1

Four Lessons in Versatility or How Query Languages Adapt to the Web

François Bry, Tim Furche, Benedikt Linse, Alexander Pohl,
Antonius Weinzierl, and Olga Yestekhina

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

Abstract. Exposing not only human-centered information, but machine-processable data on the Web is one of the commonalities of recent Web trends. It has enabled a new kind of applications and businesses where the data is used in ways not foreseen by the data providers. Yet this exposition has fractured the Web into islands of data, each in different Web formats: Some providers choose XML, others RDF, again others JSON or OWL, for their data, even in similar domains. This fracturing stifles innovation as application builders have to cope not only with one Web stack (e.g., XML technology) but with several ones, each of considerable complexity.

With Xcerpt we have developed a rule- and pattern based query language that aims to give shield application builders from much of this complexity: In a single query language XML and RDF data can be accessed, processed, combined, and re-published. Though the need for combined access to XML and RDF data has been recognized in previous work (including the W3C's GRDDL), our approach differs in four main aspects: (1) We provide a single language (rather than two separate or embedded languages), thus minimizing the conceptual overhead of dealing with disparate data formats. (2) Both the declarative (logic-based) and the operational semantics are unified in that they apply for querying XML and RDF in the same way. (3) We show that the resulting query language can be implemented reusing traditional database technology, if desirable. Nevertheless, we also give a unified evaluation approach based on interval labelings of graphs that is at least as fast as existing approaches for tree-shaped XML data, yet provides linear time and space querying also for many RDF graphs.

We believe that Web query languages are the right tool for declarative data access in Web applications and that Xcerpt is a significant step towards a more convenient, yet highly efficient data access in a "Web of Data".

1.1 Introduction

The one undeniable trend in the development of the Web has been a move from human-centered information to more machine-processable data. This trend is a part of most visions for the future of the Web, may they be called "Web 2.0", "Semantic Web",

"Web of Data", "Linked Data". There is a reason that this trend underlies so many of the visions for a future Web: With machine-processable data, other agents than the owner or publisher of data can create novel applications, e.g., by using the data in a context never envisioned by the data owner, by presenting it in different ways or media, or by enhancing or mixing it with other data.

Unfortunately, though machine-processable data is called for by many of these visions, they do not agree on the *data format*. For human-centered information, HTML has clearly dominated the Web. For machine-processable data, Web 2.0 APIs and publishers tend to use XML, JSON, or YAML, Semantic Web publishers RDF and/or OWL. This way, application designers are either impeded from using data published in, say, RDF, if they are used to data in, say, XML or they have to cope with not only one (already fairly complex) stack of Web technologies but several.

The need for a more integrated, easier access to Web data has been recognized: For instance, the W3C has proposed a means of accessing XML data as RDF (GRDDL [54]). Other approaches integrate existing RDF query languages into XML query languages (XSPARQL [7], [78]) or vice versa ([60], SPAT[1]. In this work, we present a different answer to this problem: a single, unified language, called Xcerpt, that can query both XML and RDF with the same ease. Previous approaches require the user to learn (a) an XML (usually XPath or XQuery), (b) an RDF query language (usually SPARQL), and (c) how concepts from RDF and XML are mapped to each other, if at all. In our approach, we first develop *a query language flexible enough to deal with most Web data* (in the spirit of, though with quite different focus and result than [137]). Then we only have to teach the user how to query RDF resp. XML with that query language, reusing as much of the data and query concepts between the two settings as possible. Not only does this reduce the learning curve for the user considerably, it also makes it easy to extend the approach with further Web formats such as JSON, YAML, or Topic Maps.

We introduce Xcerpt in Sections 1.4 and 1.5 after a brief recall of the basics of the two Web formats considered here, XML and RDF, in Section 1.2.

But defining a language for unified access to XML and RDF is just how the story begins. For the approach to be feasible, we require two more ingredients: 1. a simple semantics that is nevertheless versatile enough to cover the specifics of both XML and RDF. 2. an evaluation engine that is competitive to engines specialized to XML or RDF data only.

In Sections 1.8 to 1.11 we propose two different ways to define the (declarative) semantics of Xcerpt: The first uses a modified form of simulation to describe which queries match what data. It is flexible enough to deal with queries on XML and RDF data and can be defined very concisely. We show in Section 1.9 and 1.10 how to adapt the (well-founded) semantics of rule programs with negation to use simulation rather than term equality/instantiation.

This gives an easy, straightforward definition of the semantics of Xcerpt. However, the disadvantage is that required notion of simulation is not as well studied as term equality and not supported by existing database or rule technologies. Therefore, we show in Section 1.11 how Xcerpt can be translated into standard Datalog with nega-

[1] `http://www.w3.org/2007/01/SPAT/`.

tion and value invention ($\text{Datalog}^{\neg}_{new}$) which can be evaluated by most SQL-database engines and many rule engines. Not only do we show how to translate Xcerpt into $\text{Datalog}^{\neg}_{new}$, but we do the same for XPath, XQuery, and SPARQL, thus establishing a uniform formal foundation for all these languages (that we exploit in Section 1.13 for a unified evaluation engine for all these languages). Moreover, we use the translation to prove several complexity and expressiveness features. Most importantly, we show that full Xcerpt is unsurprisingly Turing-complete, that stratification does not limit the expressiveness of Xcerpt, and identify a decidable fragment (weakly-recursive Xcerpt).

For $\text{Datalog}^{\neg}_{new}$ and thus for Xcerpt, whether on XML or RDF data, we define a novel evaluation algorithm and indexing scheme in Section 1.13, thus turning to the second of the two missing ingredients, the competitive evaluation engine. We show how to extend tree labeling schemes (such as the pre/post-encoding [80]) to graph data in a novel way: Where previous such approaches [6, 150, 48, 144] can not guarantee linear time and space evaluation of acyclic conjunctive queries on interesting super-classes of trees, our approach exhibits such a class: the continuous-image graphs. On this significant super-class of trees we can still maintain linear time and space evaluation. The basic idea of the approach is a generalized interval labeling together with (most importantly) a novel join algorithm for intermediary answers represented by intervals.

Together with the results from Section 1.11, we thus obtain a surprisingly large linear time and space fragment of Xcerpt, viz. (weakly-recursive) acyclic Xcerpt on continuous-image graphs, a novel super-class of trees. The same also applies to, e.g., SPARQL.

To complete the evaluation of Xcerpt, we not only need an efficient evaluation engine for Xcerpt queries, but for Xcerpt *rules*. Section 1.14 gives a first step towards such a rule engine for Xcerpt. It introduces *simulation unification* as an extended, more flexible form of unification that is adapted to Xcerpt's notion of simulation discussed above. Based on simulation unification, we show how subsumption can be exploited to define an efficient resolution with tabling for locally stratified Xcerpt programs.

To summarize, the theme of this chapter is the investigation of how to address the increasing number of diverse data formats being introduced on the Web. We suggest as a solution, Xcerpt,

1. *a versatile query language* that allows access to both XML and RDF in the same language, sharing concepts as much as possible (Section 1.3– 1.6). It is complemented by
2. *a versatile declarative semantics* based on a form of simulation adapted to Web data that is easy to understand, yet can be translated to standard database and rule technology, as can XPath, XQuery, and SPARQL (Section 1.7– 1.11). For that semantics (and thus for Xcerpt, XPath, XQuery, and SPARQL), we propose
3. *a versatile evaluation algorithm* that is able to provide the best-known complexity for acyclic conjunctive queries on tree-shaped XML data, manages to maintain that complexity for many RDF graphs, and yet can also operate on arbitrary graphs (Section 1.13). We extend that evaluation algorithm towards a full versatile rule-based query language for the Web like Xcerpt by illustrating how resolution with tabling can be adapted to use

4. *a versatile form of subsumption* based on simulation unification for determining where previously computed answers to a sub-query can be reused for further sub-queries (Section 1.14).

The structure of this chapter follows the four perspectives on addressing the rising amount of Web data formats: data, query language, semantics, and evaluation. While the parts on data and query are to some extent necessary for understanding the parts on semantics and evaluation, the latter two are fairly independent. It is not necessary to understand the details of the semantics for the evaluation or vice versa.

Related Work. To keep the parts fairly self-contained and to avoid overly long preliminaries, we decided to address related work in each part separately.

In particular, Section 1.6.3 compares Xcerpt, in particular its features for accessing RDF, with SPARQL, the W3C proposal for querying RDF and a number of its extensions. Section 1.11.6 gives a brief comparison of the challenges when translating Xcerpt to Datalog$^{\neg}_{new}$, i.e., to existing database and rule technology, compared to XPath, XQuery or SPARQL. For the evaluation, we extensively compare our approach with existing labeling schemes for tree and graph data in Section 1.13.5. The basic principles of the evaluation algorithm are discussed in the context of related work in Section .

As pointed out there are a number of previous approaches to integrating XML and RDF access. These can be divided in two categories: Approaches such as GRDDL [57] use two separate query languages to first transform data from one format in the other and then to query only in the latter format. The advantage of this approach is that existing language engines can be used as is. The second kind of approaches is exemplified by XSPARQL [127] and [78]: Here one of the languages is embedded into the other, providing an interface between the two languages (of varying sophistication). The advantage is that we can now transfer results in both directions, the disadvantage is that new query engines or rather involved query translations are needed[2].

For both approaches there are two main shortcomings:

1. The user has to learn two different query languages that were designed entirely separate.
2. Since the language engines remain entirely separate, these approaches first transform *all data* (without respect to what is actually queried), then load all transformed data in the second query engine, only then it is filtered by the conditions of the queries in the target format. Thus, there is no chance for goal-driven query evaluation and even static propagation of query conditions from queries in the target format to the transformation queries are very hard due to the starkly varying semantics of the two languages involved.

Though Xcerpt and Xcerpt$^{\text{RDF}}$ still require the use to learn some concepts specific to XML or RDF, they are design to share concepts where possible. Furthermore, we use a unified semantics thus allowing for full static cross-format optimization and a unified evaluation allowing for dynamic cross-format optimization.

[2] In Section 1.11.6 we illustrate a first step towards a translation approach.

Neither the above approaches not Xcerpt addresses integrating also queries on (OWL) ontologies, e.g., in the style of [**?**]. Though this is certainly an important issue, it is out of the scope of this chapter. We believe that some of the discussed issues apply also in that context (in particular, the treatment of blank nodes in RDF in the semantics and evaluation of Xcerpt), but there are many more issues when considering even conjunctive queries on ontologies that would need addressing.

1.2 Versatile Data

1.2.1 Extensible Markup Language (XML)

XML [28] is, by now, *the* foremost data representation format for the Web and for semi-structured data in general. It has been adopted in a stupendous number of application domains, ranging from document markup (XHTML, Docbook [149]) over video annotation (MPEG 7 [109]) and music libraries (iTunes[3]) to preference files (Apple's property lists [9]), build scripts (Apache Ant[4]), and XSLT [94] stylesheets. XML is also frequently adopted for serialization of (semantically) richer data representation formats such as RDF or TopicMaps.

XML is a generic markup language for describing the structure of data. Unlike in HTML (HyperText Markup Language), the predominant markup language on the web, neither the tag set nor the semantics of XML are fixed. XML can thus be used to derive markup languages by specifying tags and structural relationships.

The following presentation of the information in XML documents is oriented along the XML Infoset [56] which describes the information content of an XML document. The XQuery data model [66] is, for the most parts, closely aligned with this view of XML documents.

Following the XPath and XQuery data model, we provide a *tree shaped* view of XML data. This deviates from the Infoset where valid ID/IDREF links are resolved and thus the data model is graph, rather than tree shaped. This view is adopted in some XML query languages such as Xcerpt [40] and Lorel [3], but most query languages follow XPath and XQuery and consider XML tree shaped.

XML in 500 Words The core provision of XML is a syntax for representing hierarchical data. Data items are called elements in XML and enclosed in start and end *tags*, both carrying the same tag names or *labels*. `<author>...</author>` is an example of such an element. In the place of '...', we can write other elements or character data as *children* of that element. The following listing shows a small XML fragment that illustrates elements and element nesting:

```
<bib xmlns:dc="http://purl.org/dc/elements/1.1/">
 <article journal="Computer Journal" id="12">
   <dc:title>...Semantic Web...</dc:title>
   <year>2005</year>
```

[3] http://www.apple.com/itunes/
[4] http://ant.apache.org/

```
       <authors>
6        <author>
           <first>John</first> <last>Doe</last> </author>
8        <author>
           <first>Mary</first> <last>Smith</last> </author>
10     </authors>
     </article>
12   <article journal="Web Journal">
       <dc:title>...Web...</dc:title>
14     <year>2003</year>
       <authors>
16       <author>
           <first>Peter</first> <last>Jones</last> </author>
18       <author>
           <first>Sue</first> <last>Robinson</last> </author>
20     </authors>
     </article>
22 </bib>
```

In addition, we can observe *attributes* (name, value pairs associated with start tags) that are essentially like elements but may only contain character data, no other nested attributes or elements. Also, by definition, *element order* is significant, attribute order is not. For instance

```
  <author><last>Doe</last><first>John</first></author>
```

represents different information than the `author` element in lines 6–9, but

```
  <article id="12" journal="Computer Journal">...</article>
```

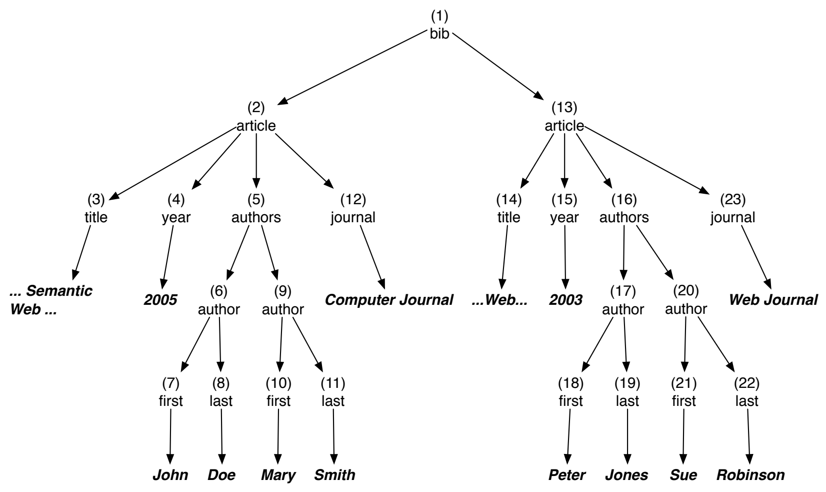
represents the same element information item as lines 2–15.

Figure 1.1 gives a graphical representation of the XML document that is referenced in preceding illustrations. When represented as a graph, an XML document without links is a labeled tree where each node in the tree corresponds to an element and its type. Edges connect nodes and their children, that is, elements and the elements nested in them, elements and their content and elements and their attributes. Since the visual distinction between the parent-child relationship can be made without edge labels and since attributes are not addressed or receive no special treatment in the research presented in this text, edges will not be labeled in the following figures.

Elements, attributes, and character data are XML's most common information types. In addition, XML documents may also contain *comments*, *processing instructions* (name-value pair with specific semantics that can be placed anywhere an element can be placed), *document level information* (such as the XML or the document type declarations), *entities*, and *notations*, which are essentially just other kinds of information containers.

On top of these information types, two additional facilities relevant to the information content of XML documents are introduced by subsequent specifications: Namespaces [26] and Base URIs [108]. Namespaces allow partitioning of element labels used in a document into different namespaces, identified by a URI. Thus, an element is no longer labeled with a single label but with a triple consisting of the *local name*, the

Fig. 1.1 Visual representation of sample XML document



namespace prefix, and the *namespace URI*. E.g., for the `dc:title` element in line 3, the local name is `title`, the namespace prefix is `dc`, and the namespace URI (called "name" in [56]) is `http://purl.org/dc/elements/1.1/`. The latter can be derived by looking for a *namespace declaration* for the prefix dc. Such a declaration is shown in line 1: `xmlns:dc="http://...` It associates the prefix `dc` with the given URI in the scope of the current element, i.e., for that element and all elements contained within unless there is another nested declaration for `dc`, in which case that declaration takes precedence. Thus, we can associate with each element a set of *in-scope namespaces*, i.e., of pairs namespace prefix and URI, that are valid in the scope of that element. Base URIs [108] are used to resolve relative URIs in an XML document. They are associated with elements using `xml:base="http://...` and, as namespaces, are inherited to contained elements unless a nested `xml:base` declaration takes precedence.

The above features of XML are covered by most query languages. Additionally some languages (most notably XQuery) also provide access to type information associated via DTD or XML Schema [65]. These features are mentioned below where appropriate but not discussed in detail here.

1.2.2 Resource Description Framework (RDF)

As the second preeminent data format on the Semantic Web, the *Resource Description Format (RDF)* [107, 100, 84] is emerging. RDF is, though much less common than XML, a widespread choice for interchanging (meta-) data together with descriptions of the schema and, in contrast to XML, a basic description of its semantics of that data.

Not to distract from the salient points of the discussion, we omit typed literals (and named graphs) from the following discussion.

RDF in 500 Words 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 [20]), such as RDF/XML [15], an XML dialect for representing RDF, or Turtle [13] which is also used in SPARQL. The following Turtle data represents roughly the same data as the XML document discussed in the previous section:

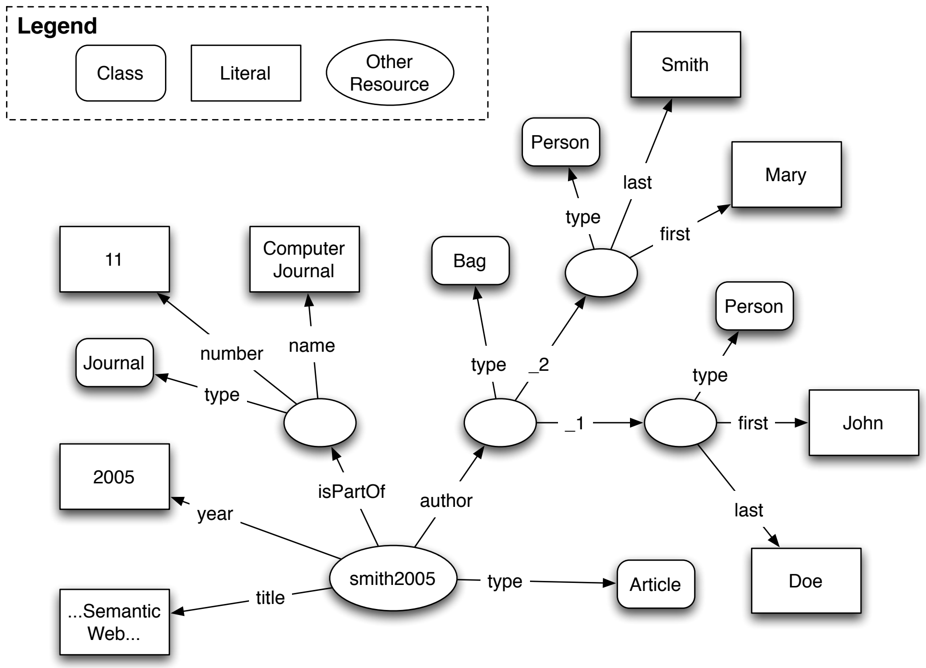
```
@prefix dc: <http://purl.org/dc/elements/1.1/> .
@prefix dct: <http://purl.org/dc/terms/> .
@prefix vcard: <http://www.w3.org/2001/vcard-rdf/3.0#> .
@prefix bib: <http://www.edutella.org/bibtex#> .
@prefix ex: <http://example.org/libraries/#> .
ex:smith2005 a bib:Article ; dc:title "...Semantic Web..." ;
     dc:year "2005" ;
     ex:isPartOf [ a bib:Journal ;
        bib:number "11"; bib:name "Computer Journal" ] ;
     bib:author [ a rdf:Bag ;
        rdf:_1 [ a bib:Person ;
           bib:last "Smith" ; bib:first "Mary" ] ;
        rdf:_2 [ a bib:Person ;
           bib:first "John" ; bib:last "Doe" ] ] .
```

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 1 reads as ex:smith2005 is a (has rdf:type) bib:Article and has dc:title “…Semantic Web…”. Lines 3–4 show a blank node: the article is part of some entity which we can not (or don’t care to) identify by a unique URI but for which we give some properties: it is a bib:Journal, has bib:number “11”, and bib:name “Computer Journal”.

Figure 1.2 shows a visual representation of the above RDF data, where we distinguish literals (in square boxes) and classes, i.e., resources that can be used for classifying other resources, and thus can be the object of an `rdf:type` statement (in square boxes with rounded edges) from all other resources (in plain ellipses).

What sets RDF apart from XML and justifies its role as *the* data format for the *Semantic* Web is that RDF data comes with attached meaning, that allows us to infer additional knowledge beyond what is stated explicitly. Query languages are usually expected to behave consistent w.r.t. some form of RDF entailment (e.g., simple, full, or RDFS entailment), i.e., graphs equivalent under the respective entailment yield the same answers. Simply stated, rather than just consulting the actual RDF data for answering a query, we might also need to consider additional, inferred triples depending on the form of entailment chosen. E.g., when querying for resources of type bib:Publication

Fig. 1.2 Visual representation of sample RDF graph



we might also want to return bib:Articles if we have the additional information that bib:Article is a sub-class of bib:Publication. SPARQL, e.g., is designed to be agnostic of the particular entailment used: it can be used to query RDF data under any of the above mentioned entailment forms.

In the following, we assume familiarity with the notion of RDF entailment, interpretation, model, as well as the RDFS semantics from [84].

1.3 Versatile Queries

With the rise of a plethora of different semi-structured Web formats, versatility [32] has become the central requirement for web query languages. Besides the well-known and ubiquitous formats HTML, XML and RDF, there are quite a lot of less familiar formats such as RDFa [5, 4] for embedding RDF information in HTML pages, the microformats [97] geo, hCard, hCalendar, hResume, etc., the ISO-standard Topic Maps [73, 122]. We call a web query language *format versatile*, if it can handle, merge or transform data in different formats within the same query program. The need for integrating data from different formats has been acknowledged by partial solutions such as GRDDL [57, 148, 71], hGRDDL [4] and XSPARQL [7]. All these solutions have in common that they try to solve the problem of web data integration by applying a mix of already established technologies such as XSLT transformations, DOM manipulations, and a combination of XML and RDF query languages such as XQuery and SPARQL. It is thus unsurprising that understanding these solutions requires a large background knowledge of the employed technologies, and that the methods are much more complicated than they could be if a format-versatile language particularly geared at integrating data from different web formats was employed.

Besides format versatility, we distinguish two other kinds of versatility: *schema* and *representational versatility*. A web query language is called *schema versatile*, if it can handle and intermediate between different schemata (i.e. schema heterogeneity) on the Web. Usage of different schemata for representing similar data is very common and well-studied in the field of data integration [145, 105]. Since the Web is being enhanced with structured and semantically rich data, data integration on the Web [101] has also received considerable attention and has spurred the growth of ontology alignment [116, 62, 63] research. Schema heterogeneity on the Web is encountered whenever two ontologies describe the same kind of information on the Web, but employ different languages for this end.

Finally, *representational heterogeneity* is encountered in XML dialects such as RDF/XML, where the same information is represented differently due to the use of syntactic sugar notations – e.g. for `rdf:type` arcs or for the concise notation of literals, URIs or RDF containers. Moreover, representational heterogeneity is present in any XML dialect that does not enforce any order of the information that it provides, since for serialization an arbitrary order must be chosen. We call a language *representational versatile*, if it can query data agnostic of the representational variant chosen.

In this and the following sections, we show how the design of Xcerpt query terms, construct terms and rules has lead to a versatile language with respect to all three issues – format, schema and representation. This section starts out by looking at Xcerpt from an abstract point of view, its relationship to logic programming and the interface defined by Xcerpt terms. In Section 1.4, we introduce XML querying, construction and transformation at the example of harvesting search results and microformat information of personal profile pages of a social network. In Section 1.5, Xcerpt's RDF querying capabilities are presented with special emphasis on treating RDF specifies such as containers, collections and reifications. Finally, in Section 1.6, we present a use-case on combining microformat information harvested with Xcerpt$^{\text{XML}}$ and RDF data queried with Xcerpt$^{\text{RDF}}$, thus combining versatile querying in XML and RDF.

Xcerpt Terms from an Abstract Point of View: Simulation, Substitutions, and Application of Substitution Sets. Xcerpt is a rule and pattern based language inspired by logic programming, but with significantly richer querying capabilities that are necessitated by the semi-structured nature of data on the Web.

In contrast to Prolog unification, Xcerpt uses a more involved kind of unification called *simulation unification*[5] to extract bindings of logical variables from Web data.

While Prolog rules consist of possibly non-ground terms in the head and the body of a rule, Xcerpt distinguishes between *construct terms* and *query terms* to be used in the heads and the bodies of rules, respectively. This differentiation is necessary because

[5] The term *simulation* is derived from graph simulation as defined in [1].

the semi-structured nature of data on the Web requires expressive query constructs – such as descendant, subterm negation, optionality – only in the query part of a rule (i.e. in the query terms), and constructs for reassembling the data – such as grouping – only in the construction part (i.e. the construct terms). Additionally, Xcerpt offers data terms as an abstraction of XML (and thus also HTML) and RDF data. Xcerpt terms fulfill the following three properties: (i) any data term is also a query term, (ii) any data term is also a construct term, and (iii) the intersection between the set of construct terms and query terms is exactly the set of data terms, where some subterms may be substituted by variables.

Also Prolog differentiates between terms and ground terms and facts. In Prolog it holds that any ground term is a fact (i.e. data). In Xcerpt, however, a term may very well be ground, but still be only an incomplete description of data – i.e. a query. Xcerpt terms are formally – but, for the sake of brevity, not in their entirety – defined in Section 1.8.

The differences between Prolog Unification and Simulation unification can be briefly summarized as follows:

- *Non-Symmetry of simulation unification.* Whereas Prolog unification is a symmetric operation on two generally non-ground terms, Xcerpt simulation unfication is a non-symmetric relation having a query term as the first argument, and a construct term as the second.
- *Different types of variables*. While Prolog Unification only allows for one single type of variable that will bind to any type of term, Xcerpt differentiates between different types of variables. Obviously the types of variables also differ with the data format that is being queried (XML, RDF, Topic Maps, Microformats, etc). When querying XML data, Xcerpt distinguishes between term variables, that bind to an entire XML fragment and label variables, that bind to a qualified or local name only.[6]
- *Notations for querying incomplete data*. Due to the almost schemaless nature of data on the Web, Xcerpt terms must be able to incompletely specify or describe the data that is being searched for. These notations include optionality of subterms, subterms at arbitrary depth and negated subterms and are introduced in detail in Section 1.4.
- *Substitution sets instead of substitutions*. While in Prolog one can find a single most general unifier for two terms t_1 and t_2 up to variable renaming, this is not true for Xcerpt. Simulation unification between two Xcerpt terms xt_1 and xt_2 results in a set of substitutions (that may very well contain only a single substitution or none at all), which is due to the richer kind of simulation and the deeper structure of data found on the Web. Imagine, for example, a biological database in XML format on the Web that contains data about enzymes and chemical reactions they catalyze. Although the database may be contained in a single XML document, the query for all pairs of enzymes and catalyzed reactions should, obviously, return more than a single tuple.

[6] Variables for term identifiers and for XML attributes are not considered in this survey for the sake of brevity.

Feature unification [93, 92], i.e. unification between feature terms, has been investigated in linguistics to aid automatic translation of natural language texts. Feature terms are used as an abstract representation of text, and are similar to semi-structured expressions as far as they can be arbitrarily nested as XML documents, may contain nodes that are entirely represented by their properties (just as RDF blank nodes), and in that the order of subterms may or may not be relevant. In contrast to simulation unification, feature unification is symmetric, and feature terms do not provide constructs for specifying incompleteness in depth or different types of variables. Finally, feature unification does not return sets of variable bindings but serves to translate text from one natural language to another.

Matching or – in Xcerpt terminology – simulating queries with data is only one of two steps in the transformation of semi-structured data. Just as Prolog, but more consequently (because of aggregation), Xcerpt clearly separates extraction of data (the data is bound to variables within *rule bodies*) and construction of new data (reassembling the data by application of substitution sets to *rule heads*).[7] This separation contrasts with XML query languages such as XQuery and XSLT, in which querying and construction is *intertwined*. Construction of new data with rule based languages is achieved by applying a substitution to a term. As mentioned above, however, Xcerpt does not deal with ordinary substitutions, but with *substitution sets*, and moreover, it differentiates between different kinds of terms. Therefore, we must be more specific: Construction of new data in Xcerpt is achieved by applying *sets* of substitutions to *construct* terms. The step from single substitutions to substitution sets allows the introduction of grouping constructs and aggregations to rule-based web querying. In the absence of grouping and aggregation constructs, application of substitution sets does not result in a single Xcerpt term, but in a set of terms (which may very well be unary or even empty).

The above discussion of Xcerpt terms can be summarized by the following interface (written as a functional type signature) of an Xcerpt term:

$$simulates :: QueryTerm \rightarrow ConstructTerm \rightarrow Bool \quad (1.1)$$

$$simulation\_unify :: QueryTerm \rightarrow ConstructTerm \rightarrow SubstitutionSet \quad (1.2)$$

$$apply\_substitution\_set :: SubstitutionSet \rightarrow ConstructTerm \rightarrow [DataTerm] \quad (1.3)$$

The function *simulates* returns true for a query term q and a construct term t if and only if the substitution set returned for $simulation\_unify(q,t)$ is non-empty. In addition to three above mentioned functions, a function which decides the subsumption relationship between two Xcerpt query terms is required if an optimized tabling algorithm for backward chaining evaluation of a multi-rule program is to be used. For more information about the subsumption relationship between Xcerpt query terms see Section 1.14.

In Section 1.4, we informally introduce the XML processing capabilities of Xcerpt, Xcerpt$^{\text{XML}}$ terms, Xcerpt$^{\text{XML}}$ simulation unification and the application of substitution sets to Xcerpt$^{\text{XML}}$ terms. In Section 1.5 we do the same for Xcerpt$^{\text{RDF}}$.

[7] Queries against a single Prolog rule, such as the `append` rule, may indeed be used to achieve both: concatenation of lists and finding components of a list. Still, querying is performed by matching *rule bodies* with terms, and data construction by filling in bindings for variables in *rule heads*.

1.4 Versatile Queries I: XML—Examples and Patterns

A large number of query languages for XML data have been proposed in the past. They range from navigational languages such as XSLT [95] XQuery [141], their common subset XPath [19], and Quilt [47] (the predecessor of XQuery) over pattern based languages such as XML-QL [58], UnQL [41] and Xcerpt to visual query languages such as visXcerpt [17], XQBE [12] and XML-GL [46]. For a comprehensive survey over XML query languages, their expressive power and language constructs, see [14], for a comparison of Lorel, XML-QL, XML-GL, XSL and XQL see [23].

In this section, we introduce the XML processing capabilities of Xcerpt, taking Web search results, personal profile pages from the LinkedIn social network and FOAF documents as a running example. With this data, the following task will be accomplished:

- We will extract links to LinkedIn profile pages from search results of the Google search engine. These search results are wrapped within deeply nested HTML which primarily serve presentation purposes, and snippets of text extracted from the indexed pages. By matching among others `class` and `id` attributes, only the relevant links will be extracted.
- From the profile pages relevant data of the curriculum vitae of the persons is identified and extracted by exploiting the microformat vocabularies `hresume`, `hcalendar` and `hcard` which are integrated into the HTML pages for semantic enrichment of the textual content.
- Finally, FOAF documents are queried to find additional information not present in the LinkedIn profile. Since FOAF is an RDF format that may be serialized in RDF/XML, we will discuss the syntactic XML structure of these documents and their correspondence to Xcerpt$^{\text{RDF}}$ query terms in this section, but use Xcerpt$^{\text{RDF}}$ to query their contents in Section 1.5.

1.4.1 Xcerpt$^{\text{XML}}$ Data and Rules.

This section introduces Xcerpt$^{\text{XML}}$ data terms, that abstract from XML documents, ignoring XML specifities such as processing instructions, comments, entities and DTDs. Xcerpt$^{\text{XML}}$ terms are introduced to allow a more concise representation of XML data that can be extended to form queries and construct patterns to be used in rules.

Rules are written in a similar fashion to Datalog or Prolog rules, and have the following general form:

```
CONSTRUCT <CONSTRUCTTERM> FROM <QUERY> END
```

Xcerpt queries are enclosed between the `FROM` and `END` keywords and are *matched* – in Xcerpt terminology *simulated* – with data. Due to Xcerpt's answer closedness (see Definition 1 for details), data may also be used as queries. To see how XML is represented as Xcerpt data, consider the FOAF document in Listing 1.1 and the corresponding Xcerpt data term in Listing 1.2.

FOAF is an acronym for "Friend-Of-A-Friend", which is a vocabulary for specifying relationships among people, their personal information such as adresses, education and contact information. FOAF is primarily an RDF vocabulary, and is therefore semantically richer than plain XML data, but most FOAF documents are serialized in RDF/XML. Therefore, FOAF documents serialized in RDF/XML can be queried or transformed *syntactically* (on the XML level) or *semantically* (on the RDF level). While this section deals with syntactic transformations of Web data, semantic queries, transformations and reasoning using Xcerpt$^{\text{RDF}}$ are discussed in Section 1.5.

```
<rdf:RDF xmlns:rdf="http://www.w3 ... rdf-syntax-ns#"
       xmlns:rdfs="http://www.w3 ... rdf-schema#"
       xmlns:foaf="http://xmlns.com/foaf/0.1/"
       xml:base="http://www.example.com/">
  <foaf:PersonalProfileDocument
      rdf:about="descriptions/Bill.foaf">
    <foaf:maker rdf:resource="#me"/>
    <foaf:primaryTopic rdf:resource="#me"/>
  </foaf:PersonalProfileDocument>
  <foaf:Person rdf:ID="me">
    <foaf:givenname>Bill</foaf:givenname>
    <foaf:mbox_sha1sum>5e22c ... 35b9</foaf:mbox_sha1sum>
    <foaf:depiction rdf:ID="images/bill.png"/>
    <foaf:knows>
      <foaf:Person>
        <foaf:name>Hillary</foaf:name>
        <foaf:mbox_sha1sum>1228 ... 2f5</foaf:mbox_sha1sum>
        <rdfs:seeAlso rdf:ID="descriptions/Hillary.foaf"/>
      </foaf:Person>
    </foaf:knows>
  </foaf:Person>
</rdf:RDF>
```

Listing 1.1. A friend-of-a-friend document

Listings 1.1 and 1.2 exhibit an overwhelming similarity. Therefore, we will only quickly discuss the points in which the data term representation deviates from the XML serialization. While attributes are given as name value pairs inside of opening tags in an XML document, they are given in round braces following a qualified name in Xcerpt$^{\text{XML}}$. Moreover, the beginning and end of an element are specified by opening and closing brackets (or braces). Namespace prefixes are declared outside of the data terms, which disallows redefinition of namespace prefixes. Nevertheless all XML documents conforming to the Namespace recommendation [25] can also be represented as an Xcerpt$^{\text{XML}}$ data term. Finally, text nodes are enclosed within quotation marks in order to be differentiated from empty element nodes.

```
declare namespace rdf "http://www.w3 ... rdf-syntax-ns#";
declare namespace rdfs "http://www.w3 ... rdf-schema#";
declare namespace foaf "http://xmlns.com/foaf/0.1/"
declare xml-base "http://www.example.com/"

rdf:RDF [
  foaf:PersonalProfileDocument
      (rdf:about="descriptions/Bill.foaf") [
  foaf:maker (rdf:resource="#me"),
  foaf:primaryTopic (rdf:resource="#me") ],
 foaf:Person (rdf:ID="#me") [
  foaf:givenname [ "Bill" ],
  foaf:mbox_sha1sum [ "5e22c ... 35b9" ],
  foaf:depiction (rdf:ID="images/bill.png"),
  foaf:knows [
   foaf:Person [
    foaf:name [ "Hillary" ],
    foaf:mbox_sha1sum [ "1228 ... 2f5" ]
    rdfs:seeAlso (rdf:ID="descriptions/Hillary.foaf") ] ] ] ]
```

Listing 1.2. A friend-of-a-friend-document written as an Xcerpt data term

1.4.2 Xcerpt^{XML} Queries: Pattern-based Filtering of Search Results

Consider the task of finding people and their curriculum vitae who study or have studied at the university of Munich. Searching for the term “LinkedIn” and “Munich” with a decent search engine returns among other search results links to pages of personal profiles of persons living in that city. The following Xcerpt query can be used to filter out other links in the search result page of Google.[8]

```
html{{
 desc div((id="res"))[[
  h2((class="hd")){ "Search Results" },
  desc h3((class="r")){{
   or(
    a((href=var Link as /.*linkedin\.com\/in\//)){{ }},
    a((href=var Link as /.*linkedin\.com\/pub\//)){{ }}
   )
  ]]
 }}
}}
```

The following features of Xcerpt must be explained to understand the above query: (in)completeness in breadth for elements and attributes, incompleteness in depth, logical variables, regular expressions and query term disjunction.

– Curly braces are used to specify subterm relationship between an element and another element or a text node. The query `h2{ "Search Results"}` finds `h2` elements with an enclosed text node with text `"Search results"`. Double curly braces signify that more subterms may be present than are specified. If more than one subterm is specified within double curly braces, they must be mapped in an injective manner, i.e. they may not match with the same subterm of the data. This injectivity requirement can be avoided by using triple curly braces `{{{ }}}`. Square

[8] We make use of the fact that all LinkedIn profile pages start either with `http://www.linkedin.com/pub/` or `http://www.linkedin.com/in/`.

parentheses may be used instead of curly braces, if the order of the subterms appearing in the query is relevant. In the presence of zero or one subterm only, using square brackets or curly braces has the same semantics. A query that uses double or triple braces or brackets is termed *incomplete in breadth*, a query with single braces or brackets only is termed *complete in breadth*.

– XML Attributes and values are given in round parentheses directly following element names. Attribute names are followed by an "=" sign and by an attribute value in quotation marks. Double parentheses may be used to state that there may be more attributes present in the data than specified in the query. Since XML attributes are always considered to be unordered, there is no way of expressing an ordered query on attributes in Xcerpt. In case of double parentheses, the attributes are said to be specified *incompletely in breadth*.
– The `desc` keyword has the same semantics as the XPath descendant axis: The subterm following the `desc` may either be a direct child of the surrounding term or nested at arbitrary depth within one of the children. A term using the `desc` keyword is termed *incomplete in depth*, the other terms are said to be completely specified in depth. As the example above shows, incompleteness significantly eases query authoring, since requires only a very basic knowledge about the structure underlying the queried data.
– Logical variables are used to extract information from an HTML or XML document. In Xcerpt^{XML} terms, variables may bind either to entire XML elements, in which case they are called *term variables*, to the labels of elements only (*label variables*), to entire attributes (*attribute variables*) or to the values of attributes only(*label variables*). Variables may additionally feature a variable restriction initiated with the `as` keyword. Variable restrictions serve to lay a restriction on the possible bindings of variables.
– Regular expressions are delimited by the sign `'/'` and can be used at the place of labels to restrict the set of XML names that are matched by an Xcerpt^{XML} query term. The query term `/ab*/`, for example, will match with the labels `a`, `ab`, `abb`, etc. only.
– Queries may be composed using the boolean connectives `and`, `or`, and `not` which have the same intuitive semantics as in logic.

