatile web challenge. In: Proc. Workshop on (Constraint) Logic Programming (WLP). (2008)
10. Chen, L., Gupta, A., Kurul, M.E.: Stack-based algorithms for pattern matching on dags. In: Proc. Int'l. Conf. on Very Large Data Bases (VLDB), VLDB Endowment (2005) 493–504
11. Chen, T., Lu, J., Ling, T.W.: On Boosting Holism in XML Twig Pattern Matching using Structural Indexing Techniques. In: Proc. ACM SIGMOD Int. Conf. on Management of Data, New York, NY, USA, ACM Press (2005) 455–466
12. Chen, Z., Gehrke, J., Korn, F., Koudas, N., Shanmugasundaram, J., Srivastava, D.: Index structures for matching xml twigs using relational query processors. Data & Knowledge Engineering (DKE) **60**(2) (2007) 283–302
13. Christophides, V., Plexousakis, D., Scholl, M., Tourtounis, S.: On labeling schemes for the semantic web. In: Proc. Int'l. World Wide Web Conf. (WWW), New York, NY, USA, ACM (2003) 544–555
14. Cohen, E., Halperin, E., Kaplan, H., Zwick, U.: Reachability and Distance Queries via 2-hop Labels. In: Proc. ACM Symposium on Discrete Algorithms, Philadelphia, PA, USA, Society for Industrial and Applied Mathematics (2002) 937–946
15. Dietz, P.F.: Maintaining order in a linked list. In: Proc. ACM Symp. on Theory of Computing (STOC), New York, NY, USA, ACM (1982) 122–127
16. Fulkerson, D.R., Gross, O.A.: Incidence Matrices and Interval Graphs. Pacific Journal of Mathematics **15**(3) (1965) 835–855

17. Furche, T.: Implementation of Web Query Language Reconsidered: Beyond Tree and Single-Language Algebras at (Almost) No Cost. Dissertation/doctoral thesis, Ludwig-Maxmilians University Munich (2008)
18. Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman (1979)
19. Goldberg, P.W., Golumbic, M.C., Kaplan, H., Shamir, R.: Four strikes against physical mapping of DNA. Journal of Computational Biology **2**(1) (1995) 139–152
20. Gottlob, G., Koch, C., Pichler, R.: Efficient Algorithms for Processing XPath Queries. ACM Transactions on Database Systems (2005)
21. Gottlob, G., Leone, N., Scarcello, F.: The Complexity of Acyclic Conjunctive Queries. Journal of the ACM **48**(3) (2001) 431–498
22. Grust, T.: Accelerating XPath Location Steps. In: Proc. ACM Symp. on Management of Data (SIGMOD). (2002)
23. Grust, T., van Keulen, M., Teubner, J.: Staircase Join: Teach A Relational DBMS to Watch its (Axis) Steps. In: Proc. Int. Conf. on Very Large Databases. (2003)
24. Habib, M., McConnell, R., Paul, C., Viennot, L.: Lex-BFS and Partition Refinement, with Applications to Transitive Orientation, Interval Graph Recognition and Consecutive Ones Testing. Theoretical Computer Science **234**(1-2) (2000) 59–84
25. Haddadi, S., Layouni, Z.: Consecutive block minimization is 1.5-approximable. Information Processing Letters **108**(3) (2008) 132–135
26. Hsu, W.L.: PC-Trees vs. PQ-Trees. In: Proc. Int'l. Conf. on Computing and Combinatorics. Volume 2108 of LNCS. (2001)
27. Hsu, W.L.: A Simple Test for the Consecutive Ones Property. Journal of Algorithms **43**(1) (2002) 1–16
28. Jiang, H., Wang, W., Lu, H., Yu, J.X.: Holistic twig joins on indexed xml documents. In: Proc. Int'l. Conf. on Very Large Data Bases (VLDB), VLDB Endowment (2003) 273–284
29. Kou, L.T.: Polynomial complete consecutive information retrieval problems. SIAM Journal of Computing **6**(1) (1977) 67–75
30. Meuss, H., Schulz, K.U.: Complete Answer Aggregates for Treelike Databases: A Novel Approach to Combine Querying and Navigation. ACM Transactions on Information Systems **19**(2) (2001) 161–215
31. Olteanu, D.: SPEX: Streamed and Progressive Evaluation of XPath. IEEE Transactions on Knowledge and Data Engineering (2007)
32. Olteanu, D., Furche, T., Bry, F.: Evaluating Complex Queries against XML streams with Polynomial Combined Complexity. In: Proc. British National Conf. on Databases (BNCOD). (2003) 31–44 17 citations [Google Scholar].
33. Olteanu, D., Furche, T., Bry, F.: An Efficient Single-Pass Query Evaluator for XML Data Streams. In: Data Streams Track,Proc. ACM Symp. on Applied Computing (SAC). (2004) 627–631 17 citations [Google Scholar].
34. Olteanu, D., Meuss, H., Furche, T., Bry, F.: XPath: Looking Forward. In: Proc. EDBT Workshop on XML-Based Data Management. Volume 2490 of Lecture Notes in Computer Science., Springer (2002) 160 citations [Google Scholar].
35. O'Neil, P., O'Neil, E., Pal, S., Cseri, I., Schaller, G., Westbury, N.: ORDPATHs: Insert-friendly XML Node Labels. In: Proc. ACM Symp. on Management of Data (SIGMOD), ACM Press (2004) 903–908
36. Paige, R., Tarjan, R.E.: Three partition refinement algorithms. SIAM Journal of Computing **16**(6) (1987) 973–989
37. Perez, J., Arenas, M., Gutierrez, C.: Semantics and complexity of sparql. In: Proc. Int'l. Semantic Web Conf. (ISWC). (2006)
38. Pérez, J., Arenas, M., Gutierrez, C.: nsparql: A navigational language for rdf. In: Proc. Int'l. Semantic Web Conf. (ISWC). (2008) 66–81

39. Polleres, A.: From sparql to rules (and back). In: Proc. Int'l. World Wide Web Conf. (WWW), New York, NY, USA, ACM (2007) 787–796
40. Prud'hommeaux, E., Seaborne, A.: SPARQL Query Language for RDF. Proposed recommendation, W3C (2007)
41. Schenkel, R., Theobald, A., Weikum, G.: HOPI: An Efficient Connection Index for Complex XML Document Collections. In: Proc. Extending Database Technology. (2004)
42. Su-Cheng, H., Chien-Sing, L.: Node labeling schemes in xml query optimization: A survey and trends. IETE Technical Review **26**(2) (2009) 88–100
43. Trißl, S., Leser, U.: Fast and practical indexing and querying of very large graphs. In: Proc. ACM Symp. on Management of Data (SIGMOD), New York, NY, USA, ACM (2007) 845–856
44. Wang, H., He2, H., Yang, J., Yu, P.S., Yu, J.X.: Dual labeling: Answering graph reachability queries in constant time. In: Proc. Int'l. Conf. on Data Engineering (ICDE), Washington, DC, USA, IEEE Computer Society (2006) 75
45. Weigel, F., Schulz, K.U., Meuss, H.: The bird numbering scheme for xml and tree databases – deciding and reconstructing tree relations using efficient arithmetic operations. In: Proc. Int'l. XML Database Symposium (XSym). Volume 3671 of LNCS., Springer-Verlag (2005) 49–67
46. Weinzierl, A.: Interval-based graph representations for efficient web querying. Diplomarbeit/diploma thesis, Ludwig-Maxmilians University Munich (2009)

Contents