1.4.3 Mining Semantic data from Microformats embedded in personal profiles.

Let us now turn to the second task of our use case. Having identified relevant URIs from the results of a search engine query, we now exploit microformats as a semantic enrichment for HTML pages to gather additional knowledge from web pages.

LinkedIn uses the microformats hcalendar, hresume, hcard, hAtom, and XFN to semantically enrich the contents of their pages. Unfortunately, the use of microformats has not been standardized, but evolves over time. Moreover, there is no underlying formal data model for microformat data as in RDF or XML. Microformats primarily use the XML attribute names `class` and `rel` for semantic information. In contrast to RDF, microformats do not use namespaces or globally unique identifiers, which makes it hard or sometimes even impossible to find out the exact semantics of an HTML fragment enriched by microformats. For example, both the hresume and the hcalendar specifications make use of a tag called `summary` for specifying either the summary of one's experience gained during a professional career or the summary of an event description.[9] With this deficiency in mind, the importance of query languages that transform semantic information embedded in HTML pages into a more precise RDF dialect becomes even more obvious. The fragment of a personal profile in Listing 1.3 pictures the use of microformats on LinkedIn and serves as further example data in this section.[10] One can observe that finding the semantic information within the HTML markup requires knowledge about the microformat standards, and that using the class attribute both for identifying elements to be formatted by stylesheets and for microformat predicate names is against the principle of separation of concerns coined by Dijkstra in [59].

```html
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en-US"
    lang="en-US">
<head><title>John Doe - LinkedIn</title></head>
<body>
<div class="hresume">
  <div class="profile-header">
    <div class="masthead vcard contact portrait">
      <h1 id="name">
        <span class="fn n">
          <span class="given-name">John</span>
          <span class="family-name">Doe</span>
        </span>
      </h1>
    </div>
  </div>
  <div id="experience">
    <h2>John Doe Experience</h2>
    <ul class="vcalendar">
      <li class="experience vevent vcard">
        <h3 class="title">Research assistant</h3>
        <h4 class="summary">
          <a href="...">University of Munich</a>
        </h4>
        <p class="organization-details">(Research industry)</p>
        <p class="period">
          <abbr class="dtstart" title="2000-02-01">February
              2000</abbr> until
          <abbr class="dtstamp" title="2008-11-24">Present</abbr>
          <abbr class="duration" title="P8Y10M">(8 years 10
              months)</abbr>
        </p>
      </li>
    </ul>
  </div>
</body>
</html>
```

Listing 1.3. A simplified personal profile page with embedded semantic information

[9] Consult the descriptions of these microformats available online `http://microformats.org/wiki/hresume` and `http://microformats.org/wiki/hcalendar` for details.

[10] The majority of the HTML markup serving presentation purposes and also most of the irrelevant content has been stripped out to shorten the presentation.

The following Xcerpt query extracts the first and last name of a Person, if she has some experience as a research assistant in some organization in Munich. Aside from that, the query extracts the duration of the working relationship between the person and the organization if present. Unlike other query subterms, the relevant subterm for the duration is marked optional, which means that the whole query is still successfull, if the optional subquery fails to match. Optional matching of subterms is only suitable if the subterm contains variables, and has also been proposed for SPARQL and other query languages. In contrast to SPARQL, however, the order of optional subterms within a query does not have any effect on the query result – see [69] for a more detailed discussion of this issue.

Listing 1.3 makes use of abbreviations for displaying information about the start, end and duration of an event. The actual date or duration is hidden within an XML attribute value that is meant for computational processing. [11]

```
html{{
 body{{
  desc{{
   desc /.*/((class="given-name")){ var FirstName },
   desc /.*/((class="family-name")){ var LastName }
  }},
  desc /.*/((class=/.*experience.*/")){{
   /.*/((class="title")){ "Research assistant" },
   /.*/((class="summary")){ /.*Munich.*/ },
   optional /.*/((class="period")){{
    /.*/((class="duration" title=var Duration)){{ }}
   }}
  }
 }}
}}
```

Listing 1.4. Finding research assistants from some organization in Munich

Listing 1.4 highlights the pecularities of matching HTML document with embedded microformat information. While element names have almost no relevance, the values of the class attributes is of primary importance. When querying plain HTML data, or XML dialects such as XMLSchema or DocBook, however, the role of the attributes will be less important, but element names will occur more often in the query. Another issue in extracting microformat information from documents is that the values of class attributes are often space separated lists of microformat predicate names such as `vcard contact portrait`. Up until now, Xcerpt has no specialized means for accessing these atomic strings in the attribute values, which results in excessive use of regular expressions. Therefore, it may be beneficial to invent a domain specific language or at least a class of query patterns that are specifically suited for querying microformat information and which would allow a less verbose notation of the query in Listing 1.4.

[11] This convention was proposed by Tantek Çelik on his blog (`http://tantek.com/log/2005/01.html\#d26t0100`) since humans prefer dates in a natural language description over a formal and concise notation, and may also deduce some information from the context.

In the following Section, we introduce the class of Xcerpt$^{\text{RDF}}$ query terms, which are geared at native and concise RDF querying.

1.5 Versatile Queries II: RDF—Examples and Patterns

In this section, the RDF processing capabilities of Xcerpt – united under the term Xcerpt$^{\text{RDF}}$ – such as data, query and construct terms particularly geared towards RDF are introduced by example. This section is structured in four parts. In Section 1.5.1 Xcerpt$^{\text{RDF}}$ data terms as a convenient way for representing RDF data are introduced. In Sections 1.5.2 and 1.5.3, Xcerpt$^{\text{RDF}}$ query and construct terms are introduced as syntactic extensions to data terms. Section 1.6.3 compares Xcerpt$^{\text{RDF}}$ to SPARQL, which is by far the most prominent RDF query language today.

1.5.1 Representation of RDF Graphs as Xcerpt$^{\text{RDF}}$ Data Terms

Many serializations for RDF Data have been proposed (RDF/XML, Notation3, Turtle, NTriples, etc.), with their inventors pursuing a set of partially competing goals: On the one hand, (i) RDF serializations are supposed to be as short as possible, on the other hand, (ii) an optimal serialization should have a canonical and unique representation for each RDF graph – put more formally, there should be an isomorphism between the set of RDF graphs and the set of RDF graph serializations. Moreover, RDF serializations should be (iii) interchangeable between software systems on the Web, and at the same time (iv) easy to author and read by humans.

RDF/XML was proposed by the W3C with the first and the third aim in mind. Due to the encoding of RDF in XML, RDF/XML is easily exchanged over the Web, and standard XML tools, such as XPath, XQuery, XSLT processors and XML Schema validators can be used to process this serialization. Furthermore, the RDF/XML syntax allows for a plethora of syntactic sugar notations that significantly reduce the verbosity of an XML encoding of data. Unfortunately, RDF/XML does not perform well in the second and fourth discipline, i.e. it is not canonical, and it is not easy to read and write by humans. Due the availability of the syntactic sugar notations, there are many different possibilities for encoding the same RDF graph, which makes parsing XML/RDF into a set of triples a major challenge, and also requires more background knowledge about the serialization format by the user than other serializations do.

Notation3 was also proposed by the W3C with the first and fourth reason in mind. Due to its non-XML serialization format and some short hand notations, it is easier to read and write for human users, and is also quite dense in comparison with other serialization formats. Notation3 does not perform well, however, taking only the second and third end into account.

Turtle being a subset of Notation3, and NTriples being a minimal subset of Turtle (and thus also of Notation3), NTriples does not provide any short hand notations and is thus significantly more verbose and redundant than Notation3. Still, it is quite readable for human users and can be easily read into or serialized from a relational database containing only one single relation for all triples in an RDF graph[12]. Due to its simplicity, NTriples comes pretty close to fulfilling the second aim: An RDF graph being a set of triples, its possible NTriples serializations only differ in the order of the triples and in the naming of the blank nodes.

With Xcerpt^{RDF} data terms, we introduce yet another format for serializing RDF graphs. Besides the common goals stated above, Xcerpt^{RDF} data terms were invented with three other goals in mind: (a) compatibility with Xcerpt^{XML} data terms, (b) extensibility to query terms involving variables and incompleteness constructs[13], and (c) support for RDF specificities such as containers and collections[14].

Consider the RDF graph displayed as an XML/RDF document in Listing 1.1 and as an Xcerpt^{XML} data term in Listing 1.2. Its representation as an Xcerpt^{RDF} data term is as follows:

```
declare namespace rdf "http://www.w3 ... rdf-syntax-ns#";
declare namespace rdfs "http://www.w3 ... rdf-schema#";
declare namespace foaf "http://xmlns.com/foaf/0.1/"
declare namespace ex "http://www.example.org/"

ex:descriptions/Bill.foaf {
  rdf:type → foaf:PersonalProfileDocument,
  foaf:maker → ex:#me,
  foaf:primaryTopic → ex:#me {
    rdf:type → foaf:Person,
    foaf:givenname → "Bill",
    foaf:mbox_sha1sum → "5e22c ... 35b9",
    foaf:depiction → base:images/bill.png,
    foaf:knows {
      _:SomePerson {
        rdf:type → foaf:Person,
        foaf:name → "Hillary",
        foaf:mbox_sha1sum "1228 ... 2f5"
        rdfs:seeAlso → base:descriptions/Hillary.foaf
} } } }
```

Listing 1.5. A friend-of-a-friend-document written as an Xcerpt^{RDF} data term

As another example consider Figure 6 from the RDF Primer [106]. Its representation as an Xcerpt^{RDF} term is as follows:

```
declare namespace exterms "http://www.example.org/terms/"
declare namespace exstaff "http://www.example.org/staffid/"

exstaff:85740 {
  exterms:address → _:A {
    !http://www.example.org/terms/city → "Bedford",
```

[12] This is a common schema for RDF stores

[13] any Xcerpt^{RDF} data term is per se also an Xcerpt^{RDF} query

[14] This last point has already been partially addressed by XML/RDF

```
    exterms:street → "1501 Grant Avenue",
    exterms:state → "Massachusetts",
    exterms:postalCode → "01730"
  }
}
```

Listing 1.6. Example from the W3C RDF Primer in Xcerpt^{RDF} notation

Similar to RDF/XML, Notation3, Turtle and SPARQL, Xcerpt^{RDF} data terms can be abbreviated using namespace prefixes in qualified names. Full URIs are distinguished from qualified names by prefixing an exclamation mark, blank nodes by the prefix `_:`, and literals by quotation marks.

In the RDF graph above, multiple statements have the blank node `_:A` as their common subject, which is factored out in the Xcerpt^{RDF} serialization. In many cases RDF statements do not only share the subject, but also the predicate, in which case also the predicate can be factored out:

```
declare namespace ex "http://www.example.org/"
declare namespace foaf "http://xmlns.com/foaf/0.1/"

ex:anna { foaf:knows → (ex:bob, ex:chuck) },
```

Xcerpt^{RDF} also supports the factorization of properties only, objects only, predicate and object, subject and object, and of all three elements – subject, predicate and object, in which case there will be one Xcerpt^{RDF} term for each RDF triple. Factoring out the predicate only could be used, for example, to represent a clique of friends, in which every member knows every other member and herself:

```
(ex:anna, ex:bob, ex:chuck) {
  foaf:knows → (ex:anna, ex:bob, ex:chuck) },
```

The RDF graph in Listing 1.6 has only a single node without incoming edges, and therefore the choice of the root of the Xcerpt^{RDF} term is trivial. RDF graphs may, however, have multiple nodes without incoming edges or none at all, or may even be entirely disconnected. In the case of no nodes without incoming edges, one can arbitrarily pick a root node for the Xcerpt^{RDF} term representation, but in the case of multiple nodes without incoming edges, and in the case of a disconnected RDF graph, the graph cannot be serialized as a single Xcerpt^{RDF} term, but only as a conjunction of terms. Therefore, the keyword `RDFGRAPH` is introduced:

```
RDFGRAPH {
  ex:anna { foaf:knows → ex:bob },
  ex:chuck { foaf:knows → ex:bob }
}
```

RDF Schema is a specification that "describes how to use RDF to describe RDF vocabularies" [112]. It therefore provides a set of URIs, with a semantics defined by RDFS entailment rules, and which are in popular use for defining new RDF ontologies. Xcerpt^{RDF} provides shorthand notations for the most common ones among them: `rdf:type`, `rdfs:range`, `rdfs:domain`, `rdf:Property` and `rdfs:Resource`.

```
ex:name { is [ex:Person -> ex:Name] }
```

The Xcerpt^{RDF} term above is a shorthand for the following Xcerpt$^{\text{RDF}}$ term:

```
ex:name {
  rdf:type → rdf:Property,
  rdfs:domain → eg:Person,
  rdfs:range → eg:Name
}
```

If the domain and/or the range of a predicate shall be left unrestricted, then the restricting classes can be simply omitted as in the Xcerpt$^{\text{RDF}}$ term `eg:name{ is [eg:Person -> ] }`. In Xcerpt$^{\text{RDF}}$ this expands to the following term:[15]

```
ex:name {
  rdf:type → rdf:Property,
  rdfs:domain → eg:Person,
}
```

Besides the RDFS vocabulary, RDF distinguishes a set of URIs for expressing reification of RDF statements and containers and collections of Resources in RDF bags, sequences, alternatives or lists. Xcerpt$^{\text{RDF}}$ provides syntactic sugar notations both for reifications on the one hand and RDF containers and collections on the other hand. Consider the following Xcerpt$^{\text{RDF}}$ term:

```
ex:bob { ex:believes →
  _:Statement1 { < ex:anna{ foaf:knows → ex:bob } >
}
```

The Xcerpt$^{\text{RDF}}$ term enclosed in angle brackets is a reified statement, and thus the entire term is equivalent to the following, significantly more verbose one:

```
ex:bob { ex:believes →
  _:Statement1 {
    rdf:type → rdf:Statement,
    rdf:subject → ex:anna,
    rdf:predicate → foaf:knows,
    rdf:object → eg:tim
  }
}
```

Whereas bags, sequences and alternatives are termed as RDF containers, and are considered to be open (i.e. there may be other elements in the container, which are not specified in the present RDF graph), RDF collections (i.e. RDF lists) are considered to be completely specified. However, this intuitive semantics is in no way reflected within the RDF/S model theory. When using only Xcerpt$^{\text{RDF}}$ shorthand notations for representing RDF graphs featuring RDF containers, collections or reification, one can be sure to respect this intuitive semantics. Xcerpt$^{\text{RDF}}$ provides the reserved words `bagOf`, `seqOf`, `altOf` and listOf to reduce the verbosity serializing RDF containers and collections. To represent a research group, one might chose the following Xcerpt$^{\text{RDF}}$ term, which would expand to four triples.

[15] Note that under the RDFS entailment rules, also the triple `ex:name rdfs:range--> rdf:Resource` would be implied. Xcerpt$^{\text{RDF}}$, however, does not enforce the RDFS semantics, since RDF/S entailment rules can be easily encoded in Xcerpt$^{\text{RDF}}$ itself.

```
_:Group1 { bagOf{ eg:anna, eg:bob, eg:chuck } }
```

Table 1.1. Syntax of Xcerpt$^{\text{RDF}}$ data terms

term	= node | node ’`{`’ arc (’`,`’ arc)* ’`}`’ | reification
node	= blank | uri | literal | qname
arc	= uri ’→’ term | container | collection
blank	= attvalueW3C
literal	= ’"’ char* ’"’ | ”′” char* ”′”
uri	= ’!’ uriW3C
qname	= qnameW3C
collection	= bag | sequence | alternative
container	= ’`listOf`’ ’`{ }`’ | ’`listOf`’ ’`{`’ term (’`,`’ term)* ’`}`’
bag	= ’`bagOf`’ ’`{ }`’ | ’`bagOf`’ ’`{`’ term (’`,`’ term)* ’`}`’
sequence	= ’`seqOf`’ ’`{ }`’ | ’`seqOf`’ ’`{`’ term (’`,`’ term)* ’`}`’
alternative	= ’`altOf`’ ’`{ }`’ | ’`altOf`’ ’`{`’ term (’`,`’ term)* ’`}`’
reification	= ’<’ term ’>’

1.5.2 Xcerpt$^{\text{RDF}}$ Query Terms

Just as in Xcerpt$^{\text{XML}}$, Xcerpt$^{\text{RDF}}$ data terms are augmented with constructs for specifying incompleteness to yield Xcerpt$^{\text{RDF}}$ query terms. Such constructs include the use of logical variables, subterm negation, subterm optionality, incompleteness in breadth and qualified descendant. While originally invented for XML processing, these constructs are also beneficial for querying RDF graphs as exemplified in Example 1.7. The query extracts variable bindings for all Persons and their nick names within an RDF graph, who know some Person with nick name ’Bill’, who in case do *not* know any other Person named ’Hillary’. As in Xcerpt$^{\text{XML}}$, the `optional` keyword is used to bind the nick name to the variable `var Nick` whenever possible, but does not cause the query to fail if the nick name is not present. Also the semantics of double curly braces and the `without` keyword is analogous to Xcerpt$^{\text{XML}}$. In Listing 1.7, the scope of the `without` and `optional` keyword is explicitly given by round parentheses. The scope of a `without` or `optional` does not have to be the entire subterm following the keyword, but may also be restricted to the edge only. Table 1.2 gives an intuition of the exact semantics of without with varying scopes by providing example data that does or does not simulate with the given query terms. The intuitive semantics for `optional` can be described by similar examples, but is left unspecified here for the sake of brevity. Note, however, that optional subterms are only useful if they contain variables for extracting data.

```
var Person{{
 optional (foaf:nick → var Nick),
 rdf:type → foaf:Person,
 foaf:knows → _:X{{
  foaf:nick → 'Bill', rdf:type → Person,
  without (
   foaf:knows → {{ _:Y{{ foaf:nick → 'Hillary' }} }}
  )
 }}
}}
```

Listing 1.7. An Xcerpt^{RDF} query term

Table 1.2. Query term simulation with different scopes for without

query term	simulating data terms	non-simulating data terms

a{{ without (b →) c }}	a{ d → c}
	a{ b → e, d →c }
		a{ b → c}
		a{ b → d, b → c }
		a{ b → d }

a{{ without (b → c) }}	a{ }
	a{ b → d }
		a{ b → c }
		a{ b → d, b → c }

a{{ b → without c }}	a{ b → d }
	a{ e → c, b → f }
		a{ b → c }
		a{ }

a{{ b → without c {{ d → e }} }}	a{ b → c }
	a{ b → c{ d → f } }
		a{ b → c{ d → e } }
		a{ }

a{{ b → (without c) {{ d → e }} }}	a{ b → f{ d → e } }	a{ b → f }
		a{ b → c }

Although the Xcerpt^{XML} constructs for specifying incomplete queries mentionend above retain their semantics in Xcerpt^{RDF}, there are some different requirements in XML and RDF processing that are also reflected in the way that Xcerpt^{RDF} variables are used in Xcerpt^{RDF} query terms.

An obvious difference between matching RDF graphs and matching XML documents is that while extracting entire subtrees from an XML document is a very common task, extracting entire RDF subgraphs from an RDF graph is less frequently used, since this may often result in the whole RDF graph being returned. Therefore, the default variable binding mechanism in Xcerpt^{RDF} is not *subgraph extraction* but *label extraction*. Therefore, the most common form of variables used in Xcerpt^{RDF} query terms are *node* and *predicate* variables. Node and predicate variables are written using the keyword var. A node (predicate) variable binds to a single node (arc) of the queried graph. *graph variables* are identified by the keyword graphVar and bind – similarly to Xcerpt^{XML} term variables – to entire subgraphs. Finally, *CBD-variables* (identified by the keyword cbdVar) bind to concise bounded descriptions[16].

[16] `http://www.w3.org/Submission/CBD/`

Another difference is that once an RDF node in an RDF graph has been identified by a query and has been bound to a variable, the very same node can be easily recovered in a subsequent query, since both URI nodes and blank nodes are uniquely named in an RDF graph, whereas an XML Document may very well contain multiple nodes having the same tag name and even the same content. XQuery and Xcerpt 2.0 deal with this problem by introducing node identity for XML elements and attributes, thereby allowing the comparison of variable bindings not only by deep equality, but also by shallow equality [68]. This distinction is not necessary in RDF processing, since the *value* of a node is already a global (in the case of resources) or local (in the case of blank nodes) identifier.

For the representation of complex values, however, the simplistic data model of RDF graphs as sets of triples is not well-suited. Here, blank nodes are used to group atomic attributes of a node together to form a complex attribute. Often, these complex attributes shall be selected together and collected in a single variable binding. This need has been addressed by the W3C consortium with the introduction of a concept known as *Concise Bounded Descriptions*. Xcerpt^{RDF} supports concise bounded descriptions by providing a special kind of variable which does not bind to the value of a node, nor to the subgraph rooted at the node, but to the concise bounded description associated with that node. Table 1.3 gives an example driven overview of the different types of variables in Xcerpt^{RDF} and their binding mechanisms.

Table 1.3. Query term simulation with variables for nodes, predicates, graphs and concise bounded descriptions

query term	data term	substitution set	
var X	a{ b → c }	{ { X ↦ a } }	(1)
a{{ b → var O }}	a{ b → c, b → _:X }	{ { O ↦ c }, { O ↦ _:X } }	(2)
a{{ var P → var O }}	a{ b → c, b → e }	{ { P ↦ b, O ↦ c }, { P ↦ b, O ↦ e } }	(3)
graphVar G	a{ b → c }	{ { G ↦ a{ b → c } } }	(4)
graphVar G as g{{ }}	a{ b → c}	{ { } }	(5)
a{{ graphVar G }}	a{ b { a }, c }	{ { G ↦ b{ a { b, c } } } }	(6)
graphVar G as var L	a{ b → c}	{ { G ↦ a{ b → c }, L ↦ a } }	(7)
cbdVar G	_:X{ b → c{ d → e } }	{ { G ↦ _:X{ b → c } } }	(8)

cbdVar G	_:X{ b → _:Y{ d → e } }	{ { G ↦
		_:X{ b → _:Y{ d→ e } } } }
		} }
			(9)

Rows 1 and 2 show the simulation of a simple Xcerpt$^{\text{RDF}}$ variable in subject and object position. Compare the binding of the graph variable *G* in row 4 with the one of the label variable *X* in row 1 under simulation with the same data term. Row 3 shows a variable in predicate position, row 5 a graph variable with a restriction, which has the same semantics as in Xcerpt$^{\text{XML}}$ (since the label *g* of the restriction does not appear within the data, the substitution set is empty).

An interesting case is row 6. Since the queried graph *d* is not a tree, but a graph, the binding for variable *G* is not a subterm of *d*, but a subgraph.

Row 7 shows the contemporary use of a graph and label variable, and rows 8 and 9 illustrate the semantics of variables for concise bounded descriptions.

Table 1.4 shows the syntax of Xcerpt$^{\text{RDF}}$ query terms as a context free grammar with terminal symbols in single quotes and the usual semantics of the meta-symbols `*` `+` `?` and `|`. The nonterminal symbols `uriW3C`, `attvalueW3C` and `qnameW3C` correspond to the syntactic definition of URIs, attribute values and qualified names in the W3C recommendation for XML[27]. The non-terminal symbol `rpe` denotes an Xcerpt$^{\text{RDF}}$ regular path expression, whose definition is omitted in this contribution for the sake of brevity.

Table 1.4. Syntax of Xcerpt$^{\text{RDF}}$ query terms

term	::= ’desc’? node | ’desc’? node ’{{’ arc (’,’ arc)* ’}}’ | ’desc’? reification
node	::= blank | uri | literal | qname | variable | graphVar | cbdVar
variable	::= ’var’ varname
varname	::= [A-Z][A-Za-z0-9*]
graphVar	::= ’graphVar’ varname | ’graphVar’ varname as term
cbdVar	::= ’cbdVar’ varname | ’cbdVar’ varname as term
arc	::= uri ’→’ term | rpe ’→’ term | container | collection
blank	::= attvalueW3C
literal	::= ’"’ char* ’"’ | ”’” char* ”’”
uri	::= ’!’ uriW3C
qname	::= qnameW3C
collection	::= bag | sequence | alternative
container	::= ’listOf’ ’{{ }}’ | ’listOf’ ’{{’ term (’,’ term)* ’}}’
bag	::= ’bagOf’ ’{{ }}’ | ’bagOf’ ’{{’ term (’,’ term)* ’}}’
sequence	::= ’seqOf’ ’{{ }}’ | ’seqOf’ ’{{’ term (’,’ term)* ’}}’
alternative	::= ’altOf’ ’{{ }}’ | ’altOf’ ’{{’ term (’,’ term)* ’}}’
reification	::= ’<’ term ’>’

1.5.3 Xcerpt$^{\text{RDF}}$ Construct Terms and Rules

Consisting of a query part and a construct part, pure Xcerpt$^{\text{RDF}}$ rules serve to transform RDF data. The query part is used to extract data from an RDF graph into sets of sets of variable bindings, also called substitution sets, and the construct part is used to reassemble these variable bindings within construct patterns, substituting bindings for variables.

Table 1.5 describes how substitution sets are applied to Xcerpt$^{\text{RDF}}$ construct terms to yield Xcerpt$^{\text{RDF}}$ data terms. Apart from the different kinds of variable bindings allowed in Xcerpt$^{\text{RDF}}$ substitution sets, the algorithm differs from the application of Xcerpt$^{\text{XML}}$ substitution sets to Xcerpt$^{\text{XML}}$ terms in the following ways:

– In accordance with the most famous RDF query languages such as SPARQL [140] and RQL [91, 29], URIs are treated as unique identifiers within an RDF graph and do not have any object identity besides the identity given by the URI itself. This convention has as an implication that a substitution set applied to different construct terms may result in semantically equivalent data terms. To see this consider rows 1 and 5 in Table 1.5. Although the Xcerpt$^{\text{RDF}}$ construct terms are syntactically different, the data terms resulting from the application of the substitution set are equivalent RDF graphs. As a result, the use of all within construct terms made up of URIs only does not change the semantics of a rule.
– Just as RDFLog [33, 34], but unlike SPARQL and other RDF query languages, Xcerpt$^{\text{RDF}}$ supports arbitrary construction of blank node identifiers. While the majority of RDF query languages does not allow blank node construction at all or only blank nodes depending on all universally quantified variables of a rule (see [34] for details), Xcerpt$^{\text{RDF}}$ and RDFLog support also construction of blank nodes that depend only on some or none of the universally quantified variables of a rule. RDFLog does this by explicit quantifier alternation, Xcerpt$^{\text{RDF}}$ on the other hand achieves the same goal by using Xcerpt's all grouping construct. To see the difference consider rows 2 and 6 in Table 1.5. In row 2 the construct contains the free variable var O, whereas in row 6 the construct term does not contain any free variable. Thus in the first case, the substitution set is divided into two substitution sets according to the binding of variable var O, and each of the substitution sets is applied to the construct term. In the second case, however, the substitution set is not divided at all, but applied as a whole to the construct term.

Special care must be taken that the result of the application of a substitution set to an Xcerpt$^{\text{RDF}}$ construct term is again an RDF graph. Guaranteeing that pure Xcerpt$^{\text{RDF}}$ programs convert RDF graphs into valid RDF graphs allows easy composition of Xcerpt programs.

Providing the same input and output format for a language is a feature of many modern query languages and is usually referred to as *answer closedness*. Popular XML query languages in general are only weakly answer closed – which means that they allow for easy authoring of programs that again produce valid XML documents, but that it still *is* possible to generate non-XML data. A notable exception to this rule is Xcerpt$^{\text{XML}}$, which is strongly answer closed in the sense that every outcome of an Xcerpt$^{\text{XML}}$ program is an XML fragment. On the other hand, the W3C languages XPath, XQuery and XSLT can also be used to output non-XML content such as PDF, Postscript, or comma separated values.

Table 1.5. Application of substitution sets to Xcerpt$^{\text{RDF}}$ construct terms

substitution set	construct term	Xcerpt$^{\text{RDF}}$ result	

$\{\{ O \mapsto c \}, \{ O \mapsto d \}\}$	a{ b → var O }	a{ b → c }
		a{ b → d }
			(1)

$\{\{ O \mapsto c \}, \{ O \mapsto d \}\}$	_:X{ b → var O }	_:X1{ b → c }
		_:X2{ b → d }
			(2)

$\{\{ S \mapsto c \}, \{ S \mapsto d\}\}$	var S{ b → a }	c{ b → a }
		d{ b → a }
			(3)

$\{\{ S \mapsto c \}, \{ S \mapsto d\}\}$	var S{ b → _:X }	c{ b → _:X1 }
		d{ b → _:X2 }
			(4)

$\{\{ O \mapsto c \}, \{ O \mapsto d \}\}$	a{ all b → var O }	a{ b → c, b → d }	(5)
$\{\{ O \mapsto c \}, \{ O \mapsto d \}\}$	_:X{ all b → var O }	_:X{ b → c, b → d }	(6)

$\{\{ O \mapsto c \}, \{ O \mapsto d \}\}$	a{ b → var O{ e → f } }	a{ b → c{ e → f } }
		a{ b → d{ e → f } }
			(7)

$\{\{ G \mapsto$ a{ b → c } $\}\}$	graphVar G	a{ b → c }	(8)
$\{\{ G \mapsto$ a{ b → c } $\}\}$	d{ e → graphVar G }	d{ e → a{ b → c } }	(9)

Definition 1 (Answer Closedness). *A web query language is called* answer closed, *if the following conditions are fulfilled:*

1. *data in the queried format can be used as queries*
2. *the result of queries is again in the same format as the data*

A web query language is called weakly answer closed, *if condition (2) is possible; it is called* strongly answer closed, *if condition (2) is always enforced.*

The assurance of answer closedness in Xcerpt$^{\text{RDF}}$ must take the following two thoughts into account:

– *Abidance of RDF triple constraints*. The evaluation of query terms may bind node variables to literals or blank nodes. RDF graphs, however, do not allow literals in subject or predicate position or blank nodes in predicate position.
– *Abidance of RDF graph constraints*. Xcerpt$^{\text{RDF}}$ supports four different kinds of variables: node variables, predicate variables, graph variables and concise bounded description variables. In general, it is only safe to substitute variables in construct terms by bindings of variables of the same type. Depending on the data, bindings for node, graph and concise bounded description variables may degenerate to plain URIs, and therefore it may be safe to substitute them for predicate or node variables.

With the above two restrictions in mind, there are three different possibilities for implementing answer closedness in Xcerpt$^{\text{RDF}}$.

- *Static Checking of Bindings:* Before an Xcerpt$^{\text{RDF}}$ program is run, it is checked that predicate variables in the construct term are also used as predicate variables in the query term, and the same for graph variables, node variables and CBD variables. To be more precise, the semantics of graph and CBD variables only differ within the query term, and thus a CBD variable binding may be substituted for a graph variable in the construct term. Moreover, the binding of a predicate variable may be substituted for a label variable in the construct term, since predicate variables always bind to URIs. On the other hand, bindings of node variables, may *not* be substituted for predicate variables. While static checking of variable bindings ensures that all terms constructed by Xcerpt$^{\text{RDF}}$ programs are valid RDF graphs, certain tasks, such as using URIs of nodes of a source graph in predicate position in the target graph, are impossible to achieve with this technique.
- *Dynamic Checking of Variable Bindings:* Dynamic checking of variable bindings is a sensible choice if there is reason to assume that the query author has some knowledge about the data to be queried. It is more flexible than static checking in the sense that a larger number of tasks can be realized, but is less reliable in the sense that runtime errors may occur.
- *Casting of Variable Bindings* unites the best of static checking of variable bindings (i.e. no runtime errors) and dynamic checking of variable bindings (i.e. a higher degree of flexibility). Consider the sources of runtime errors that may occur with dynamic checking of variable bindings – examples for each case are given in Table 1.6.
  - A literal or blank node bound to a node variable is substituted for a predicate variable in a construct term. Such triples are simply omitted from the resulting RDF graph.
  - A subgraph bound to a CBD or graph variable is substituted for a node variable in a construct term. In this case the subgraph rooted at the occurrence of the node variable in the construct term and the binding of the variable are merged.
  - A subgraph g rooted at a URI u and bound to a graph variable is substituted for a predicate variable in a construct term. The graph g is cast to u.
  - A subgraph g rooted at a blank node b and bound to a graph variable or CBD variable is substituted for a predicate variable. Since blank nodes may not appear in subject positions, the resulting triple is not included in the Xcerpt$^{\text{RDF}}$ result.
  - A literal *lit* is substituted for a node variable L appearing in subject position in the construct term. In this case a fresh blank node B is substituted for the variable instead of the literal. If L additionally appears in object position, also the literal itself is substituted for L, but the triples containing *lit* in subject position are omitted.

Since the last alternative gives an operational semantics to programs which would be either considered invalid under the first approach or would throw runtime errors under the second, Xcerpt$^{\text{RDF}}$ favors the casting of variable bindings. We acknowledge, however, that the first approach may make more sense for unexperienced users in that it is easier to understand, and that the second approach may uncover errors in the authoring of Xcerpt$^{\text{RDF}}$ programs, which would pass unnoticed by the third approach.

Table 1.6. Application of substitution sets to Xcerpt$^{\text{RDF}}$ construct terms with casting of variable bindings

substitution set	construct term	Xcerpt$^{\text{RDF}}$ result
{ { V ↦ _:X } }	a{ var V → b }	
{ { V ↦ _:X } }	a{ var V → b, c → d }	a{ c → d }
{ { V ↦ 'literal1' } }	a{ var V → b, c → d }	a{ c → d }
{ { G ↦ a{ b → c } } }	graphVar G{ d → e }	a{ b → c, d → e }
{ { G ↦ a{ b → c } } }	d{ var G → e }	d{ a → e }
{ { G ↦ _:X{ b → c } } }	d{ var G → e }	_:X{ b → c }
{ { L ↦ 'literal1' } }	a{ b → { var L{ c → d } }	a{ b → _:X { c → d }, b → 'literal1' }

1.6 Versatile Queries III: Rules for Separation of Concern and Reasoning

Having introduced queries for both XML and RDF data, this section combines both features to realize the truly versatile use case already sketched in Section 1.4. Starting out from the result pages for the terms "LinkedIn Munich" of a popular Web search engine, links to relevant LinkedIn profile pages are extracted by the use of rich XML query patterns with logical variables. In a second step, the profile pages are retrieved and semantic microformat information is exploited to gather reliable information about the users. Finally, in a third step, this information is enriched by semantic information from FOAF profiles in RDF format using the RDF processing capabilities of Xcerpt.

In this use case Xcerpt's capability of handling XML query terms and RDF construct terms in the same rule (and the other way around) comes in particularly handy. As in pure XML querying and in pure RDF querying, the interface between querying and construction is a substitution set. Substitution sets generated by XML query terms differ in the allowed variable types from substitution sets generated by RDF query terms. As a result, there must either be a way to transform XML substitution sets to RDF substitution sets and reversely, or the application of XML substitution sets to RDF construct terms and the application of RDF substitution sets to XML construct terms must be defined. While both ways are feasible, we present here the first alternative, since it is less involved.

1.6.1 Xcerpt$^{\text{XML}}$ Query Terms and Xcerpt$^{\text{RDF}}$ Construct Terms and vice versa in the Same Rule

Note that it is Xcerpt's underlying principle of clear separation of querying and construction that allows for, e.g, an XML query term in a rule body and an RDF construct term in the head of the same rule. The applicability of this design principle remains untouched if further types of query and construct terms are introduced (e.g. for topic maps or queries aimed at specific microformats or at pages of a Semantic Wiki). The only requirement for these new types of queries and construct terms are the definition of the following four algorithms: (1) a simulation algorithm matching queries with data and returning a substitution set (a set of set of variable bindings),[17] (2) an application algorithm for substitution sets that fills in bindings for logical variables occuring in a construct term[18], (3) a mapping from variable bindings in the new format to variable bindings in the other formats (until now only XML and RDF) and finally (4) a mapping from XML and RDF variable bindings to variable bindings in the new format.

The following list defines informally the mapping of XML bindings to RDF and reversely.

– The Xcerpt$^{\text{RDF}}$ URI `!http://www.example.org/#foo` is mapped to the Xcerpt$^{\text{XML}}$ qualified name `eg:#foo` with the namespace prefix `eg` bound to the namespace `http://www.example.org/`. We adopt the convention that the Xcerpt$^{\text{RDF}}$ URI is split into namespace and local name at the last '`/`', but other methods are also conceivable.
– The Xcerpt$^{\text{RDF}}$ blank node `_:B` is mapped to the Xcerpt$^{\text{XML}}$ element name `_:B`.
– The Xcerpt$^{\text{RDF}}$ literal "some literal" maps to the Xcerpt$^{\text{XML}}$ text node "some literal"[19].
– The Xcerpt$^{\text{RDF}}$ qualified name `eg:anna` is mapped to the Xcerpt$^{\text{XML}}$ qualified name `eg:anna`. An appropriate namespace binding is added to the Xcerpt$^{\text{XML}}$ term. Implementations may choose to expand the qualified name to a URI *u*, and map *u* instead.
– The Xcerpt$^{\text{RDF}}$ term `a{ b → c }` maps to the Xcerpt$^{\text{XML}}$ term `a{ b{ c } }` in correspondance to past work on querying XML serializations of RDF with Xcerpt [21]. Similarly, the Xcerpt$^{\text{RDF}}$ term `_:X{ a → b{ c → ''another literal'' } }` is mapped to the Xcerpt$^{\text{XML}}$ term `_:X{ a { b { c { ''another literal'' } } } }`.
– The Xcerpt$^{\text{RDF}}$ shorthand notation `ex:name{ is [ex:Person →ex:Name ] }` is expanded to its corresponding unabbreviated term as introduced in Section 1.5. Then this longer notation is mapped to an Xcerpt$^{\text{XML}}$ term as described above.
– The Xcerpt$^{\text{RDF}}$ reification term `a{ believes → _:S{ < b{ c → d } }` is mapped to the Xcerpt$^{\text{XML}}$ term `a { believes _:S { xcrdf:reification { b { c { d } } } } }` with the namespace prefix `xcrdf` bound to `http://www.xcerpt.org/xcrdf`.
– The Xcerpt$^{\text{RDF}}$ term `_:X { bagOf { a, b, c } }` is mapped to the Xcerpt$^{\text{XML}}$ term `_:X { xcrdf:bag { a, b, c } }`. Expansion to the normalized RDF syntax and applying the standard mapping to Xcerpt$^{\text{XML}}$ terms could also be introduced. The choice of the conversion is, however, not of primary importance, as long as all information present in the Xcerpt$^{\text{RDF}}$ term is preserved. Additional

[17] See Tables 1.2 and 1.3 for an informal description of this algorithm for Xcerpt$^{\text{RDF}}$
[18] Table 1.5 gives the relevant ideas for this algorithm in Xcerpt$^{\text{RDF}}$
[19] We leave the details of treating typed RDF literals and literals with a language tag as future work.

transformation rules can be easily written to change the XML outcome and be provided as an Xcerpt$^{\text{XML}}$ module (See [11] for more about Xcerpt modules). Xcerpt$^{\text{RDF}}$ sequences, alternatives and lists are treated in the same manner.

– The Xcerpt$^{\text{XML}}$ qualified name `eg:a` is mapped to the Xcerpt$^{\text{RDF}}$qualified name `eg:a` and the binding for the namespace prefix `eg` is preserved.
– The Xcerpt$^{\text{XML}}$ unqualified name `a` is mapped to the Xcerpt$^{\text{RDF}}$qualified name `xcxml:a` with the namespace prefix `xcxml` bound to the namespace `http://www.xcerpt.org/xcxml`. Note that the RDF graph data model does not allow for local names other than blank nodes. The unqualified name is not mapped to a blank node to avoid naming conflicts with other resources that may be contained in the resulting RDF Graph.
– The Xcerpt$^{\text{XML}}$ term `eg:a[ eg:b, eg:c ]` is mapped to the Xcerpt$^{\text{RDF}}$ term `eg:a{ xcxml:child → eg:b, xcxml:child → eg:c }`, and the binding for the namespace prefix `eg` is preserved. Note that since RDF graphs are always considered to be unordered, Xcerpt$^{\text{RDF}}$ does not provide square brackets, and the information about the order is lost in this mapping. Encodings of XML terms as RDF graphs that preserve the order are conceivable.
– The Xcerpt$^{\text{XML}}$ term `eg:a(id="2"){ eg:b }` is converted to the Xcerpt$^{\text{RDF}}$ term `eg:a{ xcxml:child → eg:b }`, i.e. XML attributes are not mapped to Xcerpt$^{\text{RDF}}$ terms. Attribute names and values may, however, also be inserted into an RDF graph by binding them to label variables. Also in this case, a different kind of mapping may be chosen, but it turns out that for the applications considered in this report, this simple mapping suffices.

1.6.2 Transforming LinkedIn embedded Microformat information to DOAC and FOAF

Reconsider the Xcerpt$^{\text{XML}}$ query term in Listing 1.4. It extracts bindings for the variables `FirstName`, `LastName` and `Duration`. It is easy to construct RDF data from those variable bindings with an Xcerpt rule featuring the construct term in Listing 1.6.2.

```
declare namespace doac "http://ramonantonio.net/doac/0.1/"
declare namespace foaf "http://xmlns.com/foaf/0.1/"

_:Person {
  rdf:type → foaf:Person,
  foaf:firstName → var FirstName,
  foaf:surname → var LastName,
  all doac:experience → _:Exp {
    doac:title → "Research Assistant",
    doac:duration → var Duration
  }
}
```

Note the semantics of the `all` construct in Listing 1.6.2. The all construct serves to collect a set of variable bindings within a data term to be constructed. The number of data terms generated for construct term *c* preceded by an `all` construct depends on the set of free variables inside of c, and the substitution set which is applied to the construct term. A variable v is free within a term t, if it does not occur within the scope of an `all` construct inside of t. Thus the variable `Duration` is free within the term `doac:duration ...`, but not inside of the entire construct term of Listing 1.6.2. The set of free variables in the term c :=`doac:experience → _:Exp { ... }` following the `all` keyword is the unary set $\{Duration\}$. The substitution set applied to the construct term is thus separated according to the bindings of the variable `Duration` only. Then each of the resulting substitution sets is applied to c independently and included as a subterm of the outermost `foaf:Person` label. Whenever a substitution set is applied to a term with a blank node, a new instantiation of this blank node is created, as showcased in Table 1.5. This is a major difference to application of substitution sets to terms starting with URIs.

Alternatively, one might want to create a single RDF bag enumerating the working relationships a person has had. This could be achieved by the following Xcerpt$^{\mathrm{RDF}}$ construct term:

```
_:Person {
  rdf:type → foaf:Person,
  foaf:firstName → var FirstName,
  foaf:surname → var LastName,
  _:Experiences {
    bagof {
      all _:Exp {
        doac:title → var Title,
        doac:duration → var Duration
      }
    }
  }
}
```

Once the microformat information from the LinkedIn page is transformed to the more precise RDF representation at the aid of this rule, it can be combined with RDF data located anywhere on the Web. These FOAF documents can be discovered in a very similar fashion as has been done for the LinkedIn profile pages in Section 1.4.

Since LinkedIn does not provide the hash sums of email-addresses or other globally unique identifiers for persons within their profile pages, combining the extracted RDF information will rely on simple joins over the names of people, which is not particularly reliable – see [98] for an overview of the problems that may occur.

With OpenID [130] becoming the de facto standard for distributed authentication and single-sign-on on the Web and with the largest corporations involved in online activities such as Google, Yahoo, Microsoft, etc already joining the bandwagon, it seems likely that also LinkedIn will provide an open identifier within its profiles. Also the extension of the FOAF vocabulary to provide for OpenIDs within FOAF profiles is already discussed. In the presence of this information, the combination of the collected microformat data and other RDF resources can easily and reliably achieved using Xcerpt$^{\mathrm{RDF}}$.

1.6.3 State of the Art: the SPARQL Query Language and its Extensions

With the publication of the SPARQL W3C recommendation on January 2008, SPARQL has become the first query language that has been standardized by a major standardization body. In contrast to most other languages that have been proposed for RDF querying, SPARQL is, due to its triple syntax, quite easy to understand and use for programmers familiar with relational query languages.

In this section, SPARQL is introduced by example, its semantics according to [123] is recapitulated, and several extensions to SPARQL are presented. Throughout the presentation, the commonalities and differences to Xcerpt$^{\text{RDF}}$ are highlighted.

A SPARQL query consists of the three building blocks *pattern matching part*, *solution modifiers* and *output*. In addition there are four different kinds of query forms. Arguably the most popular one is the *select* query form, which is inspired by SQL and returns so-called solution sets, the counterpart of Xcerpt substitution sets in SPARQL. An example of a *select* query is given in Listing 1.8. In case of a select query, the output part of the query is a selection of distinguished variables, i.e. the specification of the variables of interest in the query. If no variable bindings are of interest, the *ask* query form is to be used. It simply gives a yes/no answer to the question if a given query pattern is entailed by the RDF graph being queried. A useful query form for RDF *graph transformations* is the *construct* query form, which does not return single values, but entire RDF graphs as a result. There are, however certain limitations to the blank node construction (in database theory termed *value invention*) in the SPARQL construct query form, see [39].A final query form is given by the *describe* key word which pays attribute to the fact that a blank node identifier returned as a variable binding in a SPARQL ask-query is somewhat useless, since it only asserts the fact that something exists, and cannot be reused in a follow-up query to extract further information about the resource in question. When using the *describe* query form, not only single identifiers are returned as variable bindings, but also *descriptions* of resources. The exact nature of a resource description is left unspecified in the SPARQL recommendation, but a sensible solution would be the one of *Concise Bounded Descriptions* [142].

The SPARQL query form which is most similar to Xcerpt$^{\text{RDF}}$ rules is the *construct* query form. Xcerpt$^{\text{RDF}}$ does not distinguish between query forms, but is strongly answer closed in the sense that every Xcerpt$^{\text{RDF}}$ data term is also a Xcerpt$^{\text{RDF}}$ query, and in that every result of an Xcerpt$^{\text{RDF}}$ query is again an RDF graph. While SPARQL *construct* queries are answer closed, the remaining query forms are not. However, SPARQL *ask* and *select* queries can be simulated by *construct* queries. Similarly, boolean queries can be formulated in Xcerpt$^{\text{RDF}}$ by interpreting the empty RDF graph as false and all other RDF graphs as true, and tuple-generating queries can be expressed in Xcerpt$^{\text{RDF}}$ by wrapping the tuples within RDF containers or similar constructs. *Describe* queries are expressed in Xcerpt$^{\text{RDF}}$ by using concise-bounded-description variables.

All four SPARQL query forms make use of the pattern matching part, which is described next.

SPARQL graph patterns SPARQL is weakly answer closed in the sense that any RDF graph is also a valid SPARQL graph pattern. But only in the case of the construct query form, also the result of a SPARQL query is again an RDF graph. The syntax of

SPARQL graph patterns resembles the one of Turtle, but is augmented with variables. Listing 1.8 (from [140]) shows a query to retrieve the name and email address of persons within an RDF graph using the FOAF vocabulary. With the term *graph pattern*, one refers to the set of triples within curly braces in lines 4 to 5. The select-clause serves to specify the *distinguished* variables of the query. Any variable appearing within the graph pattern, but not within the select-clause is called a *non-distinguished* variable. The terms *distinguished* and *non-distinguished* variables have thus the same meaning as in conjunctive queries in database theory.

```
1 PREFIX foaf: <http://xmlns.com/foaf/0.1/>
  SELECT ?name ?mbox
3 WHERE
    { ?x foaf:name ?name .
5     ?x foaf:mbox ?mbox }
```

Listing 1.8. A simple SPARQL query

SPARQL allows the selection of variables that do not appear within the graph pattern as shown in Listing 1.9. The empty query pattern matches with any RDF graph, and the variable `?x` in the select clause does not appear within the query pattern. In database theory, such rules are said to violate the principle of range-restrictedness. In fact the intuitive semantics of non-range-restricted rules is unclear and varies from one language to another. While according to [140] Listing 1.9 is supposed to return a single solution with no binding for the variable `?x`, unbound variables are forbidden within construct clauses of SPARQL queries. In Prolog, on the other hand, the non ground fact `p(X)` simply remains uninstantiated and can be unified with ground bodies of other queries such as `p(a)`.

```
SELECT ?x
WHERE {}
```

Listing 1.9. A non-range-restricted SPARQL query matching with arbitrary RDF graphs

Since queries such as the one in Listing 1.9 can also be expressed with the SPARQL ask query form, and since SPARQL does not allow any kind of rule-chaining, non-range-restricted queries do not add to the expressive power of the SPARQL language, but cause the semantics of the language to be more complex than it needs to be.

The graph pattern in Listing 1.8 is termed a *basic graph pattern*. It consists of two *triple patterns*, which are ordinary RDF triples except that subject, predicate and object may be replaced by SPARQL variables. Basic graph patterns may contain *filter expressions* in addition to a set of triple patterns. Filter expressions use the boolean predicates '`=`', '`bound`', '`isIRI`' and others to construct atomic filters. Additionally the logical connectives '`&&`' for logical conjunction, '`||`' for logical disjunction and '`!`' for logical negation are used to construct compound filters from atomic ones. Atomic and compound filters are used to eliminate sets of variable bindings that do not fulfill the filter requirements.

Besides basic graph patterns, SPARQL provides group graph patterns that may either be *unions of graph patterns*, *optional graph patterns* or *named graph patterns*. *Unions of graph patterns* are similar to disjunctions in the bodies of rules in logic programming. For the query to succeed, only one of the graph patterns in the union must be successful, and the solution sets from all graph patterns in the union are collected to yield the solution set for the union. *Optional graph patterns* are patterns that may bind additional variables besides the ones present in the non-optional parts of a graph pattern, not causing the entire query to fail if the optional graph pattern fails. In contrast to unions of graph patterns, the non-optional part is obliged to match. *Named graph patterns* are introduced into the SPARQL language, because Semantic Web databases may hold multiple RDF graphs, each identified by a URI. To explain the concept of querying named graphs in SPARQL, the notion of a *dataset* must be introduced. A *dataset* is a pair (d,N) where d is the default graph to be queried, and N is a set of named graphs. Datasets are specified by the FROM and FROM NAMED clauses in SPARQL. Whereas the default graph is the merge of all RDF graphs specified in the FROM clause, the FROM NAMED clauses specify the set N of named graphs, and remain unmerged. The GRAPH key word must subsequently be used to refer to named graphs in a WHERE clause as Listing 1.10 (taken from [126]) illustrates.

```
SELECT ?N WHERE { ?G foaf:maker ?M .
          GRAPH ?G { ?X foaf:name ?N } }
```

Listing 1.10. Querying named graphs in SPARQL

As [126] points out, the query in Listing 1.10 is somewhat unintuitive, since SPARQL engines compliant with the W3C specification will search for answers to the triple pattern `?X foaf:name ?N` only in named graphs, but not in the default graph. The notion of *named graphs* is discussed in more detail in [45], and can be compared to grouping XML data in XML documents.

Blank nodes in SPARQL graph patterns Blank nodes in SPARQL graph patterns act in the same way as non-distinguished variables, and therefore cannot be used to reference specific blank node identifiers within an RDF graph. Hence, one could substitute an arbitrary blank node for the variable `?x` in Listing 1.8 and still obtain the same result.[20]

Before proceeding, we will quickly discuss this treatment of blank nodes in SPARQL. When issuing a query with a blank node, newcomers to the SPARQL language may have five different expectations in mind:

– *Syntactic equality:* The blank node in the query is supposed to match only with the data that uses exactly the same blank node identifier, as it is the case for URIs in graph patterns. While this is a valid desire, it would fall into the domain of syntactic processing of RDF data. A query on two equivalent RDF graphs should obviously return equivalent answers. But what is a sensible notion of equivalence in this context? As with all data items in information processing, one may introduce several equivalence relationships for RDF graphs. One such equivalence relationship is bi-entailment, and it is arguably the most sensible one for RDF graphs. Another such equivalence relationship would be syntactic equality, and there is certainly the

[20] Note that one could *not* use a blank node at the place of the other two variables in Listing 1.8, since they are distinguished.

necessity to compare RDF graphs for syntactic equality, but then we could also simply consider them as plain text files and run a UNIX `diff` command to test them for equality. With the decision for *syntactic equality* for blank nodes in queries, one would obtain different results for equivalent RDF graphs (under bi-entailment), and for this fact the decision of SPARQL not to use *syntactic equality* is a sensible one.

– *Treatment as non-distinguished variables:* The blank node is supposed to act as a non-distinguished variable as explained above. One minor problem with this understanding is that there are two alternative ways of specifying the same query, which may be confusing for new-comers to the language. Another more important issue with this solution is that while SPARQL remains answer closed in the sense that any RDF graph can be used as a SPARQL query, the answer to such a query would not only be graphs that are equivalent or contain an equivalent graph, but also graphs that are more specific. The simple SPARQL graph pattern `_:X a b` will also return true on the RDF graph `a b c`.
– *Banning of blank nodes within queries*: As the inclusion of blank nodes within queries does not add expressive power to SPARQL graph patterns, an obvious approach is to ban blank nodes from graph patterns. This approach has the advantage that SPARQL users cannot be fooled to assume a different semantics of blank nodes in graph patterns other than non-distinguished variables. On the other hand, this approach has the obvious drawback that SPARQL is not answer closed in the sense that an RDF graph containing blank nodes cannot be viewed as a SPARQL query.
– *Treatment as ordinary variables:* Since blank nodes are viewed as existentially quantified variables in RDF graphs, one might view them as plain variables in queries as well, and specify in the select-clause if they are to be treated as distinguished variables or non-distinguished variables. This solution has the plain advantage that any RDF graph can be viewed as a query, but shares the same deficiencies with respect to answer closedness as treating them as non-distinguished variables. Clearly this approach would mean that there is no longer the necessity for SPARQL variables.
– *Matching blank nodes only:* A final intuition query authors may have in mind is that blank node identifiers in queries must be mapped to blank node identifiers in the data only. None of the above approaches can express this semantics. The graph pattern `_:X b c` would thus return true when evaluated on the graphs `_:X b c` and `_:Y a b`, but it would not match with `a b c`. Thus with answer closedness in mind, this approach ensures that an RDF graph q considered as a SPARQL query only matches with RDF graphs that are equivalent or have a subgraph equivalent to q. The major drawback of this solution is, however, that the same query may once return true for an RDF graph g_1 and false for an equivalent (under bi-entailment) RDF graph g_2. To see this, consider again the query `_:X b c` and the graphs g_1 := `_:Y b c, a b c` and `a b c`. Under the light of this deficiency and with the availability of the filter predicate `isBlank` in SPARQL that can be used for imitating this blank node semantics, it is a good choice not to adopt this treatment of blank nodes in SPARQL graph patterns.

Testing RDF Graphs for Equivalence in SPARQL None of the above solutions are completely satisfactory in that they do not allow the specification of a query q that returns true on exactly the equivalence class $\Sigma_{\Leftrightarrow}(g)$ induced by RDF bi-entailment for an arbitrary graph g containing a blank node.

Note that SPARQL query patterns cannot express the above query even in the absence of blank nodes. Consider the RDF graph g `a b c` consisting of a single triple. Evaluating g as a SPARQL query pattern will yield all RDF graphs that *contain* g, but there is no way of expressing a query that will find all *equivalent* graphs.

In other words, a SPARQL basic graph pattern q returns true on an RDF graph g iff g rdf-entails[21] $n(q)$ where the normalization operator n replaces variables in q by blank nodes (multiple occurrences of the same variable by the same blank node identifier, and distinct variables by distinct blank nodes, that do not occur anywhere else in q). Hence, with basic SPARQL graph patterns it is only possible to demand that something *be entailed* by the graph g to be queried, but not to restrict the entailments of g. The development of the language Xcerpt$^{\text{RDF}}$, on the other hand, is influenced by the assumption that query authors would like to both demand some entailments from a graph as well as demand that something is *not* entailed by it.

There is, however, the possibility to express such queries in SPARQL at the aid of optional graph patterns, SPARQL filter constructs, and the SPARQL `bound` predicate. The query in Listing 1.11 only returns true for the one-triple graph `a b c`. For all other graphs it returns false. The graph pattern first ensures that the triple `a b c` is in fact contained in the RDF graph. Secondly it uses an optional pattern to find other triples in the graph. The filter inside the optional pattern makes sure that the optional pattern does not match with a triple other than `a b c`. The second filter expression makes sure that the optional graph pattern was unsuccessful by testing for a binding of the variable `?x`.

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
ASK
WHERE { a b c .
   OPTIONAL { ?x ?y ?z
     FILTER ( ?x != a || ?y != b || ?z != c )
   }
   FILTER (!bound(?x))
 }
```

Listing 1.11. A query that only matches with a graph consisting of a single triple `(a b c)`

Before proceeding to the study of the complexity and semantics of SPARQL, we will quickly discuss how to test for equivalence with RDF graphs containing blank nodes. Consider the graph $g =$ `_:X b c . a b d .` consisting of two triples only with a single occurrence of a single blank node. When formulating a SPARQL query to return true on exactly the set of RDF graphs equivalent to g, one first needs to test for the presence of the two triples and then for the absence of triples that are different from the two ones given in the graph. While the query in Listing 1.12 is all but trivial to figure out, testing graphs for equivalence in SPARQL becomes even more complex in the presence of multiple occurrences of the same blank node identifier, since in this

[21] There are different variants of RDF entailment. In this section we mean simple RDF entailment when when speaking of RDF entailment only.

case it does not suffice to test for the absence of single triples only, but one has to test for the absence of multiple triples connected via blank nodes.

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
ASK
WHERE {
   a b c .
   ?blank b d
   OPTIONAL { ?x ?y ?z
     FILTER ( ( ?x != a || ?y != b || ?z != c ) ) &&
             ( !(isBlank(_?x1)) || ?y1 != b || ?z != d ) )
   }
   FILTER (!bound(?x1))
  }
```

Listing 1.12. A query that only matches with a graph consisting of a single triple `(a b c)`

Obviously the queries in Listing 1.11 and 1.12 are much more complicated than they need to be. This is due to the absence of explicit negation in SPARQL, a design decision that makes implementation easier and circumvents the non-monotonicity of negation as failure.

Semantics and Complexity of SPARQL [123] recursively defines the semantics of SPARQL query patterns in terms of relational algebra operators as follows:

– The semantics $[[t]]_G$ of a possibly non-ground triple t evaluated over an RDF graph G is the set of mappings μ such that the domain of μ is the set $Var(t)$ of variables in t and the application $\mu(t)$ of the mapping μ to t is in G. The application of a mapping μ to a triple pattern t is simply the triple pattern with the variables in t replaced by their bindings in μ.
– The semantics $[[(P_1 \text{ AND } P_2)]]_G$ of a conjunction of query patterns evaluated over the RDF graph G is defined as the set $\{[[P_1]] \bowtie [[P_2]]\} = \{\mu_1 \cup \mu_2 \mid \mu_1 \in [[P_1]], \mu_2 \in [[P_2]], \mu_1$ and μ_2 are compatible$\}$ of unions of compatible pairs of mappings of P_1 and P_2. In this context two mappings are termed *compatible* if they coincide on the bindings of their common variables. The semantics of the conjunction can thus be thought of as the natural join over the relations defined by the conjuncts.[22]
  In [126] the notion of compatibility of pairs of mappings is refined to *brave compatibility*, *cautious compatibility* and *strict compatibility*. While in the absence of unbound variables within mappings, all three notions of compatibility coincides, in the presence of unbound variables, only the brave compatibility coincides with compatibility as understood by [123].
  – Two mappings σ_1 and σ_2 are *bravely compatible* if they coincide on the bindings of their common bound variables. Brave compatibility hence does not restrict the bindings of variables that are unbound in either σ_1, σ_2 or both.
  – σ_1 and σ_2 are *cautiously compatible* if for all common variables – no matter if bound or unbound – the bindings coincide.

[22] Note that the terms *relation* and *sets of mappings* can be used interchangeably here.

  - σ_1 and σ_2 are *strictly compatible* if they are cautiously compatible and if additionally there is no common variable of σ_1 and σ_2 which is unbound in both.
- The semantics of a graph pattern $[[P_1 \text{ OPT } P_2]]_G$ including an optional construct over an RDF graph G is defined as the left outer join between $[[P_1]]$ and $[[P_2]]$.
- Finally the semantics $[[P_1 \text{ UNION } P_2]]$ of a union of two graph patterns is defined as the union of $[[P_1]]$ and $[[P_2]]$.

[123] extend the semantics to SPARQL queries including filter expressions and show some important properties of SPARQL queries:

- Generally the expressions

  ```
  (P1 AND (P2 OPTIONAL P3))
  ```

  and

  ```
  (P1 AND P2)OPTIONAL P3))
  ```

  are not semantically equivalent, but they are equivalent for the class of *well-defined* graph patterns introduced in the same work.
- In the presence of optional patterns, AND is only commutative for well-designed graph patterns.

Some results on the complexity of query evaluation in SPARQL from [123] are the following:

- The combined complexity of SPARQL graph patterns involving only AND and FILTER expressions is in $O(|P| \cdot |D|)$ where $|D|$ is the size of the data and $|P|$ is the size of the query. This result is based on the assumption that the application of a mapping μ to a triple t is achieved in a constant amount of time, independently of the number of variables in μ.
- The combined complexity of SPARQL graph patterns involving AND, FILTER and UNION is NP-complete. The proof is by polynomial reduction of the satisfiability problem of propositional logic formulas in conjunctive normal form to SPARQL queries.
- The combined complexity of SPARQL graph patterns including AND UNION and OPTIONAL is PSPACE-complete, independently of the presence or absence of FILTER expressions.
- The data complexity of SPARQL graph patterns is in LOGSPACE.

1.6.4 Extensions of SPARQL

SPARQL being the most popular RDF query language and the only one which has been standardized by some standardization organization such as the W3C, it has received considerable attention from the research community. Its expressiveness and complexity has been formally studied, and as a result of its limited expressiveness, extensions of SPARQL in different directions have been proposed. With the absence of path expressions in SPARQL, nSPARQL[124] has been suggested to enhance the expressive power of SPARQL into this direction. The necessity of combined processing of XML and RDF has been acknowledged by XSPARQL[7], an extension of XQuery to RDF processing at the aid of SPARQL WHERE and CONSTRUCT clauses. Just as SQL allows the deletion and insertion of data and creation of new tables, SPARQL update [139] and SPARLQ+[23] extend SPARQL with facilities to manipulate and create RDF graphs. Finally [44], [126] and [135] eliminate the restriction of SPARQL to single rules by allowing possibly recursive multi-rule programs.

nSPARQL nSPARQL[124] is an extension of SPARQL to support arbitrary-depth navigation in SPARQL queries. It arose from the need to answer queries for finding all nodes reachable from a given node via a given predicate name, a disjunction of predicate names or simply for finding all transitively connected nodes. The syntax of nSPARQL is heavily influenced by the syntax of XPATH, and nSPARQL borrows the notions of axes, node tests, reverse axes, step expressions, and path predicates from XPATH. While path expressions in XPATH evaluate to a set of nodes of an XML document, path expressions in nSPARQL evaluate to a set of *pairs* of nodes within of an RDF graph. This is due to the fact that XPATH expressions are always evaluated with respect to a context node, while this is not necessarily the case for nSPARQL expressions.

The following examples illustrate the syntax and semantics of nSPARQL path expressions evaluated over an RDF graph G:

- `next::a` allows the navigation from one node in an RDF graph to another node via an edge labelled `a` in a composed nSPARQL path expression. It evaluates to all pairs of nodes connected via a predicate labeled a: $\{(x,y) \mid (x,a,y) \in G\}$. The axis $\texttt{next}^{-1}$ can be used to navigate in the reverse direction.
- `edge::a` allows the navigation from a node x to an edge y within an RDF graph, if the graph contains the triple (x,y,a). It evaluates to $\{(x,y) \mid (x,y,a) \in G\}$. The axis $\texttt{edge}^{-1}$ is used to navigate from predicates of triples to their subjects.
- `node:a` allows the navigation from an edge x to a node y if the corresponding triple has subject a. It evaluates to $\{(x,y)|(a,x,y) \in G\}$. $\texttt{node}^{-1}$ is used for navigating in the reverse direction.
- nSPARQL path expressions are combined just like XPATH step expressions by the `/` sign: The nSPARQL expression `next::a/next::b` finds pairs of nodes connected via two triples with predicate names `a` and `b` over an arbitrary intermediate node. The URI of the intermediate node can be checked by using the self axis: `next::a/self::c/next::b`.
- The evaluation of nested nSPARQL path expressions is more complex. The semantics of `edge::[exp]` is given by $\{(x,y) \mid \exists z,w.(x,y,z) \in G \wedge (z,w) \in [[exp]]_G\}$, where $[[exp]]_G$ is the semantics of exp over G. Nested path expressions including the axes `self`, `next` and `node` are similarly involved.

SPARQLeR A different approach for extending SPARQL with regular path expressions is taken by the language SPARQLeR described in [103]. In contrast to nSPARQL, entire paths are bound to so-called path variables, which are distinguished from ordinary SPARQL variables in that they are prefixed by `%` instead of `?`. The bindings of path

[23] `http://arc.semsol.org/home`

variables are themselves represented as RDF sequences, which allows to put further restrictions on the bindings in SPARQL WHERE clauses, as the following example from [103] demonstrates:

```
CONSTRUCT %path
WHERE { r %path s . %path rdfs:_1 p . }
```

Listing 1.13. A simple SPARQLeR path query

The query in Listing 1.13 finds all directed paths between a resource `r` and a resource `s` that have `p` as the first predicate. Bindings for the path variable `%path` in the above query are of the form $p_1, n_1, p_2, n_2, \ldots, p_i, n_i, p_{i+1}$, such that the triples (r, p_1, n_1), (n_1, p_2, n_2), …, (n_{i-1}, p_i, n_i) and (n_i, p_{i+1}, s) are in the queried graph. Since these bindings are represented as RDF sequences (as exemplified in Listing 1.14), triples in the same WHERE clause can be used to put restrictions on the bindings to path variables.

```
_:Path1 rdfs:_1 p1,
_:Path1 rdfs:_2 n1,
_:Path1 rdfs:_3 p2,
...
```

Listing 1.14. The RDF representation of bindings to SPARQLeR path variables

Since bindings to SPARQLeR path variables are represented as RDF sequences represented by blank nodes, the use of path variables within SELECT query forms hardly makes sense. Imagine Listing 1.13 with the SELECT keyword at the place of the CONSTRUCT keyword. The result of this query is a list of blank nodes generated by the SPARQLeR query generator, which means that the only information returned is the number of paths found within the queried graph. To deal with this inconvenience, SPARQLer introduces a `list` operator that extracts all resources from the paths. In the case of multiple bindings for a path variable, however, the application of the list operator merges the resources from all paths into a single list, thereby preventing the user from recognizing the actual paths.

SPARQLeR provides a second method for constraining paths at the aid of a ternary `regex` method to be used within FILTER clauses of SPARQLeR queries. The first argument to this method is the name of the path variable whose bindings are to be constrained, the second one is a regular path expression, and the third are options specifying whether the path must be directed, if it must be made up of schema classes, instances, or literals, and if `rdfs:subPropertyOf` inferencing is to be considered. SPARQLeR regular path expressions allow alternatives, concatenation, Kleene's star, wildcards, negations and reverse predicates. The SPARQLeR `length` method is used to find paths of a minimal, maximum or exact length.

While SPARQLeR seems to be a sensible suggestion for an extension of SPARQL, there are two obvious points of criticism:

– The fact that predicate names can be specified within path expressions, but subjects and objects cannot, seems to be an arbitrary design choice which is not motivated in [103].

– Representing bindings to variables as RDF sequences that are not part of the original RDF graph and allowing these RDF sequences to be queried within the SPARQLeR WHERE clause may be confusing for novices in that the WHERE clause is successfully evaluated on a graph which does not entail every single triple of the clause.

XSPARQL [7] advocates the reuse of plain XML and HTML data of the Web as RDF data on the Semantic Web, and vice versa and introduces the notions of *lifting* – i.e. transforming “syntactic” XML data into “semantic” RDF data – and lowering – transforming RDF data into XML. Starting out from the insight that current tools and languages are not adequate for translating between syntactic and semantic web data, they propose an integration of SPARQL into XQuery, which they dub XSPARQL, together with use-cases and a formal semantics. Since it aims at being data-versatile in the same sense as Xcerpt does, we take a closer look at XSPARQL in this section.

```
<relations>
 <person name="Alice">
   <knows>Bob</knows>
   <knows>Charles</knows>
 </person>
   <knows>Charles</knows>
 </person>
 <person name="Charles/>
</relations>
```

Listing 1.15. XML example data

```
@prefix foaf: <...foaf/0.1/> .
_:b1 a foaf:Person;
    foaf:name _:b2;
    foaf:knows _:b3 .
_:b2 a foaf:Person;
    foaf:name "Bob";
    foaf:knows _:b3 .
_:b3 a foaf:Person;
    foaf:name "Charles" .
```

Listing 1.16. RDF example data

Listing 1.17 shows how the lifting task is solved in XSPARQL for the example data given in Listings 1.15 and 1.16. In line 3 all names of persons of the XML input file are selected. Names are either given as the `name` attribute of a `person` element or as XML text nodes within `knows` elements. In order to make sure that the list `$persons` contains each name exactly once, duplicates are elminitated in the `where` clause by testing the absence of elements on the `following` axis that contain the same name. In this way duplicates are eliminated and only the last occurrence of a name is selected. In line 6, a numeric identifier is computed for each person which serves to construct unique blank nodes in the SPARQL construct pattern starting at line 8. The `construct` keyword is not part of the XQuery syntax, but newly introduced in XSPARQL to mark the beginning of a SPARQL construct pattern. Inside of SPARQL construct patterns, XQuery code is embedded within curly braces. In this way nested XSPARQL queries are constructed. While the outer XSPARQL query (lines 3 to 10) serves to represent the persons found in the XML source as RDF blank nodes with associated names and type, the inner SPARQL query translates the acquaintance relationships. Note that the triples constructed in line 18 are duplicates of the ones constructed in line 10, i.e. this line is superflous.

```
declare namespace foaf="...foaf/0.1/";
declare namespace rdf="...-syntax-ns#";
```

```
let $persons := //*[@name or ../knows] return
for $p in $persons
let $n := if ( @p[@name] ) then $p/name else $p
let $id := count($p/preceding::*) + count($p/ancestor::*)
where not(exists($p/following::*[@name=$n or data(.)=$n]))
construct
  _:b{$id} a foaf:Person;
         foaf:name { data($n) }.
  { for $k in $persons
    let $kn := if ( $k[@name] ) then $k/@name else $k
    let $kid := count($k/preceding::*) + count($k/ancestor::*)
    where $kn = data(//*[@name=$n/knows) and
          not(exists($kn/../following::*[@name=$kn or
              data(.)=$kn]))
    construct
      _:b{$id} foaf:knows _:b{$kid} .
      _:b{$kid} a foaf:Person .
  }
```

Listing 1.17. Lifting in XSPARQL

XSPARQL does not set out to be a query language that natively supports XML and RDF querying in an intuitive and coherent way. Instead it explores how SPARQL can be integrated into XQuery, how the semantics of this integration can be defined and proposes an implementation on top of existing XQuery and SPARQL engines. XSPARQL succeeds in its coherent treatment of schema heterogeneous RDF/XML files, and due to the large expressiveness of XQuery it allows the formulation of many queries not expressible in SPARQL alone. On the other hand it suffers from the following deficiencies:

- *Intertwined querying and construction.* As can be observed in Listing 1.17, there is no clear separation of querying and construction in XSPARQL queries, a deficiency which is inherited from XQuery. While it is clear that there are queries that cannot be expressed by a single rule with a single query and construction pattern, this is not the case for the query above.
- *Complicated blank node construction.* An RDF query language should support automatic construction of blank nodes without the need of computing blank node identifiers within a program. Since blank node construction is essentially the same as the introduction of skolem terms within logic programs, languages such as RDFLog and Xcerpt achieve the same result in a much easier and straightforward way.
- *Absence of path patterns.* While XSPARQL inherits the complexity of XQuery, it suffers also from the limitations of SPARQL such as no support for containers, collections and reification, and limited support for negation. Above all, XSPARQL lacks rich path patterns to navigate RDF graphs at arbitrary depth, such as the ones proposed by nSPARQL and SPARQLeR.
- *Jumbling of query paradigms.* Due to the popularity of XQuery as an XML query language and SPARQL as an RDF query language, Listing 1.17 is easy to understand for most people familiar with (Semantic) Web querying. For people unfamil-

iar with one or both of these languages, it may be confusing that a functional language such as XQuery is intermingled with a rule based language such as SPARQL. With Xcerpt^{RDF}we introduce a purely rule based language based on the clear design principles of Xcerpt.

SPARQL update Similar as for the XML query language XQuery, SPARQL has been conceived primarily as a data *selection* language, not as a data *manipulation* language. In fact, the SELECT, DESCRIBE and ASK query forms of SPARQL can only be used to *extract* parts of a graph, not to manipulate data or construct new data. The SPARQL CONSTRUCT query form allows limited transformations between one RDF dialect to another, but cannot be used to modify existing RDF stores. The W3C member submission *SPARQL update* sets out to elminate this restriction.

SPARQL update consists of two sets of directives – one for updating graphs and the other for graph management. The set of directives for updating existing RDF graphs with SPARQL update constists of the following seven commands:[24]

– The `DELETE DATA FROM` directive is used to delete a set of ground triples from a named or the default graph. In the latter case, the `FROM` keyword is omitted.
– The `INSERT DATA INTO` statement is used to insert a new set of ground triples into an existing graph identified by a URI. If the triples are to be inserted into the default graph, then the `INTO` keyword is omitted.
– The `MODIFY` operation consists of a delete and an insert statement (see below) issued on the same graph.
– The `DELETE FROM ... WHERE` operation is used to delete a set of triples from a graph. In contrast to the `DELETE DATA` operation discussed above, this command may specify the triples to be deleted in a non-ground form, i.e. with SPARQL variables bound in the **`WHERE`** clause. If the WHERE clause consists of the empty graph pattern, this command is indeed equivalent to the `DELETE DATA` operation above. In case the `FROM` keyword is omitted, the default graph is manipulated.
– `INSERT FROM ... WHERE` is the non-ground version of the `INSERT DATA` command. Its relationship to `INSERT DATA` is analogous to the relationship from `DELETE FROM ... WHERE` to the `DELETE DATA` operation. Together with the `DELETE FROM ... WHERE` operation, this operation can be used to move data from one RDF graph to another.
– The `LOAD` primitive copies all RDF triples from one named graph to another named graph or the default graph.
– The `CLEAR` primitive removes all triples from the default graph, or a named graph. It can be simulated by a `DELETE FROM ... WHERE` operation selecting all triples of a graph.

Graph management in SPARQL update is achieved by the two operations `CREATE GRAPH` and `DROP GRAPH` which have the exact same semantics as the SQL operations `CREATE TABLE` and `DROP TABLE`. Only when a graph has been created by the `CREATE GRAPH` operation it is available for modification by one of the seven above mentioned manipulation directives.

[24] We only briefly sketch the commands for the sake of brevity.

To sum up, SPARQL update is a straight-forward extension of SPARQL to include mechanisms for creating new and changing existing RDF graphs, much inspired by SQL. The difference between the Web considered as a huge database and ordinary databases is, however, that the Web is open and generally readable and processable by any person or computer connected to the Internet. As a result RDF graphs will more likely to be reasoned with and transformed than updated. Write access to RDF graphs is restricted to the content provider, but deriving new knowledge from existing one, which is the fundamental use case for Semantic Web use-cases, is possible for all Web users and will be achieved with rule languages, not update languages. Under these considerations, update primitives have been excluded from Xcerpt$^{\text{RDF}}$.

SPARQL and Rules [126] defines translation rules for SPARQL rules to datalog rules and thus opens up the possibility to rule chaining, i.e. the translation of multiple SPARQL rules to Datalog and the combined evaluation of the resulting rule set by a logic programming engine, thus allowing intermediate results to be constructed and queried. This extension gives SPARQL an obvious boost in expressivity (recursion) and affects its termination properties. In the following, the translation procedure from SPARQL to Datalog given in [126] is quickly illustrated by an example, as it opens up the possibility for easy implementations also of single rule SPARQL queries on top of existing logic programming engines.

For this purpose reconsider the SPARQL query in Listing 1.18 and the RDF graph in Listing 1.19 available via the URL `http://www.example.org/bob`. The result of the translation is given in Listing 1.20.

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?name ?mbox
FROM http://example.org/bob
WHERE
  { ?x foaf:name ?name .
    ?x foaf:mbox ?mbox }
```

Listing 1.18. A simple SPARQL select-query

```
_:B foaf:name bob .
_:B foaf:nick bobby .
_:B foaf:mbox bob@example.org .
```

Listing 1.19. RDF Graph with some FOAF information

```
triple(S, P, O, default) :- rdf(http://example.org/bob, S, P,
     O) .
answer_1( (Name, Mbox), default) :-
  answer_2(vars(Name, X), default),
  answer_3(vars(Mbox, X), default) .
answer_2(vars(Name, X), default) :- triple(X, foaf:name, Name,
     default) .
answer_3(vars(Mbox, X), default) :- triple(X, foaf:name, Mbox,
     default) .
```

Listing 1.20. Translation of the SPARQL query in Listing 1.18 to Datalog with external predicates

The translation makes use of the external predicate `rdf` that takes four arguments: the graph to be queried as input, and the subject, predicate and object of triples as output. The external predicate `rdf` can thus be used to enumerate all triples within an RDF graph given by the input URI. The first rule in Listing 1.20 defines the 4-ary relation `triple`. In the case of multiple `FROM` or `FROM NAMED` clauses in the original SPARQL query, the relation `triple` will obviously be defined by the corresponding number of clauses. Since Listing 1.18 only contains conjunctions of triple patterns, but no UNION, OPTIONAL or FILTER expression, the translation remains of manageable size, and we focus the discussion of the tranlsation procedure on conjunctive triple patterns.

As can be observed in Listing 1.20, each triple pattern in the SPARQL query translates to a single Datalog rule, and each conjunction of triple patterns translates to a rule with body atoms referencing the rules obtained by the translation of its conjuncts. As expected, disjunctions (UNION) of triple patterns are translated to sets of rules. For details on the tranlsation procedure, involving more complex SPARQL queries with FILTER and OPTIONAL, the interested reader is referred to [126].

While reusing existing rule languages together with the enormous body of knowledge about their semantics, evaluation methods and complexity is certainly a sensible way for designing a rule language for the Semantic Web, the approach taken in [126] is not completely satisfactory for the following reasons:

– Blank node construction in rule heads has been largely ignored, especially the different modes of blank node construction as pointed out by [34].
– This approach inherits the weakness of SPARQL concerning negation: implicit negation as failure is provided by the combination of the `OPTIONAL` directive and the `unbound` predicate. For newcomers to the language this feature is hard to discover, and should be better declared as what it is.
– The expressivity of SPARQL graph patterns is limited when compared to languages that allow possibly recursive path expressions such as Versa on RDF graphs or Conditional XPath[111, 110] and Xcerpt on XML documents. This limitation is obviously inherited by all rule extensions to SPARQL.
– Rule extensions of SPARQL remain pure RDF query languages and therefore cannot deal with the versatility requirements for modern Web query languages.

1.7 Versatile Semantics

Having given an informal, example-driven introduction to the language Xcerpt, its evaluation principles and intuitive semantics in the preceding sections, this section introduces the precise semantics for Xcerpt *query terms* through a formal definition of *query term simulation* (Section 1.8), and *programs* through an iterative fixpoint procedure (Section 1.9). Previous publications on the semantics of Xcerpt have considered the class of *stratified Xcerpt programs* only. Section 1.9 extends the semantics of Xcerpt programs to the class of *locally stratified programs*, which is a true superset of the set of stratifiable Xcerpt programs, and which is inspired by the notion of local stratification in logic programming [52]. In Section 1.10 the well-founded semantics for general logic programs is adapted to Xcerpt, thereby also giving a semantics to programs that are not locally stratified. Although not formally proven, we argue that locally stratifiable Xcerpt programs have a two-valued well-founded model which coincides with the model computed by the iterative fixpoint procedure over its local stratification.

While this section transfers the notion of local stratification and well-founded semantics to Xcerpt only, the proposed method can be applied to any other rule-based language with non-monotonic term negation and disjunction-free heads, that provides the same interface to terms as Xcerpt does (defined in Section 1.3).

1.8 Versatile Semantics I: Simulation as Foundation for Versatile Querying

Simulation between Xcerpt terms is inspired by rooted graph simulation [115, 85], but is by far more involved since Xcerpt terms feature constructs for specifying incompleteness in depth, breadth, and order, allow variables, regular expressions and negated subterms. This section formally defines a subset of Xcerpt$^{\text{XML}}$[25] variables, descendant constructs, subterm negation, incompleteness in breadth and with respect to order, multiple variables, multiple occurrences of the same variable, and variable restrictions. In comparison to full Xcerpt$^{\text{XML}}$ query terms as described in [68, 133] and for the sake of brevity, this definition does not include term identifiers and references, non-injective subterm specifications, optional subterms, qualified descendants, label variables, and the new syntax for XML attributes. Based on this definition of Xcerpt$^{\text{XML}}$ query, construct and data terms, ground and non-ground query term simulation is defined as the formal semantics for the evaluation of Xcerpt$^{\text{XML}}$ query terms on semi-structured data.

Definition 2 (Xcerpt$^{\text{XML}}$ query term). *Query terms over a set of labels $\mathcal{N}$, a set of variables $\mathcal{V}$, and a set of regular expressions $\mathcal{R}$ are inductively defined as follows:*

- for each label $l \in \mathcal{N}$, `l{{ }}` and `l{ }` are atomic query terms. l is a short hand notation for `l{{ }}`. The formal treatment of square brackets in query terms is omitted in this contribution for the sake of brevity.
- for each variable $X \in \mathcal{V}$, `var X` is a query term
- for each regular expression $r \in \mathcal{R}$, `/r/{{ }}` and `/r/{ }` are query terms. `/r/` is a shorthand notation for `/r/{{ }}`. With $\mathcal{L}(r)$ we denote the set of labels matched by r, i.e. the language defined by the regular expression.
- for each variable $X \in \mathcal{V}$ and query term t, `var X as t` is a query term. t is called a *variable restriction* for X.
- for each query term t, `desc t` is a query term and called *depth-incomplete* or *incomplete in depth.*

[25] Chapter 1.3 introduces both Xcerpt$^{\text{RDF}}$ and Xcerpt$^{\text{XML}}$ query, construct and data terms. In this section we concentrate on Xcerpt$^{\text{XML}}$ terms, but most of the results and design principles also apply to Xcerpt$^{\text{RDF}}$ terms. We write "Xcerpt term" to denote the abstract concept of terms in both Xcerpt$^{\text{RDF}}$ and Xcerpt$^{\text{XML}}$, and "Xcerpt$^{\text{XML}}$ term" to refer to Xcerpt$^{\text{XML}}$ terms only.

– for each query term t, `without` t is a query term and called a *negated subterm*.
– for each query term t `optional` t is an *optional query term*.
– for each label or regular expression l and query terms $t_1, \dots, t_n$ with $n \geq 1$,

$$q_1 = l\texttt{\{\{ } t_1\texttt{, ..., } t_n \texttt{ \}\}}$$
$$q_2 = l\texttt{\{ } t_1\texttt{, ..., } t_n \texttt{ \}}$$

are query terms. q_1 is said to be *incompletely specified in breadth*, or simply *breadth-incomplete*, whereas q_2 is *completely specified in breadth*, or simply *breadth-complete*.

A variable X is said to *appear positively* in an Xcerpt$^{\text{XML}}$ query term q, if it is included in q not in the scope of a `without` construct. It *appears negatively* within q if it is included within the scope of a `without` construct. Note that the same variable may appear both positively and negatively within q – e.g. `X` within `a{{ var X, without var X }}`.

Definition 3 (Xcerpt$^{\text{XML}}$ data terms). *An* Xcerpt$^{\text{XML}}$ data term *is a ground Xcerpt$^{\text{XML}}$ query term that does not contain the constructs* `without`*,* `optional`*,* `desc`*, regular expression and double braces.*

Definition 4 (Xcerpt$^{\text{XML}}$ construct terms). Xcerpt$^{\text{XML}}$ construct terms *over a set of variables $\mathcal{V}$ and a set of labels $\mathcal{L}$ are defined as follows:*

– *an Xcerpt$^{\text{XML}}$ data term d over $\mathcal{L}$ is a construct term*
– *for each variable $X \in \mathcal{V}$,* `var X` *is a construct term*
– *for a construct term c,* `all` *c is a construct term*
– *for a construct term c,* `optional` *c is a construct term*
– *for a construct term c, and a sequence of variables $X_1, \dots, X_k \in \mathcal{V}$* `all` *$c$* `group by` *$\{X_1, \dots, X_k\}$ is a construct term*
– *for a label $l \in \mathcal{L}$ and set of construct terms $c_1, \dots, c_n$, $l\{c_1, \dots, c_n\}$ is a construct term.*

In the following, we let $\mathcal{D}$ and $\mathcal{Q}$ denote the set of all Xcerpt$^{\text{XML}}$ data and query terms, respectively.

A query term and a data term are in the simulation relation, if the query term "matches" the data. Matching Xcerpt$^{\text{XML}}$ query terms with data terms is very similar to matching XPath queries with XML documents – apart from the variables and the injectivity requirement in query terms. The formal definition of simulation of a query term with semi-structured data is somewhat involved. To shorten the presentation, we first introduce some notation:

Definition 5 (Injective and bijective mappings). [26]

Let $I := \{t^1_1, \dots, t^1_k\}$, $J := \{t^2_1, \dots, t^2_n\}$ be sets of query terms and $\pi : I \Rightarrow J$ be a mapping.

[26] This definition of injectivity and bijectivity concerns the subterms – or nodes – of a query term only. Therefore it is also referred to as *node injectivity*. In previous publications about Xcerpt, we have used *position injectivity* instead, which concerns the edges between parent and child terms. In the absence of references (as in Definition 4), however, node and position injectivity are semantically equivalent. Therefore, and for the sake of simplicity, we use node injectivity in this contribution.

– π *is* injective*, if all $t^1_i, t^1_j \in I$ satisfy $t^1_i \neq t^1_j \Rightarrow \pi(t^1_i) \neq \pi(t^1_j)$.*
– π *is* bijective*, if it is injective and for all $t^2_j \in J$ there is some $t^1_i \in I$ such that $\pi(t^1_i) = t^2_j$.*

We use the following abbreviations to reference parts of a query term q:

$l(q)$: the string or regular expression used to build the query term. For a variable v, $l(v)$ is undefined.
$ChildT(q)$: the set of direct subterms of q
$ChildT^+(q)$: the set of positive direct subterms (i.e. those direct subterms which are not of the form *without*...),
$ChildT^-(q)$: the set of negated direct subterms (i.e. the direct subterms of the form *without*...),
$Desc(q)$: the set of direct descendant subterms of q (i.e. those of the from *desc*...),
$SubT(q)$: the direct or indirect subterms of q, i.e. all direct subterms as well as their subterms.
$ss(q)$: the subterm specification of q. It can either be *complete* (single curly braces) or *incomplete* (double curly braces).
$vars(q)$: the set of variables occurring somewhere in q.
$pos(q)$: q', if q is of the form `without` q', q otherwise.

Definition 6 (Label subsumption). *A term label l_1 subsumes another term label l_2 iff l_1 and l_2 are strings and $l_1 = l_2$, or l_1 is a regular expression and l_2 is a string such that l_1 matches with l_2, or l_1 and l_2 are both regular expressions and l_1 matches with any label that l_2 matches with.*

Definition 7 (Ground query term simulation). *Let q be a ground query term*[27] *and d a data term. A relation $\mathcal{S} \subseteq (SubT(q) \cup \{q\}) \times (SubT(d) \cup \{d\})$ is a simulation of q into d if the following holds:*

– $q\ \mathcal{S}\ d$
– *if $q := l_1\{\{q_1, \ldots, q_n\}\}\ \mathcal{S}\ l_2\{d_1, \ldots, d_m\} =: d$ then l_1 must subsume l_2, and there must be an injective mapping $\pi : ChildT^+(q) \to ChildT^+(d)$ such that $q_i\ \mathcal{S}\ \pi(q_i)$ for all $i \in ChildT^+(q)$. Moreover, there* must not *be a $q_j \in ChildT^-(q)$ and $d_l \in ChildT^+(d) \setminus range(\pi)$ such that $pos(q_j)\ \preceq\ d_l$ (note the recursive reference to '$\preceq$' here).*
– *if $q := l1\{q_1, \ldots, q_n\}\ \mathcal{S}\ l2\{d_1, \ldots, d_m\} =: d$ then l_1 must subsume l_2, and there must be a bijective mapping $\pi : ChildT^+(q) \to ChildT^+(d)$ such that $q_i\ \mathcal{S}\ \pi(q_i)$ for all $i \in ChildT^+(q)$. We impose no further requirements on the set $ChildT^-(q)$ of negated direct subterms of q. The totality of π already ensures that there is no extension of π to some element $q_j \in ChildT^-(q)$ such that $pos(q_j)\ \preceq\ d_l$ for some $d_l \in ChildT^+(d) \setminus range(\pi)$. Therefore the semantics of query terms is independent from the presence of negated direct subterms within breadth-complete query terms.*
– *if $q = desc\ q'\ \mathcal{S}\ d$ then $q'\ \mathcal{S}\ d$ or $q'\ \mathcal{S}\ d'$ for some subterm d' of d.*

We say that q simulates into *d (short: $q \preceq d$) if and only if there is a relation $\mathcal{S}$ that satisfies the above conditions. To state the contrary we write $q \npreceq d$.*

[27] For the sake of brevity we assume that q does not contain any optional subterms.

Since every Xcerpt$^{\text{XML}}$ data term is also a query term, the above definition of simulation between a query term and a data term can be extended to a relation between pairs of query terms. For the sake of brevity this full definition of *extended ground query term simulation* is given in the appendix of [35].

The existence of a ground query term simulation states that a given data term satisfies the conditions encapsulated in the query term. Many times, however, query authors are not only interested in checking the structure and content of a document, but also in extracting data from the document, and therefore query terms may contain logical variables. To formally specify the data that is extracted by matching a query term with a data term, the notion of non-ground query term simulation is introduced (Definition 8). Substitutions are defined as usual, and the application of a substitution to a query term is the consistent replacement of the variables by their images in the substitution.

Definition 8 (Non-ground query term simulation). *A query term q with variables simulates into a data term d iff there is a substitution $\sigma : Vars(q) \rightarrow \mathcal{D}$ such that $q\sigma$ simulates into d.*

In some cases query terms are not expressible enough or inconvenient for specifying a query in the body of a rule. Conjunctions of query terms are needed if more than one resource is queried and the results are to be joined. Disjunctions of query terms are convenient to extract data from different resources and wrap them into a common XML fragment or RDF graph. Finally the absence of data simulating with a given query term is tested by query negation. The notion of a query combines conjunctions, disjunctions and negations of query terms:

Definition 9 (Xcerpt query).

– *an Xcerpt query term is an Xcerpt query*
– *for a set of Xcerpt queries $q_1, \ldots, q_n$, the conjunction $\mathcal{C} :=$ `and(`$q_1, \ldots, q_n$`)`, the disjunction $\mathcal{D} :=$ `or(`$q_1, \ldots, q_n$`)` and the negation $\mathcal{N} :=$ `not(`q_1`)` are Xcerpt queries. If a variable X appears positively within a q_i $(1 \leq i \leq n)$ then it also appears positively within $\mathcal{C}$ and $\mathcal{D}$, but negatively within $\mathcal{N}$. If X appears nevatively within q_i, it also appears negatively within $\mathcal{C}$, $\mathcal{D}$ and $\mathcal{N}$.*

Definition 10 (Xcerpt rule, goal, fact, program). *Let q be a query over a set of labels $\mathcal{L}$, a set of variables $\mathcal{V}$ and a set of regular expressions $\mathcal{R}$ and c a construct term over $\mathcal{L}$ and $\mathcal{V}$. Then* **`CONSTRUCT`** `c FROM q END` *is an* Xcerpt rule, `GOAL c FROM q END` *is an* Xcerpt goal, *and* **`CONSTRUCT`** `c END` *is an* Xcerpt fact. *An* Xcerpt program *is a sequence of range-restricted Xcerpt rules, goals and facts.*[28]

The construct term c is called the *head* of an Xcerpt rule or goal, the query q is called its *body*. An Xcerpt fact can also be written as an Xcerpt rule with an empty body. An Xcerpt rule, goal or fact is called *range restricted*, if all variables that appear in its head also appear positively in its body. In a forward chaining evaluation of a program, the distinction between goals and facts is unnecessary. In a backward chaining evaluation,

[28] Since facts and goals are a kind of rules, we refer to Xcerpt programs as a sequence of rules in the following.

however, the goals are the starting point of the resolution algorithm. In contrast to Logic programming, goals are not a single term only, but an entire rule to ensure answer closedness of Xcerpt programs. Especially for the task of information integration on the Web, answer closedness is indispensable.

1.9 Versatile Semantics II: Negation and Versatile Queries—Local Stratification

While Section 1.8 defines the semantics of single query terms and queries, this section defines the semantics of Xcerpt rules and programs. Special attention is laid on the interplay between simulation unification and non-monotonic negation in rule bodies.

The problem of evaluating rule based languages with non-monotonic negation has received wide-spread attention throughout the logic programming community (See [10] and [31] for surveys). A multitude of semantics have been proposed for such languages (program completion semantics, stable-model semantics [74], well-founded semantics [147], inflationary semantics [104]). Especially the well-founded and stable-model semantics have been found to comply with the intuition of program authors and are therefore implemented by logic programming engines such as XSB [132] and DLV [61]. Several classes of logic programs have been defined for which some of the above mentioned semantics coincide. Among these classes are definite programs, stratifiable programs, locally stratifiable programs [129] and modularly stratifiable programs [131]. The well-founded semantics and the stable model semantics coincide on the class of locally stratifiable programs.

In the following we introduce stratifiable and locally stratifiable Xcerpt programs. In adapting these concepts to Xcerpt, one has to pay close attention to the differences introduced by the richer kind of unification employed.

Definition 11 (Stratification). *A stratification of an Xcerpt program P consisting of the rules $r_1, \dots r_n$ is a partitioning of $r_1, \dots r_n$ into strata $S_1, \dots, S_k$, such that the following conditions hold:*

- *All facts are in S_1.*
- *If a rule r_1 contains a positive query term q that simulates with the contstruct term c of another rule r_2, then r_1 positively depends on r_2, and r_1 is in the same or a higher stratum than r_2.*
- *If a rule r_1 contains a negated query term `not` q such that q simulates with the construct term c of another rule r_2, then r_1 negatively depends on r_2 and is in a strictly higher stratum than r_2.*

Given the stratification of a program P, its semantics can be defined by the iterative fixpoint procedure suggested for general logic programs. For finite programs, stratification is decidable. However, there are Xcerpt programs, such as the one in Listing 1.21, that are not stratifiable, but which may be evaluated bottom up.

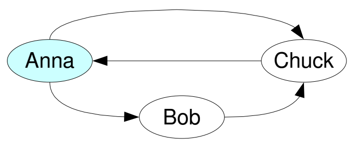
Listing 1.21 is a formulation of the single source shortest path problem over a directed social graph, which is given by the facts (lines 1 to 5) in Listing 1.21 and which is depicted in Figure 1.3. The program computes for each node n in a directed graph the shortest distance to some source node s, in this case `anna`.

This program uses a slight extension of Xcerpt's term syntax. The term

`Acquaintance[ anna, ≤i]`

simulates with the data terms `Acquaintance[ anna, j]` if and only if i and j are natural numbers and $j \leq i$. Furthermore, the terms `Acquaintance[ anna, ≤i]` and `Acquaintance[ anna, i]` simulate with `Acquaintance[ anna, >j]` if and only if $i > j$. The symbol '>' can be interpreted as a hint by the programmer to the evaluation engine, that a rule can only be used to derive atoms with integer values greater than a certain natural number. The example in Listing 1.21 serves to illustrate the problems and challenges for defining the semantics and evaluation of possibly recursive rule programs with non-monotonic negation and rich unification. These challenges are encountered independent of the specific kind of rich unification, be it SPARQL query evaluation, Xcerpt query term simulation, or XPath query evaluation.

Fig. 1.3 Social graph corresponding to the facts in Listing 1.21



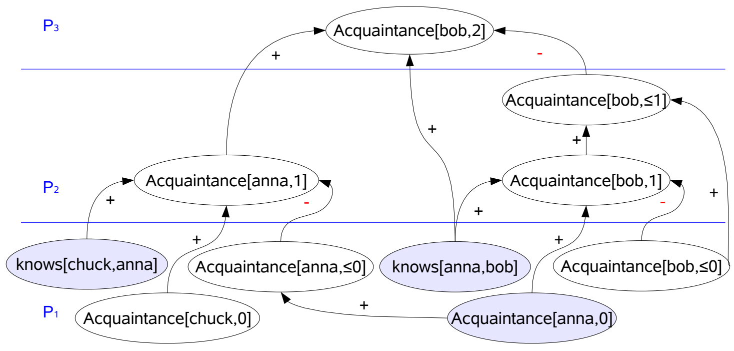
To see that Program P in Listing 1.21 is not stratifiable, consider the negated query term `not` q, with $q =$ `Acquaintance [ var P, ≤ var D ]` in the body of the only rule of P. q simulates with the head $h =$ `Acquaintance [ var P, D + 1 > 0]` of the same rule. Thus the rule should be in a strictly higher stratum than itself, which is a contradiction.

```
CONSTRUCT knows[ anna, bob ] END
CONSTRUCT knows[ bob, chuck] END
CONSTRUCT knows[ anna, chuck ] END
CONSTRUCT knows[ chuck, anna ] END

CONSTRUCT Acquaintance[ anna, 0 ] END

CONSTRUCT
  Acquaintance[ var P, var D + 1 ]
FROM
  and (
    Acquaintance[ var P', var D ],
    knows[ var P, var P'],
    not ( Acquaintance[ var P, ≤ D ] )
END
```

Listing 1.21. Single source shortest path problem for the source node 'anna'

To see that P can nevertheless be evaluated in a bottom up manner, consider a ground instance g of the recursive rule in Listing 1.21. The term constructed by the head of g contains an integer value i which is exactly by one larger than the integer values of terms that may simulate with (negated or positive) query terms in the body of g. Thus, in a bottom up evaluation of the program, we may first compute the fixpoint of the program considering only terms containing the integer value 0, followed by the fixpoint computation for terms with the value 1, and so on. Since a valid rule application will only construct terms containing the value $n+1$ using terms with values n, it may never be the case that the body of a rule once found true is invalidated by the derivation of a fact at a later point in time. Figure 1.4 visualizes the resulting stratification.

Fig. 1.4 Local stratification for Listing 1.21



With the concept of *local stratification* we distinguish the class of *locally stratifiable Xcerpt programs*, which is a true superset of the class of stratifiable Xcerpt programs, and thereby introduce a more general characterization of Xcerpt programs that guarantees that these programs can be evaluated by an iterative fixpoint procedure in a bottom up manner. A local stratification partitions the *Herbrand universe* of an Xcerpt program rather than the *rules* of the program into strata.

Definition 12 (Xcerpt Herbrand universe, Xcerpt Herbrand base). *The* Herbrand universe *of an Xcerpt program P are all Xcerpt data terms that can be constructed over the vocabulary of P.*[29] *Since Xcerpt programs consist only of terms without predicate symbols, the* Herbrand base *of P is defined to be the same as the Xcerpt Herbrand universe.*

Note that the above definition deviates from the Herbrand universe for logic programs as follows: While Prolog function symbols have always an associated arity, Xcerpt labels may be used to construct terms with arbitrary many children. Thus a program over the vocabulary $V = \{a\}$ has the Herbrand universe { `a{ }`, `a{ a }`, `a{ a{ a } }`, `a{ a, a }` ...}. In the following discussion of the well-founded semantics

[29] The vocabulary of P is the set of labels appearing in P.

we will, however, not consider the entire Herbrand universe for computing unfounded sets, but restrict them to the terms that occur in ground instances of the rules.

Definition 13 (Local stratification). *A* local stratification *of an Xcerpt program P is a partitioning of the Herband universe of P into strata such that the following conditions hold:*

– *All facts in P are in stratum 1.*
– *If a term q appears positively within the body of a rule R in the Herbrand instantiation of P, and c appears in its head, then q must be in the same or in a higher stratum than c.*
– *If a term q appears negatively within the body of R and c in its head, then q is in a strictly higher stratum than c.*
– *If a term q simulates into a term c, then q is in the same or in a higher stratum than q.*

The definition of local stratification of Xcerpt programs coincides with the definition of local stratification for general logic programs in the first three points. The fourth condition is necessitated by the richer unification relation induced by simulation unification in Xcerpt. While in logic programming two ground terms unify if and only if they are syntactically identical, this is not true for Xcerpt terms (consider e.g. the terms `a{{ }}`, `a[[ ]]` and `a{ b }`).

Example 1.22 underligns the necessity of the fourth condition in Definition 13: By Definition 13, Program *P* in Listing 1.22 is not locally stratifiable, but it would be, if the last condition were not part of the definition. In fact, the semantics for *P* is unclear, and it cannot be evaluated by an iterative fixpoint procedure. Figure 1.5 shows the dependency graph for Listing 1.22, which contains a cycle including a negative edge. The dependency graph for a ground Xcerpt program simply includes all rule heads and body literals as nodes, and all simulation relations between query and construct terms and negative and positive dependencies of rule heads on their body literals. The dependency graph for a non-ground Xcerpt program is the dependency graph of its Herbrand Instantiation. An Xcerpt program *P* is locally stratifiable, if its dependency graph does not contain any negative cycles (i.e. cycles including at least one negative edge).

```
CONSTRUCT a{ b } FROM not(c{{ desc b{{ }} }}) END
CONSTRUCT c[ b ] FROM a{{ }} END
```

Listing 1.22. An Xcerpt program that is not locally stratifiable

Since Listing 1.22 is not locally stratifiable, its semantics cannot be defined by a fixpoint procedure over its stratification. Similar programs – except for the simulation relation – have been studied in logic programming. For example, the logic program $\{(a \leftarrow \neg c), (c \leftarrow a)\}$ is not locally stratifiable, still the well-founded semantics of the program is given by the empty interpretation {}. To give Xcerpt programs a semantics, no matter if they are locally stratified or not, we adapt the well-founded semantics to Xcerpt programs in the Section 1.10.

Fig. 1.5 Dependency graph for Listing 1.22

$$\begin{tikzcd}
\texttt{c[b]} \arrow[r, "+"] & \texttt{a\{\{ \}\}} \arrow[d, "\leq"] \\
\texttt{c\{\{ desc b \}\}} \arrow[u, "\leq"] & \texttt{a\{ b \}} \arrow[l, "-"']
\end{tikzcd}$$

1.10 Versatile Semantics III: Negation and Versatile Queries—Well-Founded Semantics

For the sake of simplicity this section only considers Xcerpt programs without the grouping constructs `all`. Moreover queries are assumed to be either simple query terms, negations of query terms or conjunctions of positive or negated query terms. In the absence of grouping constructs or aggregate functions, a rule involving a disjunction in the rule body can be rewritten into an equivalent set of rules that are disjunction free. Also negations of conjunctions can be rewritten to conjunctions with only positive or negative query terms as conjuncts.[30]

Definition 14 (Xcerpt literal). *An Xcerpt literal is either an Xcerpt data term or the negation* `not` *d of some Xcerpt data term d. For a set S of Xcerpt literals, $pos(S)$ denotes the positive literals in S, $neg(S)$ the negative ones.*

Definition 15 (Consistent sets of Xcerpt literals). *For a set of Xcerpt literals S we denote with $\neg \cdot S$ the set of terms obtained by negating each element in S. Let p and $n =$*`not` *d be a positive and negative literal, respectively, and let S be a set of literals. p and S are consistent, iff* `not` *p is not in S. n and S are consistent iff d is not in S. S is consistent, if it is consistent with each of its elements.*

Definition 16 (Partial interpretation of an Xcerpt program (adapted from [146])). *Let P be an Xcerpt program, and $HB(P)$ its Herbrand base. A partial interpretation I is a consistent subset of $HB(P) \cup \neg \cdot HB(P)$.*

Definition 17 (Satisfaction of Xcerpt terms). *Let I be a partial interpretation for a program P. The model relationship between I and an Xcerpt term is defined as follows.*

- *Let q be a positive query term.*
  - *I satisfies q ($I \models q$) iff there is some data term $d \in pos(I)$ with $q \preceq d$*
  - *I falsifies q ($I \not\models q$) iff for all data terms $d \in HB_P$ holds $q \preceq d \Rightarrow d \in neg(I)$.*
  - *Otherwise, q is undefined in I.*
- *Let $q = not\ q'$ be a negative query term.*
  - *I satisfies q ($I \models q$) iff for all data terms d holds $q' \preceq d \Rightarrow d \in neg(I)$.*

[30] This normalization of Xcerpt rules is similar to finding the disjunctive normal form of logical formulae.

– *I falsifies q ($I \not\models q$) iff there is some data term $d \in pos(I)$ with $q' \preceq d$.*
– *Otherwise, q is undefined in I.*

Definition 18 (Satisfaction of Xcerpt queries). *Let I be a partial interpretation and q a conjunction of Xcerpt terms. I satisfies q if I satisfies each conjunct in q.*[31]

Definition 19 (Xcerpt Unfounded Sets (adapted from [146])). *Let P be an Xcerpt program, HB_P its Herbrand base, and I a partial interpretation. We say $A \subseteq HB_P$ is an* unfounded set *of P with respect to I if each atom $p \in A$ satisfies the following condition. For each instantiated rule R of P with head p and body Q at least one of the following holds:*

1. *For some positive literal $q \in Q$ holds that for all $d \in HB_P$ holds $q \preceq d \Rightarrow d \in A \vee d \in neg(I)$.*
2. *Some negative literal $q \in Q$ is satisfied in I.*

The greatest unfounded set *of P with respect to an interpretation I is the union of all unfounded sets of P with respect to I.*

Definition 20 (Well-founded semantics of an Xcerpt program). *The* well-founded semantics *of an Xcerpt program P is defined as the least fixpoint of the operator $\boldsymbol{W}_P(I) := \boldsymbol{T}_P(I) \cup \neg \cdot \boldsymbol{U}_P(I)$ where $\boldsymbol{U}_P$ and $\boldsymbol{I}_P$ are defined as follows:*

– *a postive Xcerpt literal l is in $\boldsymbol{T}_P(I)$ iff there is some ground instance R_g of some rule R in P with construct term l and query Q such that $I \models Q$.*
– *$\boldsymbol{U}_P(I)$ is the greatest unfounded set of P with respect to I.*

Consider the program P in Listing 1.23. Its Herbrand base is $HB(P) = \{$`a{ }`$\}$. Starting with the empty interpretation I_0, $\mathbf{T}_P(I_0) = \emptyset$, $\mathbf{U}_P(I_0) = \emptyset$, and $I_1 := \mathbf{W}_P(I_0) = \emptyset = I_0$. Thus the well-founded semantics of P is $\emptyset$.

```
CONSTRUCT a{ } FROM not( a{{ }} ) END
```

Listing 1.23. Simple Negation through recursion and simulation (A)

```
CONSTRUCT a{ } FROM not( a{{ }} ), not( b{ } ) END
CONSTRUCT a{ b } END
```

Listing 1.24. Simple Negation through recursion and simulation (B)

As a second example, consider program Q in Listing 1.24 with Herbrand base $HB(Q) = \{$ `a{ b }`, `a{ }`, `b{ }`$\}$. We obtain the following fix point calculation:

– $I_0 = \emptyset$
– $\mathbf{T}_Q(I_0) = \{$ `a{ b }` $\}$
– $\mathbf{U}_Q(I_0) = \{$ `a{ }`, `b{ }` $\}$
– $I_1 = \mathbf{W}_Q(I_0) = \{$ `a{ b }`, `not a{ }`, `not b{ }` $\}$
– $\mathbf{T}_Q(I_1) = \{$ `a{ b }` $\}$

[31] Xcerpt rules are assumed to be in disjunctive normal form. Therefore disjunctions need not be considered here. Satisfaction of negations is treated in Definition 17 above.

– $\mathbf{U}_Q(I_1) = \{$ `a{ },b{ }` $\}$
– $I_2 = \mathbf{W}_Q(I_1) = \{$ `a{ b },not( a{ } ),not( b{ } )` $\} = I_1$

As a final example, consider the stratified and locally stratified program R in Listing 1.25 with Herbrand universe $HB(R) = \{$ `b{ },a{ b },a{ },c{ c }` $\}$.

```
CONSTRUCT b{ } FROM not( a{{ }} ) END
CONSTRUCT a{ b } FROM not( c{{ }} ) END
CONSTRUCT a{ } FROM not( c{{ }} ) END
CONSTRUCT c{ c } END
```

Listing 1.25. Simple Negation through recursion and simulation (C)

We obtain the following fixpoint calculation:

– $I_0 = \emptyset$
– $\mathbf{T}_R(I_0) = \{$ `c{ c }` $\}$
– $\mathbf{U}_R(I_0) = \emptyset$
– $I_1 = \mathbf{W}_R(I_0) = \{$ `c{ c }` $\}$
– $\mathbf{T}_R(I_1) = \{$ `c{ c }` $\}$
– $\mathbf{U}_R(I_1) = \{$ `a{ },a{ b }` $\}$
– $I_2 = \mathbf{W}_R(I_1) = \{$ `c{ c },not( a{ } ),not( a{ b } )` $\}$
– $\mathbf{T}_R(I_2) = \{$ `c{ c },b{ }` $\}$
– $\mathbf{U}_R(I_2) = \{$ `a{ },a{ b }` $\}$
– $I_3 = \mathbf{W}_R(I_2) = \{$ `c{ c },b{ },not( a{ } ),not( a{ b } )` $\}$
– $\mathbf{T}_R(I_3) = \mathbf{T}_R(I_2)$
– $\mathbf{U}_R(I_3) = \mathbf{U}_R(I_2)$
– $\mathbf{W}_R(I_3) = \mathbf{W}_R(I_2)$

It is immediate that the well-founded semantics of R coincides with the fixpoint calculated over the stratification of R – a fact that is true for every locally stratified Xcerpt program.

Theorem 1. *For a locally stratified Xcerpt program P, the well-founded semantics of P is total and coincides with the fixpoint calculated over the local stratification of P.*

In [128] the class of weakly stratified logic programs is introduced, which is a true superset of the class of locally stratified programs and has a well-defined, two-valued intended semantics. Put briefly, to decide whether a logic program is locally stratifiable one considers the dependency graph constructed from the *entire* Herbrand instantiation of the logic program. In contrast, the decision for weak stratification is based on the absence of negative cycles within the dependency graph constructed *from a subset* of the Herbrand interpretation. This subset excludes instantiated rules containing literals of extensional predicate symbols that are not given in the program. The standard example for a program that is weakly stratified but not locally stratified is the following:

$$win(X) : -move(X,Y) \wedge \neg win(Y)$$

A position X is a winning position of a game, if there is a move from X to position Y and Y is a losing position. As mentioned above, weak stratification depends on the extension of extensional predicate symbols (*move* in the above example), and the program above is only weakly stratifiable in the case that *move* has an acyclic extension. Obviously this program can be formulated also as an Xcerpt program, and the class of locally stratified Xcerpt programs could be extended to the class of weakly stratified Xcerpt programs in a straight-forward manner. We leave the formal definition of weak stratification for Xcerpt and the question on how the richer kind of unification employed in Xcerpt affects the applicability of weak stratification for future work.

1.11 Versatile Semantics III: A Relational Perspective on Versatile Queries

Versatile queries form the central innovation of XML and RDF query languages, as illustrated in the previous sections: They allow the query author to introduce controlled forms of incompleteness or "don't cares" such as "here don't care about the order" or "here don't care about the path between two nodes as long as there is one". They are controlled in that they have to be explicitly requested by the query user and in that they have a precise logical semantics (rather than being based on approximation or ranking as in Web search engines).

The *logical semantics of versatile queries* is the focus of the following section. Rather than directly assigning meaning to versatile Web queries using simulation (Section 1.8) and investigating the affects on the semantics of rule languages build upon such queries (Section 1.9 and Section 1.10), we show how to reduce versatile queries to standard first-order logic, more precisely to Datalog with negation value invention. This is an interesting and well understood fragment of first-order logic: though computationally as expressive as full first-order logic it provides more controlled means for the creation of new terms (or "complex values") and can be easily mapped to SQL which provides similarly constrained means for value creation.

1.11.1 Contributions

Casting the semantics of versatile queries in general and Xcerpt in particular in terms of Datalog allows us to compare and contrast them with previous database languages. In particular, we use this logical semantics of Xcerpt

1. to study the complexity and expressiveness of Xcerpt and several sub-languages of Xcerpt. In particular, we show that
   a) Xcerpt expresses *all computable queries* modulo copy removal (Section 1.11.4);
   b) the same applies already to *stratified Xcerpt* (Section 1.11.4);
   c) weakly-recursive Xcerpt has the combined NEXPTIME-complete. Intuitively, a weakly-recursive Xcerpt program is an Xcerpt program that limits recursion to rules that do not increase either the nesting depth or the breadth. Thus we can rearrange the program to postpone value invention to the end of query evaluation and do not suffer complexity penalties for value invention (Section 1.11.4);
   d) non-recursive Xcerpt on tree data has data complexity in NC_1 and program complexity PSPACE-complete (Section 1.11.4).

2. to implement versatile queries on top of *relational databases* by translating them into SQL (Section 1.11.5). For such a translation to be efficient, we also need a relational representation of versatile graph-shaped data that is both space efficient and provides efficient access to graph properties such as edge traversal or reachability. Such a representation (by means of a novel labeling scheme) with linear space and time complexity for evaluating acyclic Web queries on many graphs is provided by CIQCAG, see Section 1.13.
3. to provide a *common logical foundation* for versatile queries. This allows us, as shown in Section 1.11.6, to integrate different Web query languages such as XQuery, SPARQL, and Xcerpt and to evaluate them with the same query engine. This differs notably from other approaches for the integration of Web query languages where the evaluation of the integrated languages remains separate and enables cross-language optimization and planning. Yet, thanks to the novel graph representation with CIQCAG, we can evaluate each language as efficient as the best known approaches limited to that language.

1.11.2 Preliminaries

XML and RDF Data as Relational Structures Following [16], we consider an **XML tree** as a relational structure: An XML tree is considered a relational structure T over the schema $((\mathsf{Lab}^{\lambda})_{\lambda\in\Sigma}, R_{\mathsf{child}}, R_{\mathsf{next\text{-}sibling}}, \mathsf{Root})$. The nodes of this tree are labeled using the symbols from Σ which are queried using Lab^{λ} (note, that λ is a single label not a label set). The parent-child relations are represented by R_{child}. The order between siblings is represented by $R_{\mathsf{next\text{-}sibling}}$. The root node of the tree is identified by Root. There are some additional derived relations, viz. $R_{\mathsf{descendant}}$, the transitive, $R_{\mathsf{descendant\text{-}or\text{-}self}}$ the transitive reflexive closure of R_{child}, $R_{\mathsf{following\text{-}sibling}}$, the transitive closure of $R_{\mathsf{next\text{-}sibling}}$, R_{self} relating each node to itself, and $R_{\mathsf{following}}$ the composition of $R^{-1}_{\mathsf{descendant\text{-}or\text{-}self}} \circ R_{\mathsf{following\text{-}sibling}} \circ R_{\mathsf{descendant\text{-}or\text{-}self}}$. Each node n is also related by R_{arity} to $|\{n' : R_{\mathsf{child}}(n,n')\}|$. Finally, we can compare nodes based on their label using $\cong$ which contains all pairs of nodes with same label, based on their node identity using $=$ which relates each node only to itself, and based on their structure deep equality $=_{\mathsf{deep}}$ which holds for two nodes if there exists an isomorphism between their respective sub-trees. The above ignores some XML specifics such as attributes, comments, or processing instructions but these can be added easily. For also allow an $\mathsf{all\text{-}distinct}(n_1,\ldots,n_k)$ constraint as generalisation of $=$ from two nodes to k nodes.

For example, the XML document (using subscripts to indicate node identities)

```
<a>1 <b/>2 <c>3<c/>4</c> </a>
```

is represented as $T = (\mathsf{Lab}^{\mathsf{a}} = \{1\}, \mathsf{Lab}^{\mathsf{b}} = \{2\}, \mathsf{Lab}^{\mathsf{c}} = \{3,4\}, R_{\mathsf{child}} = \{(1,2),(1,3),(3,4)\}, R_{\mathsf{next\text{-}sibling}} = \{(2,3)\}, \mathsf{Root} = \{1\})$ over the label alphabet $\{a,b,c\}$. All other relations can be derived from this definition.

In some contexts, a *graph view of XML* data is preferable as chosen in the description of Xcerpt in Section 1.4. This view does not affect the signature of the relational structure[32], but adds additional pairs of nodes to the extensions of R_{child} and $R_{\mathsf{next\text{-}sibling}}$

[32] Though we might obviously also choose to provide both views of the XML data simultaneously by additional relations instead of modified extensions of the existing ones.

and all relations derived from them. Say we want to treat ID/IDREF links like child relations resulting from element nesting in the XML document. This adds additional pairs of referencing and referenced node to R_{child}.

In the following, we choose this graph view of XML unless explicitly stated otherwise. We also allow unions of such structures, i.e., graphs consisting in multiple connected components each with its own root node (graph view of “XML forests”).

An **RDF graph** can be represented similarly as a relational structure. The main differences are the lack of order, the addition of edge labels, and the presence of node types such as literal, blank node, and resource: An RDF graph is considered a relational structure T over the schema $((\mathsf{Lab}^{\lambda})_{\lambda \in \Sigma}, \circ\!\!\rightarrow, \rightarrow\!\!\circ, \mathsf{Edge}, \mathsf{Literal}, \mathsf{Blank}, \mathsf{Named})$. As in the case of XML, Lab^{λ} provides labels from $\Sigma = \mathsf{U} \cup \mathsf{L}$, but labels both nodes and edges. A label is either an URI or a literal. Nodes are typed by the three characteristic relations Edge, $\mathsf{Literal}$, Blank, and Named into edges, literals, blank nodes, and named resources. The four sets are pairwise disjoint. Following [125], we represent labeled edges as first class elements of the domain and provide separate relation for navigating from the *source* node of an edge to that edge ($\circ\!\!\rightarrow$) and from that edge to its *sink* ($\rightarrow\!\!\circ$) node. There are some additional derived relations, viz. $R_{\mathsf{child}} = \circ\!\!\rightarrow \circ \rightarrow\!\!\circ$, $R^{\lambda}_{\mathsf{child}} = \circ\!\!\rightarrow \circ\, \mathsf{Lab}^{\lambda} \circ \rightarrow\!\!\circ$ and $R^{(\lambda)}_{\mathsf{descendant}}$ the transitive closure of $R^{(\lambda)}_{\mathsf{child}}$. Each node n is also related by R_{arity} to $|\{e' : n \circ\!\!\rightarrow e')\}|$ and each edge e to $|\{n' : e \rightarrow\!\!\circ n')\}|$. Finally, we can compare nodes and edges with the same equality relations as in the XML case.

For example, the following RDF graph (using subscripts to indicate node or edge identities)

```
@prefix ex: <http://example.org/libraries/#> .
@prefix bib: <http://www.edutella.org/bibtex#> .
ex:smith2005₁ ex:isPartOf₂ [₃ a₄ bib:Journal₅ ;
         bib:number₆ "11"₇; bib:name₈ "Computer Journal"₉ ] ;
```

is represented as $T = (\mathsf{Lab}^{\mathsf{ex:smith2005}} = \{1\}$, $\mathsf{Lab}^{\mathsf{ex:isPartOf}} = \{2\}$, $\mathsf{Lab}^{\mathsf{rdf:type}} = \{4\}$, $\mathsf{Lab}^{\mathsf{bib:Journal}} = \{5\}$, $\mathsf{Lab}^{\mathsf{bib:number}} = \{6\}$, $\mathsf{Lab}^{11} = \{7\}, \dots, \circ\!\!\rightarrow = \{(1,2),(3,4),(1,6),(1,8)\}$, $\rightarrow\!\!\circ = \{(2,3),(4,5),(6,7),(8,9)\}$, $\mathsf{Edge} = \{2,4,6,8\}$, $\mathsf{Literal} = \{7,9\}$, $\mathsf{Blank} = \{3\}$, $\mathsf{Named} = \{1,5\})$. All other relations can be derived from this definition.

Datalog with Value Invention For investigating the formal properties of languages with versatile queries and for implementing them in a relational database, we use Datalog with negation and value invention (short $\text{Datalog}^{\neg}_{new}$) as a convenient, well-studied fragment of first-order logic [2]. $\text{Datalog}^{\neg}_{new}$ extends Datalog with negation and a means for creating new values.

Rule bodies are as in standard $\text{Datalog}^{\neg}$, though we also allow disjunction in rule bodies. Rule heads are extended with conjunction and a means for *value invention*. We use a value invention term $\mathsf{new}(x_0, x_1, \dots, x_n)$, i.e., a function that maps each binding tuple for the invention variables $x_0, \dots, x_n$ to a unique new value. We will usually use some unique constant c domain for x_0 to distinguish different value invention terms. In this case, we write also $\mathsf{new}_c(x_1, \dots, x_n)$. It is easy to see that we can transform such value invention terms to the notation from [2]. In addition to the simple value invention, we also add a *deep copy* or clone facility. The deep clone term $\mathsf{deep\text{-}copy}(x_0, x_1, \dots, x_n)$

is also a function on the binding tuples of $x_0, \dots, x_n$ that returns a unique new value t, but also adds t to all unary relations that contain x_n and a pair (t, t') to each binary relation containing a pair (x_n, x') where $t' = \text{deep-copy}(x_0, x_1, \dots, x_{n-1}, x')$.[33]

For convenience, we allow *conditionals* in the head: some part of the head may depend on whether some variables are bound or not. A conditional rule has the form $h \wedge$ **if** X **then** hc_1 **else** $hc_2 \longleftarrow b$ and can be rewritten to rules without conditional constructions as follows:

```
h ∧ hc1 ⟵ b ∧ bound(X).
h ∧ hc2 ⟵ b ∧ not(bound(X)).
```

Answer variables are variables that occur in the head outside of the condition of a conditional expression.

The usual safety restrictions for Datalog$^\neg$ apply to ensure that all rules are *range-restricted*, see [2]: For each negation, all answer variables must occur also in a positive expression in the rule body. For each disjunction, all nested expressions have the same answer variables. Finally, each answer variable must also occur in the body.

Adapting the notation of [88], we call an *invention atom* an atom containing new terms. The relation name of such an atom is called an *invention relation name*. A rule is a *non-invention rule*, if it contains no invention atom in the head, otherwise it is an *invention rule*.

1.11.3 Logical Semantics for Xcerpt

In Section 1.8, we give a semantics for versatile Xcerpt queries by using the notion of simulation. Though simulation provides us with an intuitive, concise notion for the semantics of Xcerpt queries, it is a non-standard notion specifically designed for Xcerpt. Here, we choose a different approach: a semantics based on Datalog$^\neg_{new}$, a well-understood and extensively investigated fragment of standard first-order logic.

To keep the presentation focus on the salient points of the translation, we will only consider a slightly simplified version of Xcerpt queries (a logical semantics for full Xcerpt is given in [70, 68]). Specifically, we omit optional as well as sub-term negation (without) from query terms as they can be rewritten the queries with top-level negation (not), though potentially at exponential cost. We also omit construction of ordered terms, position, and label variables. Regular expressions to limited to $\star$ as label wildcard (matching nodes with any label). For simplicity, we assume in the following that term identifiers and variables are disjoint.

Query Terms To gently introduce the translation for Xcerpt, we start again with a few examples. In this section, we consider only query terms. Recall that Xcerpt query terms serve to select data from the input graph and to provide bindings for any contained variables. Intuitively, a query term can be seen like a pattern or example for the data to be selected. For details on Xcerpt query terms see Section 1.4. The translation of basic query terms is fairly straight-forward:

```
conference{{ desc paper{{ author{{}} }} }}
```

[33] For cyclic graphs, each node is cloned only once using standard memoization.

is translated to

$\mathsf{Root}(v_1) \wedge \mathsf{Lab}^{\mathsf{conference}}(v_1) \wedge R_{\mathsf{descendant}}(v_1, v_2) \wedge \mathsf{Lab}^{\mathsf{paper}}(v_2) \wedge R_{\mathsf{child}}(v_2, v_3) \wedge \mathsf{Lab}^{\mathsf{author}}(v_3)$.

We ask for root nodes (bound to v_1) with label conference and their descendants (bound to v_2) with label paper. For these descendants we are also interested in authors.

The previous example contains only partial query terms with a single sub-term each. *Total* query terms are translated very similarly but with an additional constraint on the *arity* of the respective node. For instance, `paper{ author{{ }} }` (where paper is total rather than partial as above) is translated to

$\mathsf{Lab}^{\mathsf{paper}}(v_2) \wedge R_{\mathsf{arity}}(v_2, 1) \wedge R_{\mathsf{child}}(v_2, v_3) \wedge \mathsf{Lab}^{\mathsf{author}}(v_3)$.

If we consider terms with more than one sub-term, we have to distinguish *ordered* and *unordered* terms. In an unordered term such as `paper{{ author{{ }}, title{{ }} }}` multiple sub-terms lead to node inequality constraints:

$\mathsf{Lab}^{\mathsf{paper}}(v_2) \wedge R_{\mathsf{child}}(v_2, v_3) \wedge \mathsf{Lab}^{\mathsf{author}}(v_3) \wedge R_{\mathsf{child}}(v_2, v_4) \wedge \mathsf{Lab}^{\mathsf{title}}(v_4) \wedge v_3 \neq v_4$.

In an ordered term such as `paper[[ author{{ }}, title{{ }} ]]` multiple sub-terms lead to order constraints:

$\mathsf{Lab}^{\mathsf{paper}}(v_2) \wedge R_{\mathsf{child}}(v_2, v_3) \wedge \mathsf{Lab}^{\mathsf{author}}(v_3) \wedge R_{\mathsf{following\text{-}sibling}}(v_3, v_4) \wedge \mathsf{Lab}^{\mathsf{title}}(v_4)$.

Finally, Xcerpt allows multiple occurrences of query variables and requires that all occurrences are structurally (or deep) equal. For instance, the following Xcerpt term

```
conference{{ desc paper{{ var X →author }}, var X }}
```

is translated to

$\mathsf{Root}(v_0) \wedge \mathsf{Lab}^{\mathsf{conference}}(v_0) \wedge R_{\mathsf{child}}(v_0,v_1) \wedge R_{\mathsf{descendant\text{-}or\text{-}self}}(v_1, v_2) \wedge \mathsf{Lab}^{\mathsf{paper}}(v_2) \wedge R_{\mathsf{child}}(v_2, v_3) \wedge \mathsf{Lab}^{\mathsf{author}}(v_3) \wedge R_{\mathsf{child}}(v_1, v_4) \wedge v_1 \neq v_4 \wedge v_3 =_{\mathsf{deep}} v_4$.

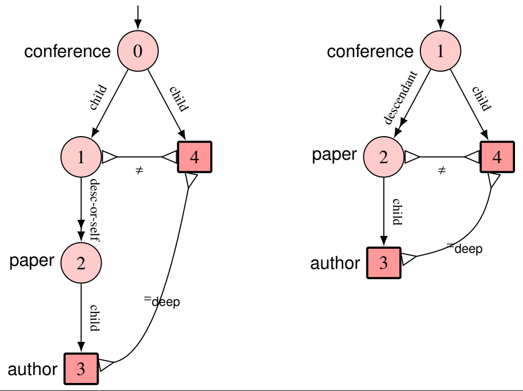
Notice also, how we split the $R_{\mathsf{descendant}}$ relation used above into R_{child} and $R_{\mathsf{descendant\text{-}or\text{-}self}}$ relations to allow for the inequality constraint amidst the children of conference. Though in this case, we can observe that bindings of v_1 and v_4 can never be the same, as bindings for v_1 must be labeled paper and bindings of v_4 (since it is deep equal to v_3) author. Therefore, we can simplify to

$\mathsf{Root}(v_1) \wedge \mathsf{Lab}^{\mathsf{conference}}(v_1) \wedge R_{\mathsf{descendant}}(v_1, v_2) \wedge \mathsf{Lab}^{\mathsf{paper}}(v_2) \wedge R_{\mathsf{child}}(v_2, v_3) \wedge \mathsf{Lab}^{\mathsf{author}}(v_3) \wedge R_{\mathsf{child}}(v_1, v_4) \wedge v_3 =_{\mathsf{deep}} v_4$.

To provide an easier to grasp manner in which denote more complex $\mathit{Datalog}^{\neg}_{\mathit{new}}$ expressions we introduce a graphical notation for queries (that also needed later to define structural properties queries). The two last $\text{Datalog}^{\neg}_{\mathit{new}}$ expressions are shown in Figure 1.6.

This representation of queries as graphs is used throughout this section and Section 1.13: Query variables are represented as nodes with labels. Root constraints are denoted by an incoming arrow. Two nodes are connected if there is an atom involving

Fig. 1.6 Translation of Xcerpt variables with $=_{\mathsf{deep}}$



the two variables. The edge is labeled with the respective relation name. Answer variables are marked by darker rectangles whereas normal variables are indicated by lighter circles.

Formally, we define the translation of Xcerpt query terms to $\text{Datalog}^{\neg}_{new}$ expressions by means of the tq_{term} function shown in Table 1.7. Xcerpt contains two context-sensitive features: multiple occurrences of Xcerpt variables as well as references (defined using `@` and referenced using `^`). Occurrences of Xcerpt variables and references are managed in a *environment* $\mathcal{E}$ that contains always the Datalog variable associated with the last occurrence of an Xcerpt variable or reference (if there is any). With $\mathcal{E}[X \leftarrow v]$ we denote the assignment (possibly overwriting existing values) of X to v in $\mathcal{E}$. Otherwise, the translation function is defined by structural recursion over the Xcerpt query term grammar. It returns for each Xcerpt term the $\text{Datalog}^{\neg}_{new}$ expression corresponding to the given term as well as the modified environment and the top-level $\text{Datalog}^{\neg}_{new}$ variable. The top-level variable is needed to express the different semantics of ordered versus unordered term lists.

The translation function tq_{term} is given in three parts, the first showing the translation of terms with sub-term specification, the second the translation of variables and references, and the third the remaining base cases. A term with sub-term specification is translated by assigning a new Datalog variable v', adding atoms for any label restriction, and translating all its sub-terms. The top-level variables returned by the translation of its sub-terms are collected and associated with v': If it is an ordered term, the top-level variable of the first child is connected to v' with R_{child}, the remaining chained with successive $R_{\mathsf{following\text{-}sibling}}$ relations (which imply that they are also children of v'). If it is an unordered term, all top-level variables are connected to v' using R_{child} and an all-distinct constraint between all top-level variables is added.

Variables and references are translated roughly in the same way: If the environment already contains a Datalog variable for the Xcerpt variable or reference, an equality constraint between the two variables is added. The environment is updated in any case (thus only linear many equality constraints are created). Variables and references differ in the choice of the equality: Variables result in a structural or deep equality constraints

query term	**Datalog$^{\neg}_{new}$ expression**
$\mathsf{tq}_{term}(\mathcal{E})\langle \lambda\{\{t_1,\ldots,t_n\}\}\rangle$	$= (\mathcal{E}_n, v', \mathsf{Lab}^{\lambda}(v') \wedge F_1 \wedge \ldots \wedge F_n \wedge R_{\mathsf{child}}(v',v_1) \wedge \ldots \wedge R_{\mathsf{child}}(v',v_n) \wedge \mathsf{all\text{-}distinct}(v_1,\ldots,v_n))$
	where v' new variable
	$(\mathcal{E}, v, F) = \mathsf{tq}_{term}(\mathcal{E}, v')\langle t_1\rangle$
	$\vdots$
	$(\mathcal{E}_n, v_n, F_n) = \mathsf{tq}_{term}(\mathcal{E}_{n-1}, v')\langle t_n\rangle$
$\mathsf{tq}_{term}(\mathcal{E})\langle \lambda\{t_1,\ldots,t_n\}\rangle$	$= (\mathcal{E}', v', F' \wedge R_{\mathsf{arity}}(v', n))$
	where $(env', v', F') = \mathsf{tq}_{term}(\mathcal{E}, v)\langle \lambda\{\{t_1,\ldots,t_n\}\}\rangle$
$\mathsf{tq}_{term}(\mathcal{E})\langle \lambda[[t_1,\ldots,t_n]]\rangle$	$= (\mathcal{E}_n, v', \mathsf{Lab}^{\lambda}(v') \wedge F_1 \wedge \ldots \wedge F_n \wedge R_{\mathsf{child}}(v',v_1) \wedge R_{\mathsf{following\text{-}sibling}}(v_1,v_2) \wedge \ldots \wedge R_{\mathsf{following\text{-}sibling}}(v_{n-1},v_n))$
	where v' new variable
	$(\mathcal{E}_1, v_1, F_1) = \mathsf{tq}_{term}(\mathcal{E}, v')\langle t_1\rangle$
	$\vdots$
	$(\mathcal{E}_n, v_n, F_n) = \mathsf{tq}_{term}(\mathcal{E}_{n-1}, v')\langle t_n\rangle$
$\mathsf{tq}_{term}(\mathcal{E})\langle \lambda[t_1,\ldots,t_n]\rangle$	$= (\mathcal{E}', v', F' \wedge R_{\mathsf{arity}}(v', n))$
	where $(env', v', F') = \mathsf{tq}_{term}(\mathcal{E}, v)\langle \lambda[[t_1,\ldots,t_n]]\rangle$
$\mathsf{tq}_{term}(\mathcal{E})\langle \mathtt{var}\ X \to t\rangle$	$= (\mathcal{E}'[\mathtt{X} \leftarrow v'], v', Q \wedge \begin{cases} (v' =_{\mathsf{deep}} v'') & \text{if } (\mathtt{X}, v'') \in \mathcal{E} \\ \top & \text{else} \end{cases})$
	where $(\mathcal{E}', v', Q) = \mathsf{tq}_{term}(\mathcal{E}, v)\langle t\rangle$
$\mathsf{tq}_{term}(\mathcal{E})\langle \mathtt{var}\ X\rangle$	$= (\mathcal{E}'[\mathtt{X} \leftarrow v'], v', \begin{cases} (v' =_{\mathsf{deep}} v'') & \text{if } (\mathtt{X}, v'') \in \mathcal{E} \\ \top & \text{else} \end{cases})$
	where v' is a new variable
$\mathsf{tq}_{term}(\mathcal{E})\langle tid\ \mathtt{@}\ t\rangle$	$= (\mathcal{E}'[tid \leftarrow v'], v', Q \wedge \begin{cases} (v' = v'') & \text{if } (tid, v'') \in \mathcal{E} \\ \top & \text{else} \end{cases})$
	where $(\mathcal{E}', v', Q) = \mathsf{tq}_{term}(\mathcal{E}, v)\langle t\rangle$
$\mathsf{tq}_{term}(\mathcal{E})\langle \hat{}\,tid\rangle$	$= (\mathcal{E}'[tid \leftarrow v'], v', \begin{cases} (v' = v'') & \text{if } (tid, v'') \in \mathcal{E} \\ \top & \text{else} \end{cases})$
	where v' is a new variable
$\mathsf{tq}_{term}(\mathcal{E})\langle \mathtt{desc}\, t\rangle$	$= (\mathcal{E}', v_1, R_{\mathsf{desc\text{-}or\text{-}self}}(v_1, v_2) \wedge Q)$
	where v_1 is a new variable
	$(\mathcal{E}', v_2, Q) = \mathsf{tq}_{term}(\mathcal{E}, v_1)\langle t\rangle$
$\mathsf{tq}_{term}(\mathcal{E})\langle \mathtt{"}string\mathtt{"}\rangle$	$= (\mathcal{E}, v', \mathsf{Lab}^{string}(v') \wedge R_{\mathsf{arity}}(v', 0))$
	where v' is a new variable

Table 1.7. Translating Xcerpt query terms

($=_{deep}$), references in node equality constraints ($=$). If we also add label variables (that are omitted here for conciseness), also label equality constraints ($\cong$) are generated, see [68].

To keep the translation concise, the resulting $\text{Datalog}^{\neg}_{new}$ expressions are not always minimal. For instance, we add an atom R_{child} followed by a $R_{\mathsf{desc\text{-}or\text{-}self}}$ atom even when there is only a single sub-term (prefixed with desc). However, it is easy to remove these redundancies, in particular to remove all occurrences of $\top$, Lab^{*}, and to compact relations where possible.

Construct Terms Construct terms serve in Xcerpt to reassemble new graphs given variable bindings obtained in related query terms. As above, we start with a few examples illustrating the translation of Xcerpt construct terms. The following assumes that we have obtained an environment $\mathcal{E}$ from the associated query term containing the mappings $(\mathtt{X}, v_1)$ and $(\mathtt{Y}, v_2)$, i.e., the representative $\text{Datalog}^{\neg}_{new}$ variable for the Xcerpt variable X (Y) is v_1 (v_2). We also abbreviate $\mathsf{new}_{\mathsf{i}}(v_1, \ldots, v_n)$ with $i(v_1, \ldots, v_n)$.

Again translating basic construct terms is fairly straight-forward:

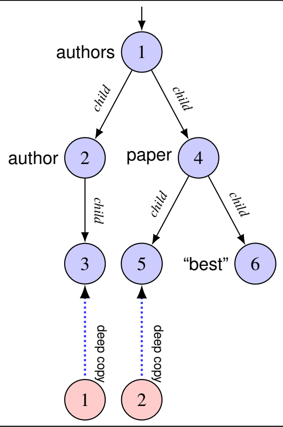
```
  authors{ author{ var X }, paper{ var Y, "best" } }
```

is translated to the following (conjunctive) $\text{Datalog}^{\neg}_{new}$ rule head:

$\mathsf{Root}(1()) \wedge \mathsf{Lab}^{\mathsf{authors}}(1()) \wedge R_{\mathsf{child}}(1(), 2()) \wedge \mathsf{Lab}^{\mathsf{author}}(1()) \wedge R_{\mathsf{child}}(2(), \mathsf{deep\text{-}copy}_3(v_1)) \wedge R_{\mathsf{child}}(1(), 4()) \wedge \mathsf{Lab}^{\mathsf{paper}}(4()) \wedge R_{\mathsf{child}}(4(), \mathsf{deep\text{-}copy}_5(v_2)) \wedge R_{\mathsf{child}}(4(), 6()) \wedge \mathsf{Lab}^{\mathsf{best}}(6())$

Graphically we denote heads of $\text{Datalog}^{\neg}_{new}$ rules similarly as their bodies (but in blue hues rather than red ones). Use of query variables for copying and grouping is indicated by dotted resp. dashed arrows. Figure 1.7 shows the graphical representation of the above rule head. The rule head specifies that there is a new value to be added to the

Fig. 1.7 Translation of Xcerpt construct term



Root relation. That same value (1()) is labeled authors and stands in R_{child} relation to two other new values. One of those is labeled author and contains a single child, the deep copy of the query variable v_1. The other is labeled paper and contains two children, one the deep copy of the query variable v_2, the other a new value labeled best.

In the translation, we only give the immediate binary relations R_{child} (and $R_{\text{following-sibling}}$ if considering ordered construct terms). Derived relations can be either automatically added to each rule head or be derived by additional rules, see [68].

Beyond what is shown in the first example, the main additional feature of construct terms is the possible presence of grouping expressed using all. The following is a simple example of such a construct term, where *all* bindings of X are listed (rather than only one as above), each wrapped in an author element which are all inside the same authors element:

```
authors{ all author{ var X } group-by(var X) }
```

This is translated very similarly as above, but now value invention (and deep copy) terms depend on query variables. More specifically, each value invention term depends on the grouping variables in whose scope the corresponding construct term occurs:

$$\text{Root}(1()) \wedge \text{Lab}^{\text{authors}}(1()) \wedge R_{\text{child}}(1(),\ 2(v_1)) \wedge \text{Lab}^{\text{author}}(1(v_1)) \wedge R_{\text{child}}(2(v_1),\ \text{deep-copy}_3(v_1,\ v_1))$$

Obviously, with nested groupings this becomes more involved as in the following, final example: Here we create one pair of author and paper elements for each unique binding of X. Within paper we group all bindings of Y for the current binding of X:

```
  authors{ all( author{ var X }, paper{
                    all var Y group-by (Y), "best" }
          ) group-by (X) }
```

The translation makes the dependence of the terms inside the second grouping on Y *and* X explicit:

$$\text{Root}(1()) \wedge \text{Lab}^{\text{authors}}(1()) \wedge R_{\text{child}}(1(),\ 2(v_1)) \wedge \text{Lab}^{\text{author}}(1(v_1)) \wedge R_{\text{child}}(2(v_1),\ \text{deep-copy}_3(v_1,\ v_1)) \wedge R_{\text{child}}(1(),\ 4(v_1)) \wedge \text{Lab}^{\text{paper}}(4(v_1)) \wedge R_{\text{child}}(4(v_1),\ \text{deep-copy}_5(v_1,\ v_2,\ v_2)) \wedge R_{\text{child}}(4(v_1),\ 6(v_1)) \wedge \text{Lab}^{\text{best}}(6(v_1))$$

Formally, we define the translation from Xcerpt construct terms to $\text{Datalog}^{\neg}_{new}$ by means of the function tc_{term} shown in Table 1.8. As for query terms, we use an environment $\mathcal{E}$ to store associations between Xcerpt query variables or references and Datalog variables. Additionally, $\mathcal{E}$ also holds the current sequence of grouping variables, which is initially empty. tc_{term} returns, similar to tq_{term}, the updated environment, the top-level construct variable, and the $\text{Datalog}^{\neg}_{new}$ (conjunctive) head formula. Again, the definition is divided in three part. The first case describes the semantics of unordered terms (ordered terms are omitted here) and empty terms. The second part that of variables and references and the final third part that of grouping terms. Grouping terms are responsible for modifying the initially empty sequence of iteration variables $\mathcal{E}$.iter: For its sub-terms the input $\mathcal{E}$.iter is extended by its grouping variables $X_1,\ldots,X_n$. Thus value invention (and deep copy) terms inside that grouping term depend also on $X_1,\ldots,X_n$.

Grouping in Xcerpt is always modulo structural or deep equivalence, i.e., all node invention and deep copy terms produce a new value only for each equivalence class

construct term	$\text{Datalog}^{\neg}_{\textit{new}}$ expression
$\mathsf{tc}_{term}(\mathcal{E})\langle tid\,\texttt{@}\,\lambda\{t_1,\ldots,t_n\}\rangle$	$= (\mathcal{E}_n, v, \mathsf{Lab}^{\lambda}(v) \wedge R_{\mathsf{child}}(v,v_1) \wedge C_1 \wedge \ldots \wedge R_{\mathsf{child}}(v,v_n) \wedge C_n)$ **where** $v = \begin{cases} v' & \text{if } (tid,v') \in \mathcal{E} \\ \mathsf{id}(\mathcal{E}.\mathsf{iter}) & \text{with } \mathsf{id} \text{ new identifier} \end{cases}$ $(\mathcal{E}_i, C_i, n_i) = \mathsf{tc}_{term}(\mathcal{E}_{i-1})\langle t_i\rangle$ with $\mathcal{E}_0 = \mathcal{E}[tid \leftarrow v]$
$\mathsf{tc}_{term}(\mathcal{E})\langle\texttt{"}string\texttt{"}\rangle$	$= (\mathcal{E}, v, \mathsf{Lab}^{string}(v))$ **where** $v = \mathsf{id}(\mathcal{E}.\mathsf{iter})$ with id new identifier
$\mathsf{tc}_{term}(\mathcal{E})\langle\texttt{var X}\rangle$	$= (\mathcal{E}, \mathsf{deep\text{-}copy}(\mathcal{E}.\mathsf{iter}, \mathcal{E}(X)), \top)$
$\mathsf{tc}_{term}(\mathcal{E})\langle\texttt{^}tid\rangle$	$= (\mathcal{E}[tid \leftarrow v], \begin{cases} v' & \text{if } (tid,v') \in \mathcal{E} \\ \mathsf{id}(\mathcal{E}.\mathsf{iter}) & \text{with } \mathsf{id} \text{ new identifier} \end{cases}, \top)$
$\mathsf{tc}_{term}(\mathcal{E})\langle\texttt{all}\ t$ $\texttt{group-by}\,(X_1,\ldots,X_n)\,\rangle$	$= \mathsf{tc}_{term}(\mathcal{E}')\langle t\rangle$ **where** $\mathcal{E}' = \mathcal{E}$ with $\mathcal{E}'.\mathsf{iter} = \mathcal{E}.\mathsf{iter} \circ [\mathcal{E}(X_1),\ldots,\mathcal{E}(X_n)]$

Table 1.8. Translating Xcerpt construct terms

modulo deep equal over the binding tuples. In other words, if there are two binding tuples where the bindings for all grouping variables are deep equal, we only produce a single new value.

Xcerpt	$\text{Datalog}^{\neg}_{\textit{new}}$ expression
$\mathsf{tr}_{\mathsf{Xcerpt}}\langle\texttt{CONSTRUCT}\ head$ $\texttt{FROM}\ body\ \texttt{END}\rangle$	$= C \longleftarrow Q$ **where** $(\mathcal{E},Q) = \mathsf{tq}(\emptyset)\langle query\rangle$ $C = \mathsf{tc}(\mathcal{E})\langle body\rangle$
$\mathsf{tc}(\mathcal{E})\langle cterm\rangle$	$= \texttt{root}(v) \wedge C$ **where** $(\mathcal{E}',v,C) = \mathsf{tc}_{term}(\mathcal{E})\langle cterm\rangle$
$\mathsf{tq}(\mathcal{E})\langle\texttt{and}\,(t_1,t_2)\,\rangle$	$= (\mathcal{E}_2,(Q_1 \wedge Q_2))$ **where** $(\mathcal{E}_1,Q) = \mathsf{tq}(\mathcal{E})\langle t_1\rangle$, $(\mathcal{E}_2,Q) = \mathsf{tq}(\mathcal{E}_1)\langle t_2\rangle$
$\mathsf{tq}(\mathcal{E})\langle\texttt{or}\,(t_1,t_2)\,\rangle$	$= (\mathcal{E}',((Q_1 \wedge v_X = v_1) \vee (Q_2 \wedge v_X = v_2)))$ **where** $(\mathcal{E}_1,Q) = \mathsf{tq}(\mathcal{E})\langle t_1\rangle$, $(\mathcal{E}_2,Q) = \mathsf{tq}(\mathcal{E})\langle t_2\rangle$ $\mathcal{E}' = \mathcal{E}_2[X \leftarrow v_X]$ for all X with $(X,v_1) \in \mathcal{E}_1, (X,v_2) \in \mathcal{E}_2$
$\mathsf{tq}(\mathcal{E})\langle\texttt{not}\,(t)\,\rangle$	$= (\mathcal{E}',\neg(Q))$ **where** $(\mathcal{E}',Q) = \mathsf{tq}(\mathcal{E})\langle t\rangle$
$\mathsf{tq}(\mathcal{E})\langle qterm\rangle$	$= (\mathcal{E}', \texttt{root}(r) \wedge Q)$ **where** $(\mathcal{E}',r,Q) = \mathsf{tq}_{term}(\mathcal{E})\langle qterm\rangle$

Table 1.9. Translating Xcerpt rules

Queries and Rules Based on the logical semantics for construct and query terms established in the previous sections, we can finally conclude the semantics by considering full Xcerpt rules. Rules are translated using $\mathsf{tr}_{\mathsf{Xcerpt}}$ as shown in Table 1.9. It delegates the translation of rule bodies and heads to different functions which each create root atoms where necessary.

An Xcerpt rule body is translated using tq which takes care of all top-level disjunction, conjunction, or negations. Note, that for conjunctions we propagate the environment returned by the translation of the first operand to the translation of the second operand, thus ensuring that matches are deep equal (as within the same query term). For disjuncts, however, we do not propagate variable mappings, but rename answer variables (i.e., variables that occur in both disjuncts) to gather bindings from both disjuncts into one variable (v_X for answer variables X). This assumes that, as usual, that non-answer variables are standardized apart for each disjunct.

The following proposition is an immediate consequence of the above construction: Variables can occur negatively only in parts of the query resulting from a negated query term where the same safety restrictions apply as for Datalog$^{\neg}_{new}$.[34]

Proposition 1. *Let R be a range-restricted Xcerpt rule. Then $\mathsf{tr}_{\mathsf{Xcerpt}}(R)$ is a safe Datalog$^{\neg}_{new}$ rule.*

It is easy to verify that in each step of the above translations the resulting Datalog$^{\neg}_{new}$ expression is linear in the input query term. Furthermore, each case treats one or more input constructs. Therefore we can surmise:

Theorem 2. *The size of the Datalog$^{\neg}_{new}$ expression Q returned by $\mathsf{tr}_{\mathsf{Xcerpt}}$ for a given Xcerpt rule P is linear in the size of P.*

To complete the semantics we also need to consider Xcerpt goals. Goals are treated the same as normal rules, but root nodes of goals are constructed in the relation answer-root rather than in Root. This also prevents the result of goals to partake in the rule chaining (observe that rule bodies only match data starting with a node in Root).

Definition 21 (Logical Semantics of Xcerpt). *Let $P = \{R_1, \ldots, R_n\}$ be an Xcerpt program. Then $P_d = \mathsf{tr}_{\mathsf{Xcerpt}}(R_1) \cup \ldots \cup \mathsf{tr}_{\mathsf{Xcerpt}}(R_n)$ is a safe Datalog$^{\neg}_{new}$ program. The logical semantics of P is the relational structure obtained by removing the Root relation and all references to nodes not reachable from a node in answer-root from the semantics of P_d (as defined in [2]).*

Example of the Full Translation To conclude the discussion of the logical semantics for Xcerpt, we give a final example of the semantics. The following Xcerpt goal selects papers containing “Cicero” as author and “puts them in a shelf”.

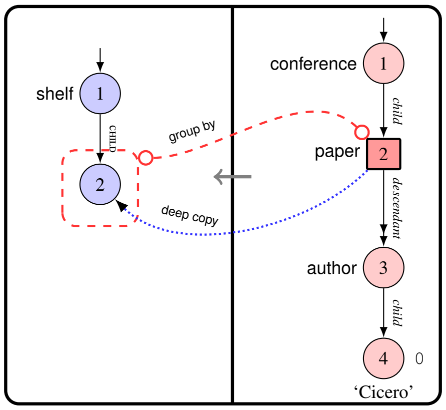
```
GOAL
  shelf{ all var X group-by(var X) }
FROM
  conference{{
    var X →paper{{
      desc author{{ "Cicero" }} }} }}
END
```

Applying $\mathsf{tr}_{\mathsf{Xcerpt}}$ to that rule yields the following Datalog$^{\neg}_{new}$ program, also depicted in Figure 1.8:

$$\mathsf{Root}(1()) \wedge \mathsf{Lab}^{\mathsf{shelf}}(1()) \wedge R_{\mathsf{child}}(1(),\ \mathsf{deep\text{-}copy}(v_2,\ v_2))$$
$$\longleftarrow \mathsf{Root}(v_1) \wedge \mathsf{Lab}^{\mathsf{conference}}(v_1) \wedge R_{\mathsf{child}}(v_1,\ v_2) \wedge \mathsf{Lab}^{\mathsf{paper}}(v_2) \wedge R_{\mathsf{descendant}}(v_2,$$
$$v_3) \wedge \mathsf{Lab}^{\mathsf{author}}(v_3) \wedge R_{\mathsf{child}}(v_3,\ v_4),\ \mathsf{Lab}^{\mathsf{Cicero}}(v_4) \wedge R_{\mathsf{arity}}(v_4,\ 0).$$

[34] We use inequalities outside of the translation of negated query terms, but only in a safe manner, see Table 1.7.

Fig. 1.8 Example of rule translation



The query variable v_2 is used in the head to specify which part of the data to copy and how often. Recall that deep-copy(v_2, v_2) indicates that, for each unique binding of v_2, a new node should be created that is a deep copy of v_2 itself.

Outlook: Xcerpt$^{\text{RDF}}$ The above treatment of Xcerpt is focused on Xcerpt$^{\text{XML}}$. Though extending the translation to Xcerpt$^{\text{RDF}}$ is not difficult, it requires a number of adjustments that are briefly summarized in the following.

– Most importantly, the above translation considers only XML data. If we also want to query *RDF data* we have to extend the translation rules to the specifics of that data model. Section 1.11.2 outlines how to represent RDF data in a relational structure that can be queried using Datalog$^{\neg}_{new}$.
– RDF and Xcerpt$^{\text{RDF}}$ distinguish different node types such as blank nodes, named resources, and literals and contain named edges. All these features require slight adaptations to the translation. To give a flavor of these adaptations consider the following Xcerpt$^{\text{RDF}}$ query term:

```
var X {{ foaf:knows → _:1{{ foaf:name → "Julius Caesar"}}
    }}
```

It queries for persons that know someone who is named "Julius Caesar". Its translation uses ○→ and →○ for named edge traversal (rather than R_{child} as for unnamed edge traversal in XML) and requires each Datalog$^{\neg}_{new}$ to be a specific kind of RDF node or edge.

$$\text{Named}(n_1) \wedge \text{Lab}^{\text{ex:anna}}(n_1) \wedge {\circ\!\!\rightarrow}(n_1, e_1) \wedge \text{Edge}(e_1) \wedge \text{Lab}^{\text{foaf:knows}}(e_1) \wedge {\rightarrow\!\!\circ}(e_1, n_2) \wedge \text{Blank}(n_2) \wedge {\circ\!\!\rightarrow}(n_2, e_2) \wedge \text{Edge}(e_2) \wedge \text{Lab}^{\text{foaf:name}}(e_2) \wedge {\rightarrow\!\!\circ}(e_2, n_3) \wedge \text{Literal}(n_3) \wedge \text{Lab}^{\text{Julius Caesar}}(n_3)$$

– Xcerpt$^{\text{RDF}}$ contains not just term (there called "graph") variables as discussed in the previous sections, but a number of additional variable kinds. Node and predicate (label) variables can easily be added to the above semantics. Essentially they are treated the same as label variables in [68]. More challenging are CBD-variables that select *concise bounded descriptions* of a matching node. A concise bounded description is similar to a term variable in that it binds to a structure (rather than just to a single label as label variables). But that structure may be only an excerpt of the actual sub-graph rooted at a matching node: It includes only all paths up to and including the first named resource on that path. CBD-variables can be added to the translation *without any changes to the target language Datalog*$_{new}^{\neg}$ as they can be expressed through recursive Datalog$_{new}^{\neg}$ rules. However, such a realisation is likely to be inefficient. Therefore adding a specific operator for these variables is preferable.

In the following, we will continue considering only Xcerpt$^{\text{XML}}$, except for Section 1.11.6 where we briefly revisit the integration of Xcerpt$^{\text{RDF}}$ and Xcerpt$^{\text{XML}}$. However, it is easy to check that all the results below transfer also to Xcerpt$^{\text{RDF}}$ as all of the added or changed features can be expressed in Datalog$_{new}^{\neg}$ over relational structures representing RDF graphs. Also the features can be translated to Datalog$_{new}^{\neg}$ expressions in linear time and space, as in the case of Xcerpt$^{\text{XML}}$.

This concludes the definition of the logical semantics of Xcerpt by translation to Datalog$_{new}^{\neg}$. The following sections exploit this semantics to prove complexity and expressiveness properties of Xcerpt and several sub-languages of Xcerpt (Section 1.11.4) and to implement Xcerpt on top of relational database (Section 1.11.5).

1.11.4 Expressiveness and Complexity of Xcerpt

From the previous section, we obtain a linear translation of Xcerpt programs to Datalog$_{new}^{\neg}$ programs. Here we show how to use that translation to adapt or extend a number of existing results on expressiveness and complexity of Datalog$_{new}^{\neg}$ to Xcerpt and some interesting sub-languages of Xcerpt.

Xcerpt: Query Complete First, let us consider full Xcerpt. The above translation establishes that we can find a Datalog$_{new}^{\neg}$ program to compute the semantics of any Xcerpt program and that this translation is linear. What about the other direction? It turns out, that we can encode each Datalog$_{new}^{\neg}$ program in an equivalent Xcerpt program of at most quadratic size:

Theorem 3. *Xcerpt has the same expressiveness, complexity, and completeness properties as Datalog*$_{new}^{\neg}$ *(and thus ILOG [88]).*

Proof. By the translation above, we can give a Datalog$_{new}^{\neg}$ program for each Xcerpt program.

On the other hand, each Datalog$_{new}^{\neg}$ program P can be encoded as an Xcerpt program P' preserving the semantics of P in the following way:

Each atom in the body is represented as an ordered, total Xcerpt query term with the predicate symbol as term label, replacing Datalog$_{new}^{\neg}$ variables by Xcerpt variables and Datalog$_{new}^{\neg}$ constants c by `"c"`. The head atom is represented as an ordered, total Xcerpt construct term, replacing Datalog$_{new}^{\neg}$ variables by Xcerpt variables and Datalog$_{new}^{\neg}$ constants c by `"c"`. An invention symbol in the head is replaced by the Xcerpt term

$\mathsf{new}[t_1,\ldots,t_n]$ where $t_1,\ldots t_n$ are the non-invention variables or constants in that head and new is a unique symbol not otherwise used in the program or data. Thus a resulting term simulates only with other instances of the same head by virtue of the unique term label new. Essentially we generate a new term for each unique binding tuple of $t_1,\ldots,t_n$ (modulo deep equal).

It is easy to see that if $P \models p(t_1,\ldots,t_n)$ then there is an isomorphism κ from Xcerpt terms with new labels to invention constants such that $p[t'_1,\ldots,t'_n]$ can be derived with the rules in P', $t'_i = t_i$ if t_i is a normal constant, and $t'_i = \kappa(t_i)$ otherwise.

This translation is at worst quadratic in the size of the Datalog$^{\neg}_{new}$ program: Invention symbols may lead to duplication of variable occurrences, but since only *non-invention* variables and constants are ever included this duplication does not lead to exponential size.

For the following corollary we exploit several results on ILOG [88], a syntactic variant of Datalog$^{\neg}_{new}$. First, we call two answers equivalent up to "copy removal" if they differ only in invented values and those invented values are deep or structurally equivalent. Second we recall the class of (list) constructive queries from [42] which are designed to capture precisely the queries expressible in languages such as Datalog$^{\neg}_{new}$, ILOG, or Xcerpt. It coincides with the class of queries where the new domain elements in the output can be viewed as hereditary finite lists constructed over the domain elements of the input. Hereditary finite lists are lists constructed over a given set U of "ur-elements" from the input domain such that each element of the list is either from U or a hereditary finite list over U. With this definitions and respective results on ILOG from [88] and [43] we obtain that

Corollary 1. *1. Xcerpt is Turing complete. 2. Xcerpt is query complete modulo copy removal, i.e., it* expresses all computable queries *modulo copy removal. 3. Xcerpt is (list) constructive complete.*

The reason for the limitation to "modulo copy removal" is that Xcerpt uses deep or structural equivalence as equivalence relation for grouping and can not distinguish between two terms that are deep equal.[35]

This result shows that while Xcerpt is indeed Turing complete that expressive power is justified as it can express all computable queries modulo deep equal. Since the whole language is Turing complete, it is worth investigating sub-languages with better computational properties. Before we turn to that question, let us briefly consider the effect of stratification on expressiveness and complexity:

Stratified Xcerpt: Still Query Complete In the translation above as well as most parts of Section 1.3 we only consider programs with a limited form of negation, viz. stratified negation.

Recall the definition of stratified Xcerpt from Section 1.9, here recast using the dependency graph on Xcerpt rules, as we make use of that notation also for defining

[35] This issue is closely related to the issue of lean vs. non-lean RDF graphs as answers in languages such as SPARQL or RDFLog [38]: That Xcerpt is complete "modulo copy removal" means that it can not create answers (or groupings) with several instances of the same, structurally equivalent graph, i.e., it can only produce the term equivalent of lean answers.

some sub-languages of Xcerpt below. Given an Xcerpt rule R we define its top-level query terms as usual: A query terms in R is called *top-level* if it occurs inside the body of R nested only inside (arbitrary combinations) of `and`, `or`, and `not`.

Definition 22 (Dependency graph for Xcerpt programs). *Let $P = R_1, \dots, R_n$ be an Xcerpt program. Then the* dependency graph $D(P) = (N, E)$ *for P is defined as follows:*

- *The nodes of $D(P)$ are the rules of P.*
- *There is an edge from R_i to R_j in $D(P)$ iff one of the top-level query terms q of R_i simulates with the construct term of R_j. The edge is negative if q occurs inside a* `not` *in R_i, otherwise it is positive.*

Using the notion of dependency graph, we can define stratified Xcerpt programs as follows:

Definition 23 (Stratified Xcerpt). *Let $P = R_1, \dots, R_n$ be an Xcerpt program. Then P is called* stratified*, if there is a partitioning of P into strata $S_1, \dots, S_k$ such that there is no negative edge from an $R \in S_i$ to an $R' \in S_j$ with $i < j$.*

Proposition 2. *Let P be a stratified Datalog$^{\neg}_{new}$ program. Then the Xcerpt encoding of P by the proof of Theorem 3 is a stratified (and therefore locally stratified) Xcerpt program.*

Proof. A stratification of P immediately gives us a stratification of its Xcerpt encoding P' as any negated query term t in P' yields from a negated atom in P and the corresponding rule can all construct terms in lower strata have top-level labels that are different from the top-level label of t and thus do not unify. Otherwise the predicate that construct term is the encoding of depends on the negated atom already in P, yet is in a lower stratum in contrast to the assumption that P is stratified.

From this result and [43] which shows that stratified Datalog$^{\neg}_{new}$ has the same expressive power as full Datalog$^{\neg}_{new}$ and thus can express all computable queries modulo copy removal we can deduce the same observation for Xcerpt:

Corollary 2. *Stratified Xcerpt already expresses all computable queries modulo copy removal.*

In other words, the class of queries expressible in Xcerpt does not shrink if we constrain ourselves to stratified programs. This contrast to the case of Datalog$^{\neg}$ without value invention where stratification is indeed a limitation on the kind of queries expressible in the language.

Weakly-Recursive Xcerpt: Finite Models A first decidable sub-language of Xcerpt is inspired by the notion of weakly acyclic Datalog$^{\neg}_{new}$ [88] that is also used extensively, e.g., in the data exchange setting. Essentially combining recursion and value invention is dangerous as we can no longer give a bound on the number of ground terms entailed by a program (in other words the active domain is no longer finite). The notion of weak acyclicity is a sufficient condition to guarantee finite active domains: We allow recursion but only if no new values can be created on by the recursive rules. In terms of the dependency graph of the program: we allow cycles in the dependency graph as long as they are not through invention atoms.

Directly applying this notion to Xcerpt is unsatisfying as every Xcerpt rule generates new nodes. However, since we generate new nodes modulo deep equality, we only need to ensure that the number of different terms a program can generate is finite.

Given two terms t and t'. We define the *nesting depth* of t in t' as usual: If $t = t'$ then the nesting depth is 0. Otherwise, if t' is nested inside a term t'' in t with nesting depth d then t' has nesting depth $d+1$. If t' occurs several times in t then its nesting depth is the minimum of the nesting depths of its occurrences.

Definition 24 (Weakly-recursive Xcerpt). *Let $P = R_1, \ldots, R_n$ be an Xcerpt program. Then P is called* weakly-recursive*, if for each edge (R_i, R_j) in $D(P)$ the following holds:*

- *The construct term c of R_j does not contain any grouping terms (no `all`).*
- *For each variable in c the nesting depth of its occurrence in c is less or equal to the nesting depth in any top-level query term q in R_i that simulates with c.*

Weakly-recursive Xcerpt is the fragment of Xcerpt containing all such programs.

Roughly speaking the absence of grouping terms prevents terms with unbounded breadth and the second condition places a bound on the breadth of terms.

Theorem 4. *Weakly-recursive Xcerpt is decidable and the combined complexity of its evaluation is* NEXPTIME*-complete.*

Proof. Weakly-recursive Xcerpt is NEXPTIME-hard as we can reduce weakly-recursive Datalog$^{\neg}_{new}$ which is known to be NEXPTIME-complete [43] to weakly-recursive Xcerpt by the construction in the proof of Theorem 3. The resulting Xcerpt programs are indeed weakly-recursive, as the construction never creates grouping terms and the nesting depth only increases when translating invention atoms.

On the other hand, weakly-recursive Xcerpt is also obviously in NEXPTIME: For a given input program we can compute the maximum depth and breadth of a term as well as the number of distinct labels for a given input term of depth d, breadth b, and number of distinct labels l.

We generate each of the $O(b^d \cdot l)$ different terms that can be generated with these bounds. Then we compute the Xcerpt program by a standard fixpoint operator, but instead of generating new terms we only mark those terms we have already derived. If there are no more rules that mark additional terms or all terms are marked, the evaluation terminates. A single derivation step is obviously in NP. Since each step marks at least one term, there are at most $O(b^d \cdot l)$ steps

Non-Recursive Xcerpt: Parallelizable Though weakly-recursive Xcerpt is decidable it is still fairly expensive to evaluate. An obvious further restriction is to allow no recursion in Xcerpt at all:

Definition 25 (Non-recursive Xcerpt). *Let $P = R_1, \ldots, R_n$ be an Xcerpt program. Then P is called* non-recursive*, if its dependency graph $D(P)$ is acyclic. Non-recursive Xcerpt is the fragment of Xcerpt containing all such programs.*

Though this restriction limits the construction of new values, it turns out that even the application of a single Xcerpt rule can be potentially expensive due to the use of deep-equal. For arbitrary Xcerpt terms this is as hard as graph isomorphism for which no polynomial time algorithms are known. Therefore, we also limit ourselves to trees-shaped data as input and disallow references in rules.

It turns out that with these two restriction, we obtain a sub-language that is efficiently parallelizable (wrt. data complexity):

Proposition 3. *Non-recursive Xcerpt on trees has data complexity in* $\text{NC}_1 \subseteq \text{L}$ *and program complexity* PSPACE*-complete.*

Proof. We can obtain a non-recursive Datalog$^\neg$ program with deep-equal by 1. computing the (now acyclic) dependency graph, 2. indexing all relations in the head of each rule with a unique identifier, 3. replacing references to the relation in the body of each rule with a disjunction referencing the indexed relations of all rules they may depend on. The resulting program is a non-recursive Datalog$^\neg$ program with deep-equal and is at most exponential in the size of the input Xcerpt program. A Datalog$^\neg$ program with deep-equal can be evaluated with data complexity in NC_1 and program complexity in PSPACE(which is thus not affected by the exponential translation size) since deep-equal on trees is NC_1-complete [90].

1.11.5 From Xcerpt to SQL: A Foundation for a Relational Implementation

With the translation to Datalog$^\neg_{new}$ for Xcerpt programs, we not only achieve a purely logical semantics, but also the foundation for a relational implementation: First notice, that each stratified Datalog$^\neg$ program can be translated into a, possibly recursive, SQL expression. SQL recursion (introduced in SQL:1999 and refined in SQL:2003) is expressed using `with` and is limited to monoton recursion: A relation P may be defined by means (including negation) of a relation Q only if adding tuples to Q cannot cause any triple of P to be deleted. Fortunately, stratified Datalog$^\neg$ programs are designed to be allow only monoton recursion.

With the addition of ranking operators in SQL:1999 controlled value generation has been standardized as well. When translating a Datalog$^\neg_{new}$ program to SQL, we employ the ROW_NUMBER or `DENSE_RANK` function to generate new node IDs based on the invention variables. For details see [81] where these are used in the context of XQuery iteration.

The chief disadvantage of translating Xcerpt (in this or other ways) to SQL for implementation is that the nave relational representation discussed in Section 1.11.2 does not perform well in practice. This has been observed frequently and, for tree data, labeling schemes such as the pre-/post-encoding [80], ORDPATH [121], or BIRD [152] have been suggested to provide better XML storage. They provide linear time and space processing of XML tree queries on tree data. However, when querying graph data these approaches do not immediately apply. Therefore, we have developed a labeling scheme for graph data that not only provides linear time and space evaluation for tree data but also for many graphs, see Section 1.13.

1.11.6 Versatile Semantics: Adding XPath, XQuery, and SPARQL

The above translation has been focused so far on Xcerpt with a brief outlook to Xcerpt$^{\text{RDF}}$. However, Datalog$^{\neg}_{new}$ together with the relation representations for RDF and XML data from Section 1.11.2 can form a common basis for analysing and evaluating a far larger set of query languages.

In fact in [70, 68], we show how to map not only Xcerpt but also XPath, XQuery, and SPARQL to Datalog$^{\neg}_{new}$. Combined with the labeling scheme and evaluation for tree and graph data discussed in Section 1.13 this allows us the use of the same, efficient evaluation engine for all this languages. In particular, we can integrate queries written in these very different languages. Though such integration has been suggested previously (e.g., in [127]), none of the previous approaches achieves language integration also on the level of the evaluation engine. By translating both languages to Datalog$^{\neg}_{new}$ we open up opportunities for cross language optimization. Furthermore, the labeling scheme propose in Section 1.12 allows for such integration without sacrificing efficiency for the more restricted languages (e.g., for XPath on tree data).

Example: Language Integration As illustration let us consider an example of such language integration where we allow XPath and SPARQL queries to occur in the body of an Xcerpt rule. XPath queries are always only filters, i.e., they do not provide variable bindings. SPARQL queries may provide variable bindings, though such variables are always *label variables* only. The same variables may be used in body parts of different languages and are understood as multiple variable occurrences in Xcerpt. However, if Xcerpt term variables are used in SPARQL or XPath only their top-level label is considered. For simplicity we assume that XPath and Xcerpt query the same XML data, but SPARQL queries a separate RDF graph. Of course, we could also access different data sets in each language.

The following example selects the names of authors of conference papers in a variable `X` if they contain “Cicero” in an Xcerpt query. An XPath filter constraints these bindings by requiring that there is also a member of the organizers from Plato’s “Akademia” with the same name. Finally, we also select all resources in the RDF data whose dc:creator has the same full-name.

```
GOAL
 shelf{ all author { var X, all var A group-by A } group-by X
     }
FROM
 and(
  conference{{
   paper{{
    desc author{{ var X →/.*/ }} }} }},

  //organizers/member[affiliation[text() = 'Akademia']
      name[text()=$X]],

  SELECT ?A
  WHERE { ?A dc:creator ?P AND ?P vcard:FN ?X }
 )
```

```
END
```

The translation to Datalog$^{\neg}_{new}$ is very much along what we have seen in Section 1.11.3:

$$\begin{aligned}
&\mathsf{Root}(1()) \wedge \mathsf{Lab}^{\mathsf{shelf}}(1()) \wedge R_{\mathsf{child}}(1(),\ 2(v_2)) \wedge \mathsf{Lab}^{\mathsf{author}}(2(v_2)) \wedge R_{\mathsf{child}}(2(v_2),\\
&\quad \mathsf{deep\text{-}copy}(v_2,\ v_2)) \wedge R_{\mathsf{child}}(2(v_2),\ \mathsf{deep\text{-}copy}(v_2,\ s_1,\ s_1))\\
&\longleftarrow \mathsf{Root}(v_1) \wedge \mathsf{Lab}^{\mathsf{conference}}(v_1) \wedge R_{\mathsf{child}}(v_1,\ v_2) \wedge \mathsf{Lab}^{\mathsf{paper}}(v_2) \wedge R_{\mathsf{descendant}}(v_2,\\
&\quad v_3) \wedge \mathsf{Lab}^{\mathsf{author}}(v_3) \wedge R_{\mathsf{child}}(v_3,\ v_4),\ R_{\mathsf{arity}}(v_4,\ 0) \wedge\\
&\\
&\quad \mathsf{Root}(x_1) \wedge R_{\mathsf{descendant\text{-}or\text{-}self}}(x_1,\ x_2) \wedge \mathsf{Lab}^{\mathsf{organizers}}(x_2) \wedge R_{\mathsf{child}}(x_2,\ x_3) \wedge\\
&\qquad \mathsf{Lab}^{\mathsf{member}}(x_3) \wedge R_{\mathsf{child}}(x_3,\ x_4) \wedge \mathsf{Lab}^{\mathsf{affiliation}}(x_4) \wedge R_{\mathsf{child}}(x_4,\ x_5) \wedge\\
&\qquad \mathsf{Lab}^{\mathsf{Akademia}}(x_5) \wedge R_{\mathsf{child}}(x_3,\ x_6) \wedge \mathsf{Lab}^{\mathsf{name}}(x_6) \wedge R_{\mathsf{child}}(x_6,\ x_7) \wedge\\
&\qquad x_7 \cong v_4 \wedge\\
&\\
&\quad \circ\!\!\rightarrow(s_1,\ e_1) \wedge \mathsf{Edge}(e_1) \wedge \mathsf{Lab}^{\mathsf{dc:creator}}(e_1) \wedge \rightarrow\!\!\circ(e_1,\ s_2) \wedge \circ\!\!\rightarrow(s_2,\ e_2) \wedge\\
&\qquad \mathsf{Edge}(e_2) \wedge \mathsf{Lab}^{\mathsf{vcard:FN}}(e_2) \wedge \rightarrow\!\!\circ(e_2,\ s_3) \wedge s_3 \cong v_4.
\end{aligned}$$

Obviously, in this case the use of XPath affords little gain compared to Xcerpt only queries, but the same technique can be applied to integrate XPath into SPARQL or SPARQL into XQuery. Full translations for SPARQL, XPath, and XQuery can be found in [68].

1.11.7 Outlook

Xcerpt and versatile Web queries in general are a powerful and convenient tool for accessing Web data. In this section, we show that, both their semantics and evaluation, can nevertheless be cast in terms of existing logical foundations and technology. In particular, we show how Xcerpt can be translated to Datalog$^{\neg}_{new}$ and use that translation to proof several formal properties of Xcerpt and interesting sub-languages thereof. Perhaps even more important is that the suggested translation can also be achieved for such diverse Web query languages as SPARQL, XPath, or XQuery. Not only does that provide us with a playground for comparing and investigating these languages, it also allows us, as discussed in the last two Sections, to integrate and implement these languages in a common engine. We have only outlined first ideas towards this integration here. There remain a plethora of open issues such as the right mapping of variable bindings. One of the most crucial of these issues is the question whether the use of such a common engine does not sacrifice performance for the more restricted languages such as XPath. The following Section 1.12 essentially answers this question negative: We can provide a common engine for these languages based on a uniform evaluation of tree and graph data, that nevertheless provides a linear time and space (and thus optimal) evaluation for XPath (tree queries on tree data). It even extends this complexity to many graphs in contrast to all previous approaches.

approach	reachability time	labeling time	labeling size
2-Hop [53]	$O(n)$	$O(n^4)$	$O(n)$
HOPI [136]	$O(n)$	$O(n^3)$	$O(n)$
Graph labeling [6]	$O(n)$	$O(n^3)$	$O(n)$
SSPI [48]	$O(n)$	$O(n^2)$	$O(n)$
Dual labeling [150]	$O(1)$	$O(n^3)$	$O(n)$
GRIPP [144]	$O(n)$	$O(n^2)$	$O(n)$

Table 1.10. Complexity of graph labeling schemes for reachability test on arbitrary graphs (labeling size is per node). $n \geq e_g$: number of non-tree edges.

1.12 Versatile Evaluation

In the previous sections, we have shown how versatile query languages advance the state-of-the-art for querying the Web, where often the same application needs access to data published in different formats.

Employing a versatile query language may be convenient, but what about the cost? If the price is that we have to forgo efficient evaluation methods that exploit the specific limitations of the involved data formats, versatile query languages may often not be practical.

Fortunately, we show in this section that in two crucial aspects this concern is not justified: First, we present a uniform evaluation algorithm that is capable of dealing with arbitrary graphs (as they occur in RDF data), but (seamlessly) processes trees and even many non-trees

1.13 Versatile Evaluation I: Structure Scaling with CIQCAG

What makes Web queries different from those used in centralized, relational databases is the emphasize on versatile, flexible structure conditions. Web queries are often written against data, where neither the exact shape of the children of a node nor of the paths connecting two nodes is known. This observation leads to emphasize on a flexible representation of Web data, be it tree- or graph-shaped, where we can not assume a fixed, recursion- and repetition-free schema.

Labeling schemes have become a popular means for providing efficient queries to Web data, in particular if that Web data is represented relationally. Labeling schemes assign each node a unique (constant[36]) label such that we can decide whether two nodes stand in a certain structural relations given only their labels. For tree data, several labeling schemes with constant time membership test have been proposed [80, 121, 152].

[36] As most other works on labeling schemes, we consider label size to be bounded in practice and thus as constant. More precisely, label size is in $O(\log n)$ where n is the number of nodes in the document.

On arbitrary graph data testing adjacency (or reachability) in both constant time *and* constant per-node space is not possible.[37] Labeling schemes have therefore focused on heuristics for finding compact representations of reachability and adjacency. These heuristics come in roughly two kinds (here and in the following n, m are the number of nodes and edges in a given graph):

– For reachability in *arbitrary graphs*, the 2-hop cover [53] is a set of shortest paths such that for any two nodes there is a concatenation of two such paths that is a shortest path for those two nodes. Such a 2-hop cover can be exploited to assign labels for reachability testing: Each node v is labeled with two sets $(L_{out}(v), L_{out}(v))$ such that $L_{out}(v) \cap L_{in}(v')$ for each node v' reachable from v. However, finding an optimal 2-hop labeling NP-hard and there are graphs whose optimal 2-hop labeling is at least $\Omega(n \cdot m^{1/2})$ in size. Further work on 2-hop labeling has focused mostly on efficient approximation algorithms [136].
– Often sparse graphs are almost tree-shaped with only few non-tree edges. This is exploited by a several approaches [6, 48, 150, 144] for extending tree labelings, mostly pre-/post-labelings [80], to graphs. However, for all of these approaches the largest interesting class of graphs where they can still guarantee constant time and per-node space reachability tests are trees. On arbitrary graphs they either degenerate in space or time complexity.
  Roughly speaking all three four approaches extend pre-/post-tree labeling to arbitrary graphs by first labeling a spanning tree. They differ in how they deal with non-tree edges: In [6] nodes get additional pre-/post-intervals for descendants not reachable by tree edges at the cost of up to linear space per node. In [48] non-tree edges are iterated at query time at the cost of up to linear time for testing reachability. In [150] the transitive closure of non tree edges is computed and stored at the cost of up to linear space per node space. Finally, [144] presents a refined combination of [6] and [48] that performs on sparse graphs often significantly better, but does not improve the worst-case space or time complexity.

Table 1.10 summarizes these time and per-node space complexity.

1.13.1 Contributions

In this chapter, we present a *novel characterization* of a class of graphs that is a proper, non-trivial superclass of trees that still exhibits a *labeling scheme with constant time, constant per-node space* adjacency and reachability tests. Furthermore, we give a quadratic algorithm that computes, for an arbitrary graph, such a labeling if one exists.

Constant time membership test almost immediately yields linear time evaluation for existential acyclic conjunctive queries on tree data. However, nave approaches for n-ary universal queries take at least quadratic time in the graph size. We show how the above labeling scheme can be exploited to give an *algorithm for evaluating acyclic conjunctive queries* that is $O(n \cdot q)$ wrt. time and space complexity, i.e., linear in both

[37] As there are 2^{n^2} different graphs, yet constant per-node space allows only for 2^n different representations.

Fig. 1.9 Sharing: On the Limits of Continuous-image Graphs

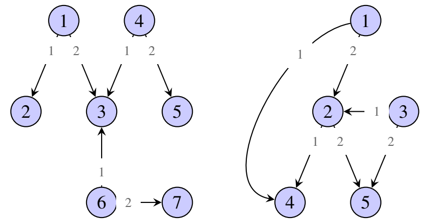
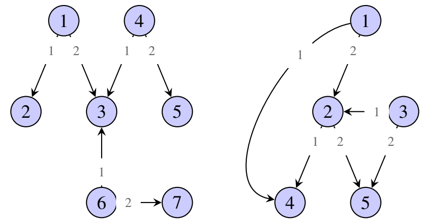


Fig. 1.10 Sharing: On the Limits of Continuous-image Graphs



data and program complexity. Furthermore, our algorithm guarantees iteration in the size of the related nodes rather than in all nodes.

1.13.2 Labeling Beyond Trees: Continuous-Image Graphs

Tree data, as argued above, allows us to represent relations on that data more compactly, e.g., using various interval-based labeling schemes. Here, we introduce a new class of graphs, called *continuous-image graphs* (or CIGS for short), that generalize features of tree data in such a way that we can evaluate (tree) queries on CIGS with the same time and space complexity as techniques such as twig joins [30] which are limited to tree data only.

Continuous-image graphs are a proper superset of (ordered) trees. On trees we require that each node has at most one parent. For continuous-image graphs, however, we only ask that we can find a single order on all nodes of the graph such that the children of each parent form a continuous interval in that order. Formally, we define a continuous-image graph by means of the image interval property (a generalization of corresponding properties of tree-shaped relations or closure relations of tree-shaped base relations. Recall that we denote with $R(n)$ for a node $n \in N$ and a binary relation R over the domain N the set $\{n' \in N : (n, n') \in R\}$.

Definition 26 (Continuous-image Graph). *Let R be a binary relation over a domain (of nodes) N. Then R is a continuous-image graph, short* CIG*, if it carries the* image interval *property: there is a total order $<_i$ on N with the induced sequence S over N such that for all nodes $n \in N$, $R(n) = \emptyset$ or $R(n) = \{S[s], \dots, S[e] : s \leq e \in \mathbb{N}\}$.*

The definition of continuous image graphs allows graphs where some or all children of two parents are "shared" (in contrast to trees where this is never allowed). However, it limits the degree of sharing: Figure 1.10 shows two minimal graphs that are *not* CIGs. Incidentally, both graphs are acyclic and, if we take away any one edge in either graph, the resulting graph becomes a CIG. The second graph is actually the smallest (w.r.t. the number of nodes and edges) graph that is not a CIG. The first is only edge minimal but illustrates an easy to grasp sufficient but not necessary condition for violating the image interval property: 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.

On continuous-image graphs we can exploit similar techniques for representing structural relations as on trees, most notably we can label each node with a single, continuous interval for its children and/or descendants. Together with a simple index to represent that nodes position in the underlying order, we obtain constant space labels (three integers), yet can test adjacency and/or reachability in constant time (with two integer comparisons).

Testing for CIGs: consecutive ones property Moreover, whether a given graph is a CIG (and in what order its node must be sorted to arrive at continuous intervals for each parent's children) is just another way of saying that its adjacency matrix carries the *consecutive ones* property [67]. For the consecutive-ones problem [24] gives the first linear time (in the size of the matrix) algorithm based on so called PQ-trees, a compact representation for permutations of rows in a matrix. More recent refinements in [83] and [87] show that simpler algorithms, e.g., based on the PC-tree [86], can be achieved. We *adapt* these algorithms to obtain a linear time (in the size of the adjacency matrix) algorithm for deciding whether a graph is a CIG and computing a CIG-order.

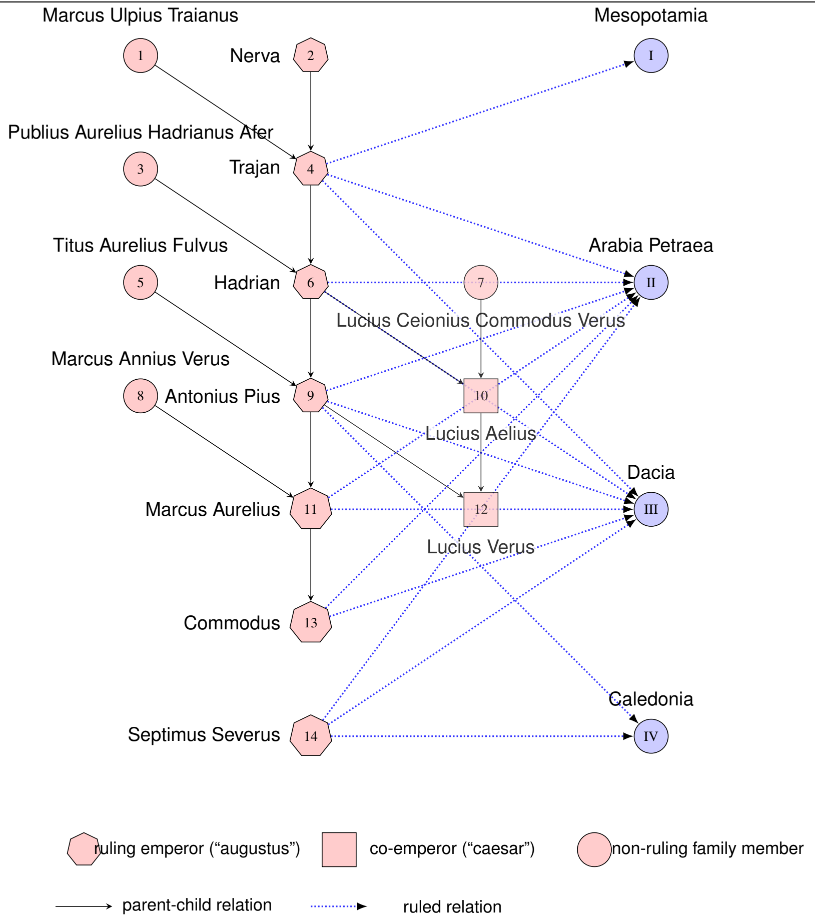
From a practical perspective, CIGs are actually quite common, in particular, where time-related or hierarchical data is involved: If relations, e.g., between Germany and kings, are time-related, it is quite likely that there will be some overlapping, e.g., for periods where two persons were king of Germany at the same time. Similarly, hierarchical data often has some limited anomalies that make a modelling as strict tree data impossible. Figure 1.11 shows actual data[38] on relations between the family (red nodes, non-ruling member ①, co-emperor or heir designate $\boxed{10}$, emperors 2) of the Roman emperors in the time of the "Five Good Emperors" (Edward Gibbon) in the 2nd century. It also shows, for actual emperors, which of the four new provinces (Ⓘ) added to the roman empire in this period each emperor ruled (the other provinces remained mostly unchanged and are therefore omitted). Arrows between family members indicate, natural or adoptive, fathership[39]. Arrows between emperors and provinces show rulership, different colors are used to distinguish different emperors. 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 previous example also highlights the intuition behind continuous-image graphs: we allow some overlapping between among the children of different nodes, but only in

[38] 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.

[39] Note that all emperors of the Nervan-Antonian dynasty except Nerva and Commodus were adopted by their predecessor and are therefore often referred to as "Adoptive Emperors".

Fig. 1.11 "The Five Good Emperors" (after Edward Gibbon), their relations, and provinces.



such a way that the images can still be represented (over some order on the nodes) as continuous intervals. Figure 1.12 illustrates the intervals on the Roman provinces for representing the ruled provinces of each emperor: With the given order on the provinces, each image is a single interval (e.g., Trajan I–III and Septimus Severus II–IV) even though the data is clearly not tree-shaped (or a closure relation of a tree-shaped relation).

How continuous-image graphs differ from tree-shaped data (or closure relations over tree-shaped data) is further detailed in Figure 1.13: Tree data carries the image disjointness property as, under the order on the nodes induced by a breadth-first traversal, the nodes in the image of any parent node in the tree form a continuous, non-overlapping interval. Closure relations over tree data (i.e., relations such as XPath's descendant) carry the image containment property as, e.g., under the order on the nodes induced by a depth-first traversal, again the nodes in the image of any parent node form a continu-

Fig. 1.12 Overlapping of province children in the "The Five Good Emperors" example, Figure 1.9

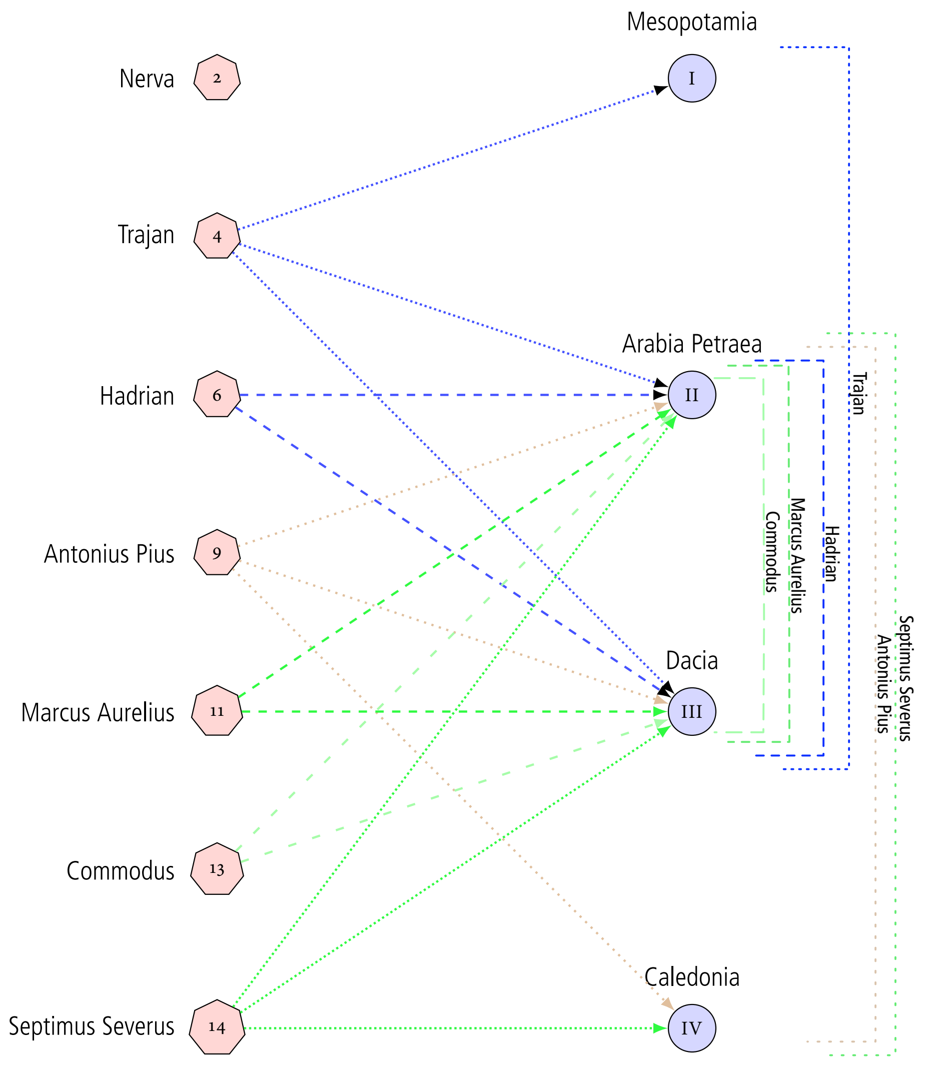
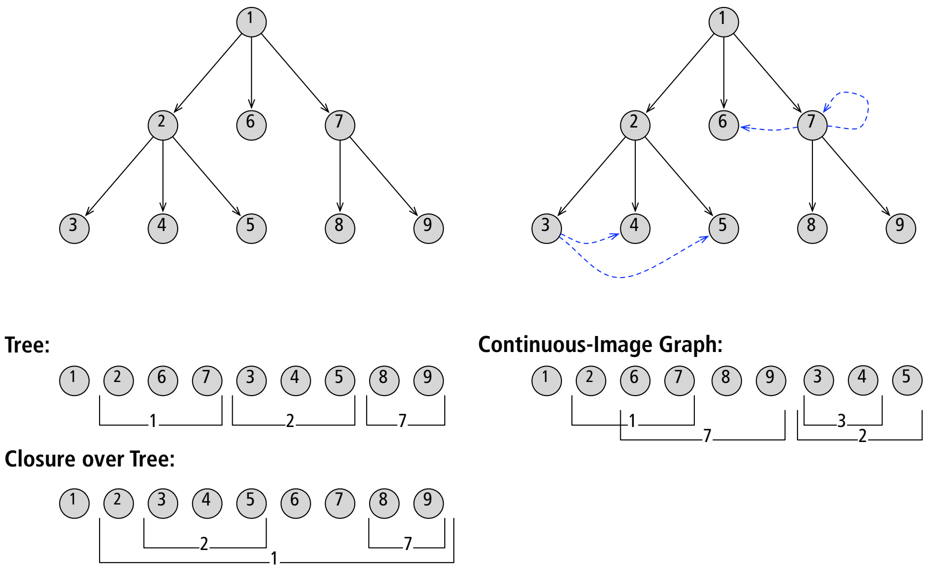


Fig. 1.13 Overlapping of images in trees, closure relations over trees, and continuous-image graphs



ous interval and overlapping is limited: either two such intervals do not overlap at all or one is contained within the other.

Continuous-image graphs (as shown in the right of Figure 1.13) carry, as stated above, the image interval property, i.e., there is some order on the nodes such that the nodes in the image of each parent form a continuous interval. Here, the intervals may overlap arbitrarily as illustrated in Figure 1.13. However, in contrast to the tree or closure relation over tree case the required order on the nodes is no longer known a-priori but must be determined for each graph using, e.g., the above described algorithms.

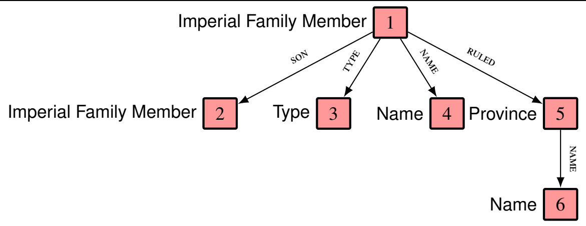
1.13.3 Intermediary Answers as Interval Labels

When we evaluate acyclic conjunctive or tree queries, we can observe that for determining matches for a given query node only the match for its parent and child in the query tree are relevant. Intuitively, this "locality" property holds as in a tree there is at most one path between two nodes. To illustrate, consider, e.g., the XPath query //a//b//c selecting c descendants of b descendants of a's. Say there are n a's in the data nested into each other with m b's nested inside the a's and finally inside the b's (again nested in each other) l c's. Then a naive evaluation of the above query considers all triples (a,b,c) in the data, i.e., $n \times m \times l$ triples. However, whether a c is a descendant of a b is independent of whether a b is a descendant of an a. If a b is a descendant of several a's makes no difference for determining its c descendants. It suffices to determine in at most $n \times m$ time and space all b's that are descendants of a, followed by a separate determination of all c's that are descendants of such b's in at most $m \times l$ time and space.

Indeed, if we consider the answer relation for a tree query, i.e., the relation with the complete bindings as rows and the query's nodes as columns, this relation always exhibits multivalue dependencies [64]: We can normalize or decompose such a relation for a query with n nodes into $n-1$ separate relations that together faithfully represent the original relation (and from which the original relation can be reconstructed using $n-1$ joins). This allows us to compact an otherwise potentially exponential answer (in the data size) into a polynomial representation.

This is the first principle of the algorithm: decompose the query into separate binding sequences for each query node with "links" or pointers relating bindings of different nodes. We thus obtain an exponentially more succinct data structure for (intermediary) answers of tree queries than if using standard (flat) relational algebra. In this sense, a sequence map can be considered a fully decomposed *column store* for the answer relation.

Fig. 1.14 Selecting sons, type, name, and ruled provinces for all members of the imperial family in the data of Figure 1.11.



To illustrate this, consider the query in Figure 1.14 on the data of Figure 1.11. The query selects sons and ruled provinces of members of the imperial family. We also record type and name of the family member and name of the province to easier talk about the retrieved data. The answers for such a query, if expressed, e.g., in relational algebra or any language using standard, flat relations to represent n-ary answers, against the data from Figure 1.11 yields the flat relation represented in Figure 1.15. As argued above, we can detect multivalue dependencies and thus redundancies, e.g., from emperor to province, from province to province name, from emperor (Imp-ID) to type and name.

To avoid these redundancies, we first decompose or normalize this relation along the multivalue dependencies as in Figure 1.16. For the sequence map, we use always a full decomposition, i.e., we would also partition type and name into separate tables as in a column store.

Storing Intermediary Results as Intervals Once we have partitioned the answer relation into what subsumes to only link tables as in column stores, we can observe even more regularities (and thus possibilities for compaction) if the underlying data is a tree or continuous-image graph. Look again at the data in Figure 1.11 and the resulting answer representation in Figure 1.16: Most emperors have not only ruled one of the new provinces Mesopotamia, Arabia Petraea, Dacia, and Caledonia but several. However,

Fig. 1.15 Answers for query from Figure 1.14, single, flat relation.

Imp-II	Type	Name	Son-II	Ruled-II	Ruled-Name
1	non-ruling	Marcus Ulpius Traianus	4	–	–
2	augustus	Nerva	4	–	–
3	non-ruling	P. Aurelius Hadrianus Afer	6	–	–
4	augustus	Trajan	6	I	Mesopotamia
4	augustus	Trajan	6	II	Arabia Petraea
4	augustus	Trajan	6	III	Dacia
5	non-ruling	Titus Aurelius Fulvus	9	–	–
6	augustus	Hadrian	9	II	Arabia Petraea
6	augustus	Hadrian	10	II	Arabia Petraea
6	augustus	Hadrian	9	III	Dacia
6	augustus	Hadrian	10	III	Dacia
7	non-ruling	L. Ceionius Commodus Verus	10	–	–
8	non-ruling	M. Annius Verus	11	–	–
9	augustus	Antonius Pius	11	II	Arabia Petraea
9	augustus	Antonius Pius	12	II	Arabia Petraea
9	augustus	Antonius Pius	11	IIi	Dacia
9	augustus	Antonius Pius	12	III	Dacia
9	augustus	Antonius Pius	11	IV	Caledonia
9	augustus	Antonius Pius	12	IV	Caledonia
10	caesar	Lucius Aelius	12	–	–
11	augustus	Marcus Aurelius	13	II	Arabia Petraea
11	augustus	Marcus Aurelius	13	III	Dacia
12	caesar	Lucius Verus	–	–	–
13	augustus	Commodus	–	II	Arabia Petraea
13	augustus	Commodus	–	III	Dacia
14	augustus	Septimus Severus	–	II	Arabia
14	augustus	Septimus Severus	–	III	Arabia
14	augustus	Septimus Severus	–	IV	Caledonia

Fig. 1.16 Answers for query from Figure 1.14, no multivalue dependencies.

Imp-II	Type	Name
1	non-ruling	M. Ulpius Traianus
2	augustus	Nerva
3	non-ruling	P. A. Hadrianus Afer
4	augustus	Trajan
5	non-ruling	Titus Aurelius Fulvus
6	augustus	Hadrian
7	non-ruling	L. C. Commodus Verus
8	non-ruling	M. Annius Verus
9	augustus	Antonius Pius
10	caesar	Lucius Aelius
11	augustus	Marcus Aurelius
12	caesar	Lucius Verus
13	augustus	Commodus
14	augustus	Septimus Severus

Imp-II	Son-ID
1	4
2	4
3	6
4	6
5	9
6	9
6	10
7	10
8	11
9	11
9	12
10	12
11	13

Imp-II	Prov-ID
4	I
4	II
4	III
6	II
6	III
9	II
9	III
9	IV
13	II
13	III
14	II
14	III
14	IV

Prov-II	Name
I	Mesopotamia
II	Arabia Petraea
III	Dacia
IV	Caledonia

Fig. 1.17 Answers for query from Figure 1.14, multiple relations, interval pointers. The first table from Figure 1.16 remains unchanged.

Imp-II	Son Range
1	4
2	4
3	6
4	6
5	9
6	9–10
7	10
8	11
9	11–12
10	12
11	13

Imp-II	Prov Range
4	I–III
6	II–III
9	II–IV
13	II–III
14	II–IV

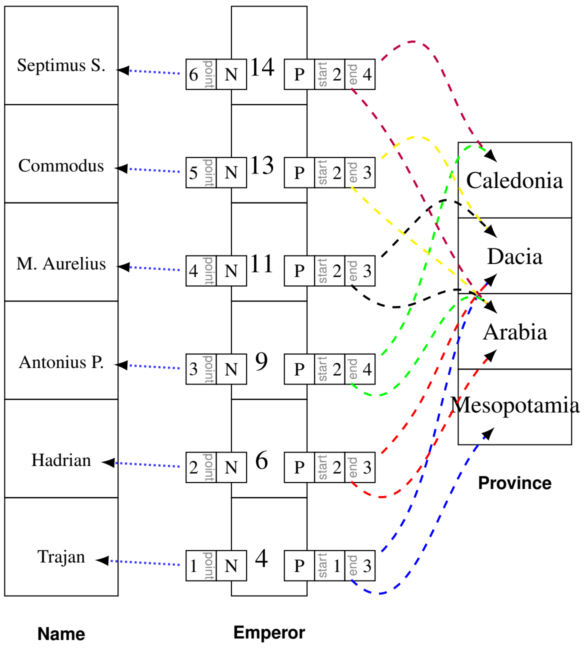
since the data is a continuous-image graph there is an order (indeed, the order of the province IDs if interpreted as roman numerals) on the provinces such that the provinces ruled by each emperor form a continuous interval w.r.t. that order. Thus we can actually represent the same information much more compactly using interval pointers or links as in Figure 1.17 where we do the same also for the father-son relation (although there is far less gain since most emperors already have only a single son).

Instead of a single relation spanning 28 rows and 6 columns (168 cells), we have thus reduced the information to $5 \cdot 2 + 11 \cdot 2 + 14 \cdot 3 = 74$ cells. This compaction increases exponentially if there are longer paths in a tree query (e.g., if the provinces would be connected to further information not related to the emperors). It increases quadratically with the increasing size of the tables, e.g., if we added the remaining n provinces of the Roman empire ruled by all emperors in our data we would end up with $7 \cdot n$ additional rows of 6 columns in the first case (each of the 7 emperors in our data ruled all these provinces), but only $n \cdot 2$ additional cells when using multiple relations and interval pointers.

Formally, we represent (intermediary) answers to an acyclic query as a mapping from the set of query variables V to sequences of matches for that query node. A *match* for query node v in itself is the actual data node or edge v is matched with and a set of pairs of child nodes of v to start and end positions. Intuitively, it connects the match for v to matches of its child nodes in the tree query. We obtain in this way a data structure as shown in Figure 1.18 for a query selecting roman emperors with their names and ruled provinces on the data of Figure 1.11.

Note, that we allow for each child node of v *multiple* intervals. If the data is a CIG, it is guaranteed that only a single interval is needed and thus the overall space complexity of a sequence map is linear in the data size. However, we can also employ a sequence map for non-CIG graphs. In this case, we often still benefit from the interval pointers, but in worst-case we might need $|N \cup E|$ many interval pointers to relate to all bindings of a child variable. Overall, a sequence map for non-CIG graphs thus may use up to quadratic space in the data size.

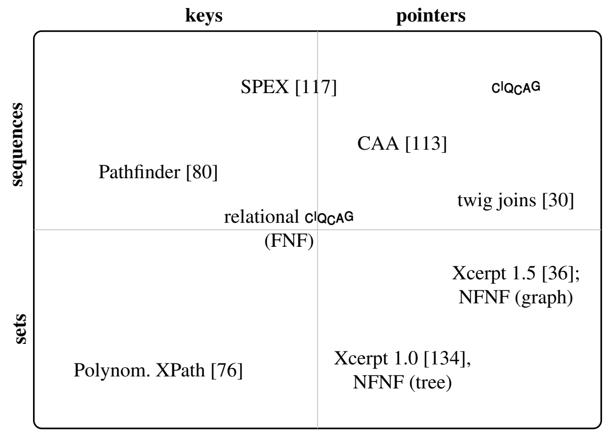
Fig. 1.18 Sequence Map: Example. For a query selecting roman emperors together with their name and ruled provinces on the data of Figure 1.11.


Septimus S.
Commodus
M. Aurelius
Antonius P.
Hadrian
Trajan
6 point N 14 P start 2 end 4
5 point N 13 P start 2 end 3
4 point N 11 P start 2 end 3
3 point N 9 P start 2 end 4
2 point N 6 P start 2 end 3
1 point N 4 P start 1 end 3
Caledonia
Dacia
Arabia
Mesopotamia
Province
Name
Emperor

Representing intermediary results: A Comparison As stated above, the sequence map is heavily influenced by previous data structures for representing intermediary answers of tree queries. Figure 1.19 shows the most relevant influences. Complexity and supported data shapes are compared below after discussing the actual evaluation of tree queries using the interval labeling for data and intermediary results. Here, we illustrate that the above discussed choices when designing a data structure for intermediary answers of tree queries are actually present in many related systems: We can find systems such as Xcerpt 1.0 [134], many early XPath processors (according to [76]), and tree algebras such as TAX [89] that use exponential size for storing all combinations of matches for each query node explicitly. [76] shows that XPath queries can in fact be evaluated in polynomial time and space, which is independently verified in SPEX [117], the first streaming processor for navigational XPath with all structural axes. Like SPEX and our approach, complete answer aggregates [113] use interval compaction for relating matches between different nodes in a tree query. CAAs are also most closely related to our approach w.r.t. the decomposition of the answer relation: fully decomposed without multivalue dependencies. In contrast, Pathfinder [22] uses standard relational algebra but for the evaluation of structural joins a novel staircase join [82] is employed that exploits the same interval principles used in CAAs and our approach.

Streaming or cursor-based approaches such as twig join approaches [30, 48] consider the data in a certain order rather than all at once. In such a model, it is possible and desirable to skip irrelevant portions of the input stream (or relations) and to prune

Fig. 1.19 Data structures for intermediary results (of a tree query)



partial answers as soon as it is clear that we can not complete such answers. Recent versions of SPEX [37, 117] contain as most twig join approaches, mechanisms to skip over parts of the stream (at least for some query nodes) if there can not be a match (e.g., because there is no match for a parent node and we know that matches for parent nodes must come before matches for child nodes). Both twig joins and SPEX also prune results as soon as possible. However, twig joins are limited to vertical relations (child and descendant) whereas SPEX and CIQCAG can evaluate all XPath axes, though only on tree data.

To summarize, though our approach to representing intermediary answers is similar in its principles to several of the related approaches in Figure 1.19, it combines *efficient intermediary answer storage* as in CAAs with *fully algebraic* processing as in Pathfinder and *efficient skipping and pruning* as in twig joins.

Furthermore, where most of the related approaches are limited to tree data (with the notable exception of Xcerpt), our approach allows processing of *many graphs*, viz. CIGS, as efficient as previous approaches allow for trees.

1.13.4 Evaluating Tree Queries on Interval Labels

For evaluating acyclic conjunctive queries (or tree queries) the essential operation is a join that allows us to gather results of the evaluation of the constituent query atoms in a consistent manner: Only where variables are bound the same in answers to different query atoms those answers are "joined" together to form a larger answer.

There are some other operations needed for implementing full acyclic conjunctive queries, most importantly selection, but they are omitted here for clarity of presentation. For full details see [68].

The join operation for interval representations of relations is defined as follows:

Definition 27 (Sequence map join (disjoint edge covers)). *Let D be a relational structure, Q a tree query, and S_1, S_2 two interval representations for D over Q such that there* *is no edge in Q that is covered by both S_1 and S_2. Then $\bowtie_{[]}(S_1, S_2)$ returns an interval representation S_3 such that*

1. *the induced relation of S_3 is the natural join of the induced relations of S_1 and S_2.*
2. *$S_3|_{dom S_1 \cup dom S_2} = S_3$ (S_3 contains bindings only for variables mapped either in S_1 or in S_2).*

Note that this definition yields an interval representation that leaves bindings for *non-shared* variables unchanged from the input representations. These variables occur only in one of the query parts covered by the input, but not in the other. For *shared* variables, only those bindings are retained that occur in both representations. This also applies to the (interval pointer) references from bindings of a parent variable v to a child variable v' of v: They are contained only in one of the sequence maps (due to the edge cover restriction), for the other sequence map the induced relation records *any* combination of bindings by definition.

The restriction on the edge covers on S_1 and S_2 is imposed to ensure that for any pair of variables v, v' only one of the interval representations may contain interval pointers from v to v', though both may contain *bindings* for v and v'. In other words, each *edge* of the query is enforced by at most one of the two sequence maps.

For a given tree query expression, the edge cover of each sub-expression can be determined statically, without knowledge about the data the expression is to be evaluated against. Thus, we can also statically determine whether a join expression is valid or violates the edge cover restriction defined above. For the evaluation of tree queries we never need joins with overlapping edge covers.

Algorithm 25 computes an interval representation that represents the join of the induced relations as demanded in the definition of $\bowtie_{[]}()$, but may be inconsistent: It "bombs" bindings not contained in both sequence maps rather than dropping them entirely. This has the effect that interval pointers can remain unchanged (but now point to an interval containing possibly bombed entries). Note, that interval pointers to bindings of a variable occur only in one of the two input interval representations as the incoming edge of each variable is unique (since the query is tree-shaped) and the edge covers are disjoint. This allows line 16 where we simply throw together intervals from both sequence maps. Finally, observe that by the definition of the initialization of a sequence map, bindings for the same query variable occur in the *same* order in all sequence maps for that query. Thus the bindings of a variable shared between the two interval representations are ordered the same.

These observations are exploited in Algorithm 25 to give a merge-join [72] style algorithm for the join of two interval representations with disjoint edge cover that has linear time complexity in the (combined) size of the inputs. Since the bindings are already in the same order, we can omit the sort phase of the merge join and immediately merge the two binding sequences. However, we need to ensure that not only the order but also the number of bindings (and the position of eventual failure markers, cf. lines 18–20) reflects that for the same variable v in the sequence map where v's incoming edge is in the edge cover (lines 9–11).

Theorem 5. *Algorithm 25 computes $\bowtie_{[]}(S_1, S_2)$ for interval representations with disjoint edge cover and set of shared variables Shared in $O(b^{total}_{\mathsf{Shared}} \cdot i) \leq O(|\mathsf{Shared}| \cdot n \cdot i)$*

Algorithm 1: $\ltimes_{[]}(S_1, S_2)$

```
input  : Interval representations $S_1$ and $S_2$ with *disjoint* edge covers
output: Interval representation res representing the join of the induced relations
        of the inputs
1  $\textsc{ec}_1 \leftarrow$ edgeCover($S_1$); $\textsc{ec}_2 \leftarrow$ edgeCover($S_2$);
2  AllVars $\leftarrow domS_1 \cup domS_2$;
3  SharedVars $\leftarrow domS_1 \cap domS_2$;
4  res $\leftarrow \emptyset$ ;
5  foreach $v \in$ AllVars do
6      if $v \notin domS_2$ then res $\leftarrow$ res $\cup \{(v, S_1(v))\}$ ;
7      else if $v \notin domS_1$ then res $\leftarrow$ res $\cup \{(v, S_2(v))\}$;
8      else                                                  // v is in both
           // 1 is the primary (fallback if v is in neither edge cover)
9          iter $\leftarrow S_1(v)$; alt $\leftarrow S_2(v)$ ;
10         if $(v', v) \in \textsc{ec}_2$ for some $v'$ then
               // v is sink in $\textsc{ec}_2$, thus the order and number of entries must be as in 2
                 (it can not be sink in $\textsc{ec}_1$ as Q tree query and edge covers disjoint)
11             iter $\leftarrow S_2(v)$; alt $\leftarrow S_1(v)$;
12         $S \leftarrow \emptyset$; $i, j, k \leftarrow 1$ ;
13         while $i \leq$ |iter| do
14             $(n_1, i) \leftarrow$ nextBinding($S_1(v), i$); $(n_2, j) \leftarrow$ nextBinding($S_2(v), j$); if
               $n_1 = n_2$ then                               // Retain binding if same
15                 $S[k] = (n_1,$ intervals(iter[$i$]) $\cup$ intervals(alt[$j$])) ;
16                 $i$++; $j$++; $k$++;
17             else if $n_1 < n_2$ then              // "bomb" if in iter but not in alt
18                 $S[k] = ⚡$;
19                 $i$++; $k$++;
20             else                                  // skip binding if in alt but not in iter
21                 $j$++;
22
23         res $\leftarrow$ res $\cup \{(v, S)\}$ ;
24
25 return res
```

time where $b^{total}_{\mathsf{Shared}}$ *is the total number of bindings associated in either input with a variable in* Shared *and i is the maximum number of intervals associated with any such binding. For tree, forest, and* CIG *data* $i = 1$, *for arbitrary graph data* $i \leq n$.

Proof. Algorithm 25 computes $S = \ltimes_{[]}(S_1, S_2)$: For any variable v, if a binding for v occurs in the induced relation of both interval representations, it occurs also in S due to lines 15–17. If v's incoming edge is in the edge cover of one of the interval representations S', lines 9–11 ensure that the sequence of bindings for v is the same

Algorithm 2: NextBinding(S, i)

```
input  : Sequence $S$ containing, possibly, failure markers and start index $i$
output: The next element in $S$ at or after $i$ that is not a failure marker and its
        index or $(\infty,\infty)$ if no such binding exists
1 for $j \leftarrow i$ to $|S|$ do
2     if $S[j]$ is not a failure marker then break;
3 if $j = |S|$ and $S[j]$ failure marker then return $(\infty,\infty)$ ;
4 return $(S[j], j)$
```

(except that some bindings are "bombed") in S as in S'. For the parent v' of v, if a binding is retained the set of intervals from both interval representations are copied en block. There are only intervals in S' (as (v', v) is not in the edge cover of the other interval representation) and thus only those relations between bindings of v and v' as in the induced relation of S' are retained. This is proper as in the induced relation of the other interval representation *all* bindings of v are related to all bindings of v' by definition of the induced relation. Both input sequences may be inconsistent: The presence of failure markers in either sequence does not affect the correctness of the algorithm: failure markers in alt are skipped, failure markers in iter are retained (lines 15–17) as intended. Dangling bindings do not affect the algorithm.

Algorithm 25 loops over all shared variables of S_1 and S_2 and for each such variable it iterates over all bindings in the primary interval representation iter and corresponding bindings in alt, skipping, if necessary, bindings in alt not in iter. In the loop lines 13–22 i or j is incremented (possibly multiple times, if failure markers are skipped in NextBinding) until either $i > |\mathsf{iter}|$. If j ever becomes $> |\mathsf{alt}|$ subsequent calls of $\mathsf{binding}(()\mathsf{alt}[j])$ return, by definition, a value larger than all $n \in \mathsf{Nodes}(D)$.

Thus the algorithm touches, for each shared variable, each entry in either interval representation at most once (and touches one proper (not a failure mark) entry in each step of the loop 13–22). Thus it runs in $O(b^{total}_{\mathsf{Shared}} \cdot i)$ where $b^{total}_{\mathsf{Shared}}$ is the total number of bindings in both interval representations for a shared variable and i is the maximum number of intervals per binding. This is bound by $O(|\mathsf{Shared}| \cdot n \cdot i)$ for any interval represenation (including those for arbitrary graphs).

It is worth pointing out, that a tree query any variable is shared at most once for each in- or outgoing edge and for each unary relation associated with the variable. Thus, even if there are $O(q)$ joins in the expression, the accumulated number of shared variables among all those joins is also only $O(q)$ an thus the total complexity for only those joins is bounded by $O(q \cdot n \cdot i)$.

1.13.5 Comparison

The above outlined algorithm for a join on interval representations can be easily extended to an algorithm for full tree queries, see [68]. In the following we compare our approach, called CIQCAG, to previous approaches for tree query evaluation on XML (i.e., limited to *tree data*).

	time	space
Structural Joins, relational join	$O(q \cdot n \cdot \log n)$	$O(q \cdot n^2)$
———, *structure-aware join*	$O(q \cdot n)$	$O(q \cdot n^2)$
Twig or Stack Joins	$O(q \cdot n)$	$O(q \cdot n + n \cdot d)$
PDA-based (here: SPEX)	$O(q \cdot n \cdot d)$	$O(q \cdot n)$
Interval-based (here: CIQCAG)	$O(q \cdot n)$	$O(q \cdot n)$

Table 1.19. Approaches for XML Tree Query Evaluation. n: number of nodes in the data, d: depth of data; q: size of query. We assume constant membership test for all structural relations.

To keep the discussion focused we ignore index-based evaluation of XML. Though path indices such as the DataGuide [75] or IndexFabric [55] and more recent variants [51] can significantly speed up path queries they suffer from two anomalies: First, if a tree query contains many branching nodes (i.e., nodes with more than one children) they generally do not perform better than, e.g., the structural join approach below. Second, even though only path queries can be directly answered from the index, the index size can be significantly higher than the size of the original XML documents.

We can classify most of the remaining approaches to the evaluation of XML tree queries in four classes (the corresponding complexity for evaluation XPath (and similar) tree queries on tree data is summarized in Table 1.19):

1. *Structural joins:* The first class is most reminiscent of query evaluation for relational queries and arguable inspired by earlier research on acyclic conjunctive queries on relational databases [77]. Tree queries are decomposed into a series of (structural) joins. Each structural join enforces one of the structural properties of the given query, e.g., a child or descendant relation between nodes or a certain label. Proposed first in [8], structural joins have also been used to great effect for studying the complexity of XPath evaluation and proposing the first polynomial evaluation of full XPath [76]. Due to its similarity with relational query evaluation it has proved to be an ideal foundation for implementing XPath and XQuery on top of relational databases [79]. It turns out, however, that the use of standard joins is often not an ideal choice and structure- or tree-aware joins [22] (that take into consideration, e.g., that only nodes in the sub-tree routed at another node can be that nodes a-descendants) can significantly improve XPath and XQuery evaluation.

2. *Twig joins:* In sharp contrast, the second class employs a single (thus called *holistic*) operator for solving an entire tree query rather than decomposing it into structural joins. These approaches are commonly referred to as twig or stack join [30, 49] and essentially operate by keeping one stack for each step in, e.g., an XPath query representing partial answers for the corresponding node-set. Theses stacks are organized hierarchically with (where possible, implicit) parent pointers connecting partial answers for upper stack entries to those of lowers. The approaches mostly vary in how the stacks

are populated. In contrast to the other approaches, twig joins are limited to vertical, i.e., child and descendant, axes and have not been adapted for the full range of XPath axes. They also, like structure-aware joins [22], exploit the tree-shape of the data and can, at best, be adapted to DAGs [48].

3. *PDA-based:* Where twig joins assume one stream of nodes from the input document for each stack (and thus XPath step), the third class of approaches based on pushdown automata aims to evaluate XPath queries on a single input stream similar to a SAX event stream. SPEX, e.g., [118, 119, 117] also maintains a record of partial answers for each XPath step, but minimizes used memory more efficiently and exploits the existential nature of most XPath steps by maintaining only generic conditions rather than actual pointers to elements from the XML stream (except for candidates of the actual results set, of course). Also it supports all XPath axes in contrast to the twig join approaches. The cost is a slightly more complex algorithm.

4. *Interval-based*: Finally, interval-based approaches are a combination of the tree awareness in twig joins and SPEX and the structural join approach: The query is decomposed into a series of structural relations, but each relation is organised in such a way that all elements related to one element of its parent step are in a single continuous interval. This allows both an efficient storage and join of intermediate answers. The first interval-based approach are the Complete Answer Aggregates (CAA) [114, 113]. Here theCIQCAG algebra is proposed which improves on the complexity of CAA (to the linear complexity given in Table 1.19) and covers, in contrast to CAA, arbitrary tree-shaped relations. It is also shown that interval-based approaches can be extended even to a large, efficiently detectable class of graph data (so called continuous-image graphs) that is not covered by any of the other linear time approaches discussed above.

Currently, extensions of the above algorithms for larger classes of graph data are investigated, e.g., in [48] and [68].

1.13.6 Conclusion

In this chapter, we present a *novel characterization* of a class of graphs that is a proper, non-trivial superclass of trees that still exhibits a *labeling scheme with constant time, constant per-node space* adjacency and reachability tests. Furthermore, we give a quadratic algorithm that computes, for an arbitrary graph, such a labeling if one exists.

Constant time membership test almost immediately yields linear time evaluation for existential acyclic conjunctive queries on tree data. However, nave approaches for n-ary universal queries take at least quadratic time in the graph size. We show how the above labeling scheme can be exploited to give an *algorithm for evaluating acyclic conjunctive queries* that is $O(n \cdot q)$ wrt. time and space complexity, i.e., linear in both data and program complexity. Furthermore, our algorithm guarantees iteration in the size of the related nodes rather than in all nodes.

1.14 Versatile Evaluation II: Rule Scaling under Rich Unification

In the preceeding sections, we have considered the efficient evaluation of Xcerpt queries by a translation to a relational normal form. This section deals with the subsumption relationship between Xcerpt query terms. Deciding subsumption has traditionally been an important means for optimizing multiple queries against the same set of data and can be used for improving termination of Xcerpt programs in a backward chaining evaluation engine.

Xcerpt query terms (Definition 2) are an answer to accessing Web data in a rule-based query language. Like most approaches to Web data (or semi-structured data, in general), they are distinguished from relational query languages such as SQL by a set of query constructs specifically attuned to the less rigid, often diverse, or even entirely schema-less nature of Web data. As Definitions 2 (Xcerpt Query Term) and 7 suggest, Xcerpt terms are similar to normalized forward XPath (see [120]) but extended with variables, deep-equal, a notion of injective match, regular expressions, and full negation. Thus, they achieve much of the expressiveness of XQuery without sacrificing the simplicity and pattern-structure of XPath.

When used in the context of Xcerpt, query terms serve a similar role to terms of first-order logic in logic languages. Therefore, the notion of unification has been adapted for Web data in [133], there called "simulation unification". Simulation is recapitulated in Definition 8. This form of unification is capable of handling all the extensions of query terms over first-order terms that are needed to support Web data: selecting terms at arbitrary depth ($R_{\mathsf{descendant}}$), distinguishing partial from total terms, regular expressions instead of plain labels, negated subterms (`without`), etc.

The notions of query term, simulation and substitution sets are exemplified in Section 1.3 and formally defined in 1.7. In this section, we consider query containment between two Xcerpt terms.

Subsumption or containment of two queries (or terms) is an established technique for optimizing query evaluation: a query q_1 is said to be *subsumed* by or *contained* in a query q_2 if every possible answer to q_1 against every possible data is also an answer to q_2. Thus, given all answers to q_2, we can evaluate q_1 only against those answers rather than against the whole database.

For first-order terms, subsumption is efficient and employed for guaranteeing termination in tabling (or memoization) approaches to backward chaining of logic [143, 50]. However, when we move from first-order terms to Web queries, subsumption (or containment) becomes quickly less efficient or even intractable. Xcerpt query terms have, as pointed out above, some similarity with XPath queries. Containment for various fragments of XPath is surveyed in [138], both in absence and in presence of a DTD. Here, we focus on the first setting, where no additional information about the schema of the data is available. However, Xcerpt query terms are a strict super-set of (navigational) XPath as investigated in [138]. In particular, the Xcerpt query terms may contain (multiple occurrences of the same) variables. This brings them closer to *conjunctive queries* (with negation and deep-equal), as considered in [151] on general relations, and in [18] for tree data. Basic Xcerpt query terms can be reduced to (unions of) conjunctive queries with negation. However, the injectivity of Xcerpt query terms (no two siblings may match with the same data node) and the presence of deep-equal (two nodes are deep-equal iff they have the same structure) have no direct counterpart in conjunctive query containment. Though [99] shows how inequalities in general affect conjunctive query containment, the effect of injectivity (or all-distinct constraints) on query containment has not been studied previously. The same applies to deep-equal, though the results in [102] indicate that in *absence* of composition deep-equal has no effect on evaluation and thus likely on containment complexity.

For Xcerpt query terms, subsumption is, naturally, of interest for the design of a terminating, efficient Xcerpt engine. Beyond that, however, it is particularly relevant in a Web setting. Whenever we know that one query subsumes another, we do not need to access whatever data the two queries access twice, but rather can evaluate both queries with a single access to the basic data by evaluating the second query on the answers of the first one. This can be a key optimization also in the context of search engines, where answers to frequent queries can be memorized so as to avoid their repeated computation. Even though today's search engines are rather blind of the tree or graph structure of HTML, XML and RDF data, there is no doubt that some more or less limited form of structured queries will become more and more frequent in the future (see Google scholar's "search by author, date, etc."). Query subsumption, or containment, is key to a selection of queries, the answers to which are to be stored so as to allow as many queries as possible to be evaluated against that small set of data rather than against the entire search engine data. Thus, the notion of simulation subsumption proposed in this chapter can be seen as a building block of future, structure-aware search engines.

Therefore, we study in this section subsumption of Xcerpt query terms. The main building blocks of this section are the following.

– we introduce and formalize a notion of subsumption for Xcerpt query terms, called *simulation subsumption*, in Section 1.14.2. To the best of our knowledge, this is the first notion of subsumption for queries with injectivity of sibling nodes and deep-equal.
– we show, also in Section 1.14.2, that simulation on ground query terms is equivalent to simulation subsumption.[40] This also shows that ground query term simulation as introduced in [133] indeed captures the intuition that a query term that simulates into another query term subsumes that term.
– we define, in Section 1.14.3, a *rewriting system* that allows us to reduce the test for subsumption of q in q' to finding a sequence of syntactic transformations that can be applied to q to transform it into q'.
– we show, in Section 1.14.4, that this rewriting system gives rise to an algorithm for testing subsumption that is sound and complete and can determine whether q subsumes q' in time $O(n!^n)$. In particular, this shows that simulation subsumption is decidable.

[40] With small adaptions of the treatment of regular expressions and negated subterms in query term simulation.

1.14.1 Xcerpt Basics: Query Terms and Simulation

Query terms are an abstraction for queries that can be used to extract data from semi-structured trees. In contrast to XPath queries, they may contain (multiple occurrences of the same) variables and demand an *injective mapping* of the child terms of each term. For example, the XPath query `/a/b[c]/c` demands that the document root has label `a`, and has a child term with label `b` that has itself a child term with label `c`. The subterm `c` that is given within the predicate of `b` can be mapped to the same node in the data as the child named `c` of `b`. Therefore, this XPath query would be equivalent to the query term $a\{\{b\{\{c\}\}\}\}$, but not to $a\{\{b\{\{c,c\}\}\}\}$. Simulation could be, however, easily modified to drop the injectivity requirement.

1.14.2 Simulation Subsumption

In this section, we first introduce simulation subsumption (Definition 28), then for several query terms we discuss whether one subsumes the other to give an intuition for the compositionality of the subsumption relationship. Subsequently, the transitivity of the subsumption relationship is proven (Lemma 4), some conclusions about the membership in the subsumption relationship of subterms, given the membership in the subsumption relationship of their parent terms are stated. These conclusions formalize the compositionality of simulation subsumption and are a necessary condition for the completeness of the rewriting system introduced in Section 1.14.3.

In tabled evaluation of logic programs, solutions to subgoals are saved in a solution table, such that for equivalent or subsumed subgoals, these sets do not have to be recomputed. As mentioned before, this avoidance of re-computation does not only save time, but can, in certain cases be crucial for the termination of a backward chaining evaluation of a program. In order to classify subgoal as solution or look-up goals, boolean subsumption as specified by Definition 28 must be decided. Although Xcerpt query terms may contain variables, n-ary subsumption as defined in [138] would be too strict for our purposes. To see this, consider the Xcerpt query terms $q_1 := a\{\{var\ X\}\}$ and $q_2 := a\{\{c\}\}$. Although all data terms that are relevant for q_2 can be found in the solutions for q_1, q_1 and q_2 cannot be compared by n-ary containment, because they differ in the number of their query variables.

Definition 28 (Simulation Subsumption). *A query term* q_1 subsumes *another query term* q_2 *if all data terms that* q_2 *simulates with are also simulated by* q_1.

Example 1 (Examples for the subsumption relationship). Let the query terms $q_1, \ldots q_5$ be given by:

- $q_1 := a\{\{\}\}$
- $q_2 := a\{\{desc\ b, desc\ c, d\}\}$
- $q_3 := a\{\{desc\ b, c, d\}\}$
- $q_4 := a\{\{without\ e\}\}$
- $q_5 := a\{\{without\ e\{\{without\ f\}\}\}\}$

Then the following subsumption relationships hold:

- q_2 subsumes q_3 because it requires less than q_3: While q_3 requires that the data has outermost label a, subterms c and d as well as a descendant subterm b, q_2 requires not that there is a direct subterm c, but only a descendant subterm. Since every descendant subterm is also a direct subterm, all data terms simulating with q_3 also simulate with q_2.
  But the subsumption relationship can also be decided in terms of simulation: q_2 subsumes q_3, because there is a mapping π from the direct subterms $ChildT(q_2)$ of q_2 to the direct subterms $ChildT(q_3)$ of q_3, such that q_i subsumes $\pi(q_i)$ for all q_i in $ChildT(q_2)$.
- q_3 does not subsume q_2, since there are data terms that simulate with q_2, but not with q_3. One such data term is $d := a\{b, e\{c\}, d\}$.
  Again, the subsumption relationship between q_3 and q_2 (in this order) can be decided by simulation. There is no mapping π from the direct subterms of q_3 to the direct subterms of q_2, such that a simulates into $\pi(a)$.
- q_1 subsumes q_4 since it requires less than q_4. All data terms that simulate with q_4 also simulate with q_1.
- q_4 does not subsume q_1, since the data term $a\{\{e\}\}$ simulates with q_1, but does not simulate with q_4.
- q_5 subsumes q_4, but not the other way around.

Proposition 4. *The subsumption relationship between query terms is transitive, i.e. for arbitrary query terms q_1, q_2 and q_3 it holds that if q_1 subsumes q_2 and q_2 subsumes q_3, then q_1 subsumes q_3.*

Proposition 4 immediately follows from the transitivity of the subset relationship. Query term simulation and subsumption are defined in a way such that, given the simulation subsumption between two query terms, one can draw conclusions about subsumption relationships that must be fulfilled between pairs of subterms of the query terms. Lemma 1 formalizes these sets of conclusions.

Lemma 1 (Subterm Subsumption). *Let q_1 and q_2 be query terms such that q_1 subsumes q_2. Then there is an injective mapping π from $ChildT^+(q_1)$ to $ChildT^+(q_2)$ such that q_1^i subsumes $\pi(q_1^i)$ for all $q_1^i \in ChildT^+(q_1)$.*

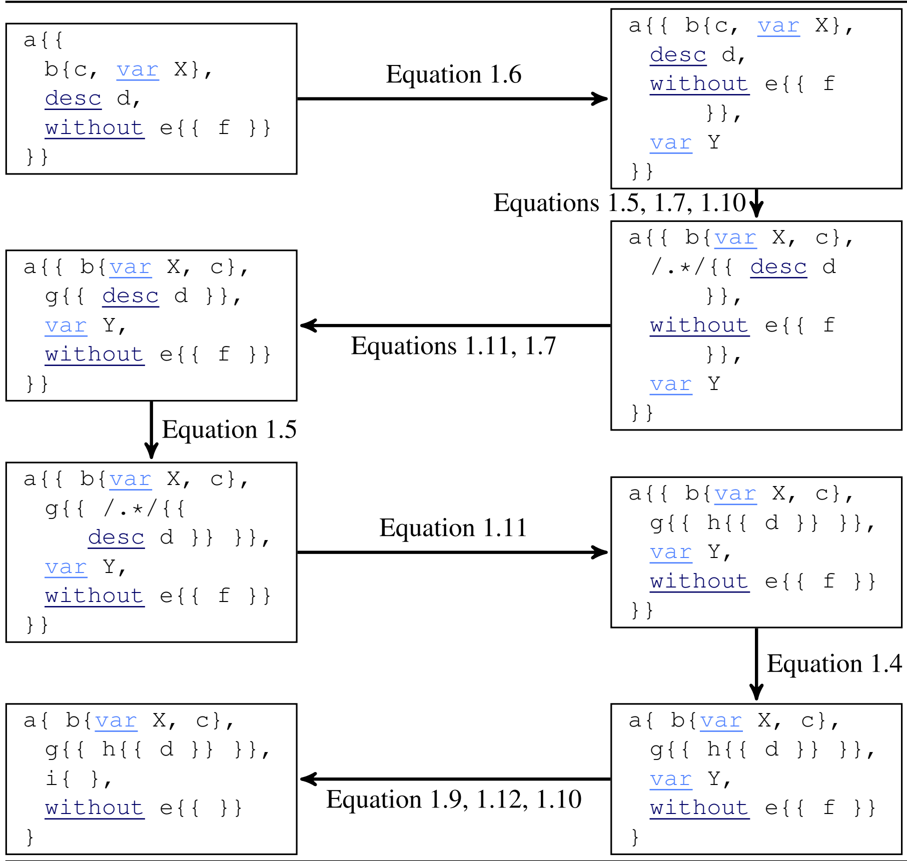
Furthermore, if q_1 and q_2 are breadth-incomplete, then there is a (not necessarily injective) mapping σ from $ChildT^-(q_1)$ to $ChildT^-(q_2)$ such that $pos(\sigma(q_1^j))$ subsumes $pos(q_1^j)$ for all $q_1^j \in ChildT^-(q_1)$.

If q_1 is breadth-incomplete and q_2 is breadth-complete then there is no q_1^j in $ChildT^-(q_1)$ and $q_2^k \in ChildT^+(q_2) \setminus range(\pi)$ such that $pos(q_1^j) \preceq q_2^k$.

Lemma 1 immediately follows from the equivalence of the subsumption relationship and the extended query term simulation (see Lemma 4 in the appendix of [35]).

1.14.3 Simulation Subsumption by Rewriting

In this section, we lay the foundations for a proof for the decidability of subsumption between query terms according to Definition 28 by introducing a rewriting system from



```
a{{
  b{c, var X},
  desc d,
  without e{{ f }}
}}
```

```
a{{ b{c, var X},
  desc d,
  without e{{ f
      }},
  var Y
}}
```

```
a{{ b{var X, c},
  /.*/{{ desc d
      }},
  without e{{ f
      }},
  var Y
}}
```

```
a{{ b{var X, c},
  g{{ desc d }},
  var Y,
  without e{{ f }}
}}
```

```
a{{ b{var X, c},
  g{{ /.*/{{
      desc d }} }},
  var Y,
  without e{{ f }}
}}
```

```
a{{ b{var X, c},
  g{{ h{{ d }} }},
  var Y,
  without e{{ f }}
}}
```

```
a{ b{var X, c},
  g{{ h{{ d }} }},
  var Y,
  without e{{ f }}
}
```

```
a{ b{var X, c},
  g{{ h{{ d }} }},
  i{ },
  without e{{ }}
}
```

one query term to another, which is later shown to be sound and complete. Furthermore, this rewriting system lays the foundation for the complexity analysis in Section 1.14.4.

The transformation of a query term q_1 into a subsumed query term q_2 is exemplified in Figure 1.14.3.

Definition 29 (Subsumption monotone query term transformations). *Let q be a query term. The following is a list of so-called* subsumption monotone *query term transformations.*

– *if q has incomplete subterm specification, it may be transformed to the analogous query term with complete subterm specification.*

$$\frac{a\{\{q_1,\ldots,q_n\}\}}{a\{q_1,\ldots,q_n\}}, \tag{1.4}$$

– *if q is of the form desc q′ then the descendant construct may be eliminated or it may be split into two descendant constructs separated by the regular expression* `/.*/`*, the inner descendant construct being wrapped in double curly braces.*

$$\frac{desc\ q}{q}, \qquad \frac{desc\ q}{desc\ /.*/\{\{desc\ q\}\}} \tag{1.5}$$

– *if q has incomplete-unordered subterm specification, then a fresh variable X may be appended to the end of the subterm list. A* fresh *variable is a variable that does not occur in q_1 or q_2 and is not otherwise introduced by the rewriting system.*

$$X\ \text{fresh} \Rightarrow \frac{a\{\{q_1,\ldots,q_n\}\},}{a\{\{q_1,\ldots,q_n, var\ X\}\}} \tag{1.6}$$

– *if q has unordered subterm specification, then the subterms of q may be arbitrarily permuted.*

$$\pi \in Perms(\{1,\ldots,n\}) \Rightarrow \frac{\mathtt{a\{\{}q_1\mathtt{,\ \ldots,\ }q_n\mathtt{\}\}}}{\mathtt{a\{\{}q_{\pi(1)}\mathtt{,\ \ldots,\ }q_{\pi(n)}\mathtt{\}\}}} \tag{1.7}$$

$$\pi \in Perms(\{1,\ldots,n\}) \Rightarrow \frac{\mathtt{a\{}q_1\mathtt{,\ \ldots,\ }q_n\mathtt{\}}}{\mathtt{a\{}q_{\pi(1)}\mathtt{,\ \ldots,\ }q_{\pi(n)}\mathtt{\}}} \tag{1.8}$$

– *if q contains a variable* `var X`*, which occurs in q at least once in a positive context (i.e. not within the scope of a without) then all occurrences of* `var X` *may be substituted by another Xcerpt query term.*

$$X \in PV(q), t \in QTerms \Rightarrow \frac{q}{q\{X \mapsto t\}} \tag{1.9}$$

This rule may only be applied, if q contains all *occurrences of X in q_1. Furthermore, no further rewriting rules may be applied to the replacement term t.*
If a variable appears within q only in a negative context (i.e. within the scope of a `without`*), the variable cannot be substituted by an arbitrary term to yield a transformed term that is subsumed by q. The query terms* `a{{ without var X }}` *and* `a{{ without b{ } }}` *together with the data term* `a{ c }` *illustrate this characteristic of the subsumption relationship. For further discussion of substitution of variables in a negative context see Example 2.*

– *if q has a subterm q_i, then q_i may be transformed by any of the transformations in this list except for Equation 1.9 to the term $t(q_i)$, and this transformed version may be substituted at the place of q_i in q, as formalized by the following rule:* [41] [42]

$$\frac{q_i}{t(q_i)} \Rightarrow \frac{a\{\{q_1,\ldots,q_n\}\}}{a\{\{q_1,\ldots,q_{i-1},t(q_i),q_{i+1},\ldots q_n\}\}} \tag{1.10}$$

– *if the label of q is a regular expression e, this regular expression may be replaced by any label that matches with e, or any other regular expression e′ which is subsumed by e (see Definition 8 in the appendix of [35]).*[41]

[41] The respective rules for complete-unordered subterm specification, incomplete-ordered subterm specification and complete-ordered subterm specification are omitted for the sake of brevity.

[42] The exclusion of Equation 1.9 ensures that variable substitutions are only applied to entire query terms and not to subterms. Otherwise the same variable might be substituted by different terms in different subterms.

$$e \in RE, e \textit{ subsumes } e' \Rightarrow \frac{e\{\{q_1,\ldots,q_n\}\}}{e'\{\{q_1,\ldots,q_n\}\}} \tag{1.11}$$

– *if q contains a negated subterm q_i = without r and r' is a query term such that $t(r') = r$ (i.e. r' subsumes r) for some transformation step t, then q_i can be replaced by q'_i := without r'.*

$$(q_i = \textit{without } r) \wedge \frac{r'}{r} \wedge (q'_i = \textit{without } r') \Rightarrow \frac{a\{\{q_1,\ldots,q_i,\ldots,q_n\}\}}{a\{\{q_1,\ldots,q'_i,\ldots q_n\}\}} \tag{1.12}$$

1.14.4 Properties of the Rewriting System

In this section, we show that the rewriting system introduced in the previous section is sound (Section 1.14.4) and complete (Section 1.14.4). Furthermore, we study the structure of the search tree induced by the rewriting rules, show that it can be pruned without losing the completeness of the rewriting system and conclude that simulation subsumption is decidable. Finally we derive complexity results from the size of the search tree in Section 1.14.4.

Subsumption Monotonicity and Soundness

Lemma 2 (Monotonicity of the transformations in Definition 29). *All of the transformations given in Definition 29 are subsumption monotone, i.e. for any query term q and a transformation from Definition 29 which is applicable to q, q subsumes $t(q)$.*

The proof of Lemma 2 is straight-forward since each of the transformation steps can be shown independently of the others. For all of the transformations, inverse transformation steps t^{-1} can be defined, and obviously for any query term q it holds that $t^{-1}(q)$ subsumes q.

Lemma 3 (Transitivity of the subsumption relationship, monotonicity of a sequence of subsumption monotone query term transformations). *For a sequence of subsumption monotone query term transformations $t_1,\ldots,t_n$, and an arbitrary query term q, q subsumes $t_1 \circ \ldots \circ t_n(q_1)$.*

The transitivity of the subsumption relationship is immediate from its definition (Definition 28) which is based on the subset relationship, which is itself transitive.

As mentioned above, the substitution of a variable X in a negative context of a query term q by a query term t, which is not a variable, results in a query term $q' := q[X \mapsto t]$ which is in fact more general than q. In other words $q[X \mapsto t]$ subsumes q for any query term q if X only appears within a negative context in q. On the other hand, if X only appears in a positive context within q, then q' is less general – i.e. q subsumes q'. But what about the case of X appearing both in a positive and a negative context within q? Consider the following example:

Example 2. Let q := `a{{ var X, without b{{ var X }} }}`. It may be tempting to think that substituting X by `c[]` to give q' makes the first subterm of q less general, but the second subterm of q more general. In fact, a subterm `b[ c ]` within a data term would cause the subterm `without b{{ var X }}` of q to fail, but the respective subterm of q' to succeed, suggesting that there is a data term that simulation unifies with q', but not with q, meaning that q does not subsume q'. However, there is no such data term, which is due to the fact that the second occurrence of `X` within q is only a consuming occurrence. When this part of the query term is evaluated, the variable `X` is already bound.

The normalized form for Xcerpt query terms is introduced, because for an unnormalized query term q_1 that subsumes a query term q_2 one cannot guarantee that there is a sequence of subsumption monotone query term transformations $t_1, \ldots, t_n$ such that $t_n \circ \ldots \circ t_1(q_1) = q_2$. To see this, consider example 3.

Example 3 (Impossibility of transforming an unnormalized query term). Consider $q_1 := a\{\{var\ X\ as\ b\{\{c\}\}, var\ X\ as\ b\{\{d\}\}\}\}$ and $q_2 := a\{\{b\{\{c,d\}\}, b\{\{c,d\}\}\}\}$. q_2 subsumes q_1, in fact both terms are even simulation equivalent. But there is no sequence of subsumption monotone query term transformations from q_2 to q_1, since one would have to omit one subterm from both the first subterm of q_2 and from the second one. But such a transformation would in general not be subsumption monotone.

Besides opening up the possibility of specifying restrictions on one subterm non-locally, duplicate restrictions for the same variable also allow the formulation of *unsatisfiable* query terms, as the following example shows:

Example 4 (Unsatisfiable query terms due to variable restrictions). Consider the query terms $q_1 := a\{\{var\ X\ as\ b,\ var\ X\ as\ c\}\}$ and $q_2 := b\{\{\}\}$. It is easy to see that q_1 is unsatisfiable, and thus q_2 subsumes q_1. However, there is no transformation sequence from q_2 to q_1.

Also single variable restrictions may in some cases be problematic, because they allow the specification of infinite, or at least graph structured data terms as example 5 shows:

Example 5 (Nested variable restrictions). Consider the query terms $q_1 := a\{\{var\ X\ as\ b\{\{var\ X\}\}\}\}$ and $q_2 := a\{\{var\ Y\ as\ b\{\{b\{\{var\ Y\}\}\}\}\}\}$.

To overcome this issue, query terms are assumed to be in normalized form (Definition 30). In fact, almost all Xcerpt query terms can be transformed into normalized form.

Definition 30 (Query terms in normalized form). *A query term containing only a single variable restriction for each variable is a query term in* normalized form. *A query term which can be converted into an equivalent query term in normalized form is said to be* normalizable.

Unsatisfiability of query terms makes the decision procedure for subsumption more complex, and thus it is to be avoided whenever possible. Allowing the specification of unsatisfiable query terms does not add expressive power to a query language, and should thus be disallowed. Apart from the normal form, also *subterm injectivity* is a means for preventing the user of the Xcerpt query language from specifying unsatisfiable queries.

Example 6 (Unsatisfiability due to non-injectivity). In this example we use triple curly braces to state that the mapping from the siblings enclosed within the braces need not be injective. With this notation queries become less restrictive as the number of braces in the subterm specification increases. Let $q_1 := a\{\{\{b,\ \mathit{without}\ b\}\}\}$. Since q_1 both requires and forbids the presence of a subterm with label b, it is clearly unsatisfiable. Let $q_2 := b\{\{\}\}$. Although q_2 subsumes q_1, we cannot find a subsumption monotone transformation sequence from q_2 to q_1.

The above example shows that the the proof for the decidability of the subsumption relationship given in this section relies on the injectivity of the subterm mapping. Since there is no injectivity requirement for multiple consecutive predicates in XPath, the proof cannot be trivially used to show decidability of subsumption of XPath fragments.

Completeness

Theorem 6 (Subsumption by transformation). *Let q_1 and q_2 be two query terms in normalized form such that q_1 subsumes q_2. Then q_1 can be transformed into q_2 by a sequence of subsumption monotone query term transformations listed in Definition 29.*

Proof. We distinguish two cases:

- q_1 and q_2 are subsumption equivalent (i.e. they subsume each other)
- q_1 strictly subsumes q_2

The first case is the easier one. If q_1 and q_2 are subsumption equivalent, then there is no data term t, such that t simulates with one, but not the other. Hence q_1 and q_2 are merely syntactical variants of each other. Then q_1 can be transformed into q_2 by consistent renaming of variables (Equation 1.10), and by reordering sibling terms within subterms of q (Equation 1.7). This would not be true for unnormalized query terms as Example 3 shows.

The second is shown by structural induction on q_1.

For both the induction base and the induction step, we assume that q_1 subsumes q_2, but that the inverse is false. Then there is a data term d, such that q_1 simulates into d, but q_2 does not. In both the induction base and the induction step, we give a distinction of cases, enumerating all possible reasons for q_1 simulating into d but q_2 not. For each of these cases, a sequence of subsumption monotone transformations $t_1, \ldots t_n$ from Definition 29 is given, such that $q_1' := t_n \circ t_{n-1} \circ \ldots \circ t_1(q_1)$ does *not* simulate into d. By Lemmas 2 and 3, q_1' still subsumes q_2. Hence by considering d and by applying the transformations, q_1 is brought “closer” to q_2. If q_1' is still more general than q_2, then one more dataterm d' can be found that simulates with q_1', but not with q_2, and another sequence of transformations to be applied can be deduced from this theorem. This process can be repeated until q_1 has been transformed into a simulation equivalent version of q_2. For the proof, see the appendix of [35].

Decidability and Complexity In the previous section, we establish that, for each pair of query terms q_1, q_2 such that q_1 subsumes q_2, there is a (possibly infinite) sequence of transformations $t_1, \ldots, t_k$ by one of the rules in Section 1.14.3 such that $t_k \circ \ldots \circ t_1(q) = q_2$.

However, if we reconsider the proof of Theorem 6, it is quite obvious that the sequence of transformations can in fact not be infinite: Intuitively, we transform at each step in the proof q_1 further towards q_2, guided by a data term that simulates in q_1 but not in q_2. In fact, the length of a transformation sequence is bounded by the sum of the sizes of the two query terms. As size of a query term we consider the total number of its subterms.

Proposition 5 (Length of Transformation Sequences). *Let q_1 and q_2 be two Xcerpt query terms such that q_1 subsumes q_2 and n the sum of the sizes of q_1 and q_2. Then, there is a sequence of transformations $t_1, \ldots, t_k$ such that $t_k \circ \ldots \circ t_1(q_1) = q_2$ and $k \in O(n)$.*

Proof. We show that the sequences of transformations created by the proof of Theorem 6 can be bounded by $O(n+m)$ if computed in a specific way: We maintain a mapping μ from subterms of q_1 to subterms of q_2 indicating how the query terms are mapped. μ is initialized with (q_1, q_2). In the following, we call a data term d *discriminating* between q_1 and q_2 if q_1 simulates in d but not q_2.

(1) For each pair (q, q') in μ, we first choose a discriminating data term that matches case 1 in the proof of Theorem 6. If there is such a data term, we apply Equation (1.11), label replacement, once to q obtaining $t(q)$ and update the pair in μ by $(t(q), q')$. This step is performed at most once for each pair as $(t(q), q')$ have the same label and thus there is no more discriminating data term that matches case 1.

(2) Otherwise, we next choose a discriminating data term that matches case 2.a.i or 2.b.i. In both cases, we apply Equation (1.6), variable insertion, to insert a new variable and update the pair in μ. This step is performed at most $|q_2| - |q_1| \leq n$ times for each pair.

(3) Otherwise, we next choose a discriminating data term that matches case 2.a.ii and apply Equation (1.4), complete term specification and update the pair in μ. This step is performed at most once for each pair.

(4) Finally, the only type of discriminating data term that remains is one with the same number of positive child terms as q_2. We use an oracle to guess the right mapping σ from child terms of q_1 to child terms of q_2. Then we remove the pair from μ and add $(c, \sigma(c))$ to μ for each child term of q_1. This step is performed at most once for each pair in μ.

Since query subterms have a single parent, we add each subterm only once to μ in a pair. Except for case 2, we perform only a constant number of transformations to each pair. Case 2 allows up to n transformations for a single pair, but the total number of transformations (over all pairs) due to case 2 is bound by the size of q_2. Thus in total we perform at most $4 \cdot n$ transformations where n is the sum of the number of the sizes of q_1 and q_2.

Though we have established that the length of a transformation sequence is bound by $O(n)$, we also have to consider how to *find* such a transformation sequence. The proof of Proposition 5, already spells out an algorithm for finding such transformation sequences. However, it uses an oracle to guess the right mapping between child terms of two terms that are to be transformed. A naive deterministic algorithm needs to consider all possible such mappings whose number is bound by $O(n!)$. It is worth noting, however, that in most cases the actual number of such mappings is much smaller as most query terms have fairly low breadth and the possible mappings between their child terms are severely reduced just by considering only mappings where the labels of child terms simulate. However, in the worst case the $O(n!)$ complexity for finding the right mapping may be reached and thus we obtain:

Theorem 7 (Complexity of Subsumption by Rewriting). *Let q_1 and q_2 be two Xcerpt query terms. Then we can test whether q_1 subsumes q_2 in $O(n!^n)$ time.*

Proof. By proposition 5 we can find a $O(n)$ length transformation sequence in $O(n!^n)$ time and by Theorem 6 q_1 subsumes q_2 if and only if there is such a sequence.

1.14.5 Future Work in the area of Xcerpt Query Term Subsumption

Starting out from the problem of improving termination of logic programming based on rich kinds of simulation such as simulation unification, this section investigates the problem of deciding simulation subsumption between query terms. A rewriting system consisting of subsumption monotone query term transformations is introduced and shown to be sound and complete. By convenient pruning of the search tree defined by this rewriting system, the decidability of simulation subsumption is proven, and an upper bound for its complexity is identified.

Future work includes (a) a proof-of-concept implementation of the rewriting system, (b) the development of heuristics and their incorporation into the prototype to ensure fast termination of the algorithm in the cases when it is possible, (c) the study of the complexity of the problem in absence of subterm negation, descendant constructs, deep-equal, and/or injectivity, (d) the implementation of a backward chaining algorithm with tabling, which uses subsumption checking to avoid redundant computations and infinite branches in the resolution tree, and (e) the adaptation of the rewriting system to XPath in order to decide subsumption and to derive complexity results for the subsumption problem between XPath queries.

1.15 Conclusion

The Merriam-Webster dictionary defines versatile as "embracing a variety of subjects, fields or skills", as "turning with ease from one thing to another", and as "having many uses or applications". As shown in this chapter, the query language Xcerpt embraces both tree and graph-shaped data (in particular also relational data), Web and Semantic Web data, semantic data embedded in HTML as microformats and purely semantic data. It can be used to query data on a syntactic and on a semantic level, and it turns easily between the formats that it supports, allowing the transformation of one format to another within a single rule.

Having isolated the concept of simulation unification as a matching algorithm that can be adapted to any kind of semi-structured data, the single rule and multi-rule semantics of Xcerpt become versatile in the sense that new forms of simulations for new Web formats (e.g., topic maps) can be "plugged into" Xcerpt without having to adapt the semantics of single rules, and the semantics of negation as failure of possibly recursive multi-rule programs.

Datalog with negation and value invention can be used to precisely formulate the semantics of Xcerpt rules, no matter of the type of data being queried. Since it is a well-studied fragment of first order logic this provides an easy-to-understand semantics of Xcerpt for query authors that have some background knowledge in rule based formalisms. In particular, we use this translation for proving a number of computational properties of Xcerpt and some of its sub-languages.

Furthermore, the translation to Datalog with negation and value invention can serve as a basis for an implementation of Xcerpt in a relational database. For this aim, we also need a compact and efficient representation of both tree- and graph-shaped semi-structured data. Such a representation is discussed in Section 1.13. We showed that this representation allows constant time and constant per-node space reachability and adjacency test for all trees and many graphs.

While this article aims at giving an answer to the design questions for versatile web query languages, it has also raised a number of new questions and desires:

– Section 1.3 shows that Xcerpt is suitable for querying HTML, XML, RDF and microformat data. Xcerpt queries for extracting data from microformats, however, exhibit all the same underlying characteristics: excessive use of the *descendant* axis, ignoring XML element labels by using regular expressions, filtering elements according to the value of the `class` attribute. While in regular XML querying, the child axis is often more prevalent than the attribute axis, and element labels are more distinguishing than attribute values (except for `id`-attributes), these pairs have switched roles in microformat querying. With microformats becoming the de facto standard of the "lowercase semantic web" [96], query patterns specifically aimed at micro-formats and sharing the same characteristics as simulation are a valuable investigation. Alternatively a domain specific language for microformats only could be of use for Semantic Web programmers.
– As mentioned in Section 1.7, the idea of weak stratification could be carried over to rule based languages with a rich unficication algorithm such as simulation unification.
– Guaranteeing termination of backward chaining evaluation of possibly recursive multi-rule programs involving negation has received a large amount of attention in the past [143, 132] [129]. Termination is even a bigger issue for recursive rule based languages with a rich unification algorithm, since there is a larger variety of infinite branches of subsumption monotone subgoals. A subsumption-aware resolution algorithm for rule based languages with rich unfication and negation as failure is currently being implemented by the authors.

References

1. Abiteboul, S., Buneman, P., Suciu, D.: Data on the Web: from relations to semistructured data and XML. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2000)
2. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley Publishing Co., Boston, MA, USA (1995)
3. Abiteboul, S., Quass, D., McHugh, J., Widom, J., Wienerm, J.L.: The Lorel Query Language for Semistructured Data. Intl. Journal on Digital Libraries **1**(1) (1997) 68–88
4. Adida, B.: hGRDDL: Bridging microformats and RDFa. J. Web Sem. **6**(1) (2008) 54–60
5. Adida, B., Birbeck, M.: RDFa primer 1.0 embedding RDF in XHTML. W3c working draft, W3C (October 2007)
6. 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
7. Akhtar, W., Kopecky, J., Krennwallner, T., Polleres, A.: XSPARQL: Traveling between the XML and RDF worlds – and avoiding the XSLT pilgrimage. In Hauswirth, M., Koubarakis, M., Bechhofer, S., eds.: Proceedings of the 5th European Semantic Web Conference. LNCS, Berlin, Heidelberg, Springer Verlag (June 2008)
8. 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
9. Apple Inc.: plist — Property List Format. (2003)
10. Apt, K.R., Bol, R.N.: Logic programming and negation: A survey. J. Log. Program. **19/20** (1994) 9–71
11. Assmann, U., Berger, S., Bry, F., Furche, T., Henriksson, J., Johannes, J.: Modular web queries — from rules to stores. In: On the Move to Meaningful Internet Systems 2007: OTM 2007 Workshops, Proceedings of International Workshop on Scalable Semantic Web Knowledge Base Systems, Vilamoura, Algarve, Portugal (25th–30th November 2007). Volume 4805/2007 of LCNS. (2007)
12. Augurusa, E., Braga, D., Campi, A., Ceri, S.: Design and implementation of a graphical interface to XQuery. In: SAC '03: Proceedings of the 2003 ACM symposium on Applied computing, New York, NY, USA, ACM (2003) 1163–1167
13. Backett, D.: Turtle—Terse RDF Triple Language. Technical report, Institute for Learning and Research Technology, University of Bristol (2007)
14. Bailey, J., Bry, F., Furche, T., Schaffert, S.: Web and semantic web query languages: A survey. In: Reasoning Web, First International Summer School 2005. Volume 3564 of LNCS. Springer-Verlag (2005)
15. Beckett, D., McBride, B.: RDF/XML Syntax Specification (Revised). Recommendation, W3C (2004)
16. Benedikt, M., Koch, C.: Xpath leashed. ACM Computing Surveys (2007)
17. Berger, S., Bry, F., Bolzer, O., Furche, T., Schaffert, S., Wieser, C.: Xcerpt and visxcerpt: Twin query languages for the semantic web. In: Proceedings of 3rd International Semantic Web Conference, Hiroshima, Japan (7th–11th November 2004). LNCS (2004)
18. Björklund, H., Martens, W., Schwentick, T.: Conjunctive query containment over trees. In Arenas, M., Schwartzbach, M.I., eds.: DBPL. Volume 4797 of Lecture Notes in Computer Science., Springer (2007) 66–80
19. Boag, S., Berglund, A., Chamberlin, D., Siméon, J., Kay, M., Robie, J., Fernández, M.F.: XML path language (XPath) 2.0. W3C recommendation, W3C (January 2007) http://www.w3.org/TR/2007/REC-xpath20-20070123/.

20. Bolzer, O.: Towards Data-Integration on the Semantic Web: Querying RDF with Xcerpt. Diplomarbeit/diploma thesis, University of Munich (2005)
21. Bolzer, O.: Towards data-integration on the semantic web: Querying RDF with Xcerpt. Diplomarbeit/diploma thesis, Institute of Computer Science, LMU, Munich (2005)
22. 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
23. Bonifati, A., Ceri, S.: Comparative analysis of five xml query languages. SIGMOD Rec. **29**(1) (2000) 68–79
24. 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
25. Bray, T., Hollander, D., Layman, A., Tobin, R.: Namespaces in XML 1.0 (second edition) (2006) W3C Rec. 16 August 2006.
26. Bray, T., Hollander, D., Layman, A., Tobin, R.: Namespaces in XML (2nd Edition). Recommendation, W3C (2006)
27. Bray, T., Paoli, J., Sperberg-McQueen, C.M., Maler, E., Yergeau, François: Extensible markup language (xml) 1.0 (fourth edition) (2006)
28. Bray, T., Paoli, J., Sperberg-McQueen, C.M., Maler, E., Yergeau, F.: Extensible Markup Language (XML) 1.0 (Third Edition). Recommendation, W3C (2004)
29. Broekstra, Kampman, Harmelen: Sesame: A generic architecture for storing and querying RDF and RDF Schema. (2003)
30. 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
31. Bry, F., Eisinger, N., Eiter, T., Furche, T., Gottlob, G., Ley, C., Linse, B., Pichler, R., Wei, F.: Foundations of rule-based query answering. In Antoniou, G., Aßmann, U., Baroglio, C., Decker, S., Henze, N., Patranjan, P.L., Tolksdorf, R., eds.: Reasoning Web. Volume 4636 of Lecture Notes in Computer Science., Springer (2007) 1–153
32. Bry, F., Furche, T., Badea, L., Koch, C., Schaffert, S., Berger, S.: Querying the web reconsidered: Design principles for versatile web query languages. Journal of Semantic Web and Information Systems (IJSWIS) **1**(2) (2005)
33. Bry, F., Furche, T., Ley, C., Linse, B.: RDFLog—taming existence - a logic-based query language for RDF (2007)
34. Bry, F., Furche, T., Ley, C., Linse, B., Marnette, B.: RDFLog: It's like datalog for RDF. In: Proceedings of 22nd Workshop on (Constraint) Logic Programming, Dresden (30th September–1st October 2008). (2008)
35. Bry, F., Furche, T., Linse, B.: Simulation subsumption or déjà vu on the web (extended version). Technical Report PMS-FB-2008-01, University of Munich (2007)
36. Bry, F., Furche, T., Linse, B., Schroeder, A.: Efficient Evaluation of n-ary Conjunctive Queries over Trees and Graphs. In: Proc. ACM Int'l. Workshop on Web Information and Data Management (WIDM), ACM Press (2006).
37. Bry, F., Coskun, F., Durmaz, S., Furche, T., Olteanu, D., Spannagel, M.: The XML Stream Query Processor SPEX. In: Proc. Int'l. Conf. on Data Engineering (ICDE). (2005) 1120–1121.
38. Bry, F., Furche, T., Ley, C., Linse, B.: Rdflog: Filling in the blanks in rdf querying. Technical Report PMS-FB-2008-01, University of Munich (2007)
39. Bry, F., Furche, T., Ley, C., Linse, B., Marnette, B.: Taming existence in rdf querying. In: Proc. Int'l. Conf. on Web Reasoning and Rule Systems (RR). (2008)

40. Bry, F., Schaffert, S.: A Gentle Introduction into Xcerpt, a Rule-based Query and Transformation Language for XML. In: Proc. Intl. Workshop on Rule Markup Languages for Business Rules on the Semantic Web. (2002)
41. Buneman, P., Fernandez, M.F., Suciu, D.: UnQL: a query language and algebra for semistructured data based on structural recursion. VLDB Journal: Very Large Data Bases **9**(1) (???? 2000) 76–110
42. Bussche, J.V.D., Gucht, D.V., Andries, M., Gyssens, M.: On the completeness of object-creating database transformation languages. Journal of the ACM **44**(2) (1997) 272–319
43. Cabibbo, L.: The expressive power of stratified logic programs with value invention. Information and Computation **147**(1) (1998) 22–56
44. Carlos, J., Polleres, A., Polleres, A.: Sparql rules. Technical report, Universidad Rey Juan Carlos (2006)
45. Carroll, J.J., Bizer, C., Hayes, P., Stickler, P.: Named graphs, provenance and trust. In: WWW '05: Proceedings of the 14th international conference on World Wide Web, New York, NY, USA, ACM (2005) 613–622
46. Ceri, S., Comai, S., Damiani, E., Fraternali, P., Paraboschi, S., Tanca, L.: XML-GL: a graphical language for querying and restructuring XML documents. (1998)
47. Chamberlin, D.D., Robie, J., Florescu, D.: Quilt: An XML query language for heterogeneous data sources. In Suciu, D., Vossen, G., eds.: WebDB (Selected Papers). Volume 1997 of Lecture Notes in Computer Science., Springer (2000) 1–25
48. 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
49. 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
50. Chen, W., Warren, D.S.: Tabled evaluation with delaying for general logic programs. J. ACM **43**(1) (1996) 20–74
51. 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
52. Cholak, P., Blair, H.A.: The complexity of local stratification. Fundam. Inform. **21**(4) (1994) 333–344
53. 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
54. Connolly, D.: Gleaning resource descriptions from dialects of languages (grddl). Recommendation, W3C (2007)
55. Cooper, B., Sample, N., Franklin, M.J., Hjaltason, G.R., Shadmon, M.: A Fast Index for Semistructured Data. In: Proc. Int. Conf. on Very Large Databases, San Francisco, CA, USA, Morgan Kaufmann Publishers Inc. (2001) 341–350
56. Cowan, J., Tobin, R.: XML Information Set (2nd Ed.). Recommendation, W3C (2004)
57. Davis, I.: GRDDL primer (2006)
58. Deutsch, A., Fernández, M.F., Florescu, D., Levy, A.Y., Suciu, D.: XML-QL. In: QL. (1998)
59. Dijkstra, E.W.: On the role of scientific thought (EWD447). In: Selected Writings on Computing: A Personal Perspective. (1982) 60–66
60. Droop, M., Flarer, M., Groppe, J., Groppe, S., Linnemann, V., Pinggera, J., Santner, F., Schier, M., Schöpf, F., Staffler, H., Zugal, S.: Translating xpath queries into sparql queries. In: On the Move (OTM 2007) Federated Conferences and Workshops (DOA, ODBASE, CoopIS, GADA, IS), 6th International Conference on Ontologies, DataBases, and Applications of Semantics (ODBASE 2007). (2007) 9–10

61. Eiter, T., Faber, W., Koch, C., Leone, N., Pfeifer, G.: DLV - a system for declarative problem solving. In: Proceedings of the 8th International Workshop on Non-Monotonic Reasoning (NMR'2000). (2000)
62. Euzenat, J., Valtchev, P.: An integrative proximity measure for ontology alignment. In Doan, A., Halevy, A., Noy, N., eds.: Proceedings of the 1st Intl. Workshop on Semantic Integration. Volume 82 of CEUR. (2003)
63. Euzenat, J., Valtchev, P.: Similarity-based ontology alignment in OWL-lite. In de Mántaras, R.L., Saitta, L., eds.: Proceedings of the 16th European Conference on Artificial Intelligence (ECAI-04), IOS Press (2004) 333–337
64. Fagin, R.: Multivalued dependencies and a new normal form for relational databases. ACM Transactions on Database Systems **2**(3) (1977) 262–278
65. Fallside, D.C., Walmsley, P.: XML Schema Part 0: Primer Second Edition. Recommendation, W3C (2004)
66. Fernández, M., Malhotra, A., Marsh, J., Nagy, M., Walsh, N.: XQuery 1.0 and XPath 2.0 Data Model. Recommendation, W3C (2007)
67. Fulkerson, D.R., Gross, O.A.: Incidence Matrices and Interval Graphs. Pacific Journal of Mathematics **15**(3) (1965) 835–855
68. 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)
69. Furche, T., Linse, B., Bry, F., Plexousakis, D., Gottlob, G.: RDF querying: Language constructs and evaluation methods compared. In: Reasoning Web, Second International Summer School 2006. Volume 4126 of LNCS. (2006)
70. Furche, T., Weinzierl, A., Bry, F.: Scalable, space-optimal implementation of xcerpt single rule programs—part 1: Data model, queries, and translation. Deliverable I4-D15a, REWERSE (2007)
71. Gandon, F.: GRDDL use cases: Scenarios of extracting RDF data from XML documents. W3c working group note 6 april 2007, W3C (2007)
72. Garcia-Molina, H., Ullman, J.D., Widom, J.: Database Systems: The Complete Book. Prentice Hall (2002)
73. Garshol, L.M., Moore, G.: ISO 13250-2: Topic Maps — Data Model. International standard, ISO/IEC (2006)
74. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: Proceeding of the Fifth Logic Programming Symposium. (1988) 1070–1080
75. Goldman, R., Widom, J.: Dataguides: Enabling query formulation and optimization in semistructured databases. In: Proc. Int'l. Conf. on Very Large Data Bases (VLDB), San Francisco, CA, USA, Morgan Kaufmann Publishers Inc. (1997) 436–445
76. Gottlob, G., Koch, C., Pichler, R.: Efficient Algorithms for Processing XPath Queries. ACM Transactions on Database Systems (2005)
77. Gottlob, G., Leone, N., Scarcello, F.: The Complexity of Acyclic Conjunctive Queries. Journal of the ACM **48**(3) (2001) 431–498
78. Groppe, S., Groppe, J., Linnemann, V., Kukulenz, D., Hoeller, N., Reinke, C.: Embedding sparql into xquery/xslt. In: SAC '08: Proceedings of the 2008 ACM symposium on Applied computing, New York, NY, USA, ACM (2008) 2271–2278
79. Grust, T.: Accelerating XPath Location Steps. In: Proc. ACM Symp. on Management of Data (SIGMOD). (2002)
80. Grust, T., Keulen, M.V., Teubner, J.: Accelerating XPath Evaluation in any RDBMS. ACM Transactions on Database Systems **29**(1) (2004) 91–131
81. Grust, T., Teubner, J.: Relational Algebra: Mother Tongue - XQuery: Fluent. In: Proc. Twente Data Management Workshop on XML Databases and Information Retrieval. (2004)

82. 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)
83. 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
84. Hayes, P., McBride, B.: Rdf semantics. Recommendation, W3C (2004)
85. Henzinger, M.R., Henzinger, T.A., Kopke, P.W.: Computing simulations on finite and infinite graphs. In: FOCS. (1995) 453–462
86. Hsu, W.L.: PC-Trees vs. PQ-Trees. In: Proc. Int'l. Conf. on Computing and Combinatorics. Volume 2108 of LNCS. (2001)
87. Hsu, W.L.: A Simple Test for the Consecutive Ones Property. Journal of Algorithms **43**(1) (2002) 1–16
88. Hull, R., Yoshikawa, M.: Ilog: Declarative creation and manipulation of object identifiers. In: Proc. Int'l. Conf. on Very Large Data Bases (VLDB), San Francisco, CA, USA, Morgan Kaufmann Publishers Inc. (1990) 455–468
89. Jagadish, H.V., Lakshmanan, L.V.S., Srivastava, D., Thompson, K.: TAX: A Tree Algebra for XML. In: Proc. Int. Workshop on Database Programming Languages. (2001)
90. Jenner, B., Köbler, J., McKenzie, P., Torán, J.: Completeness results for graph isomorphism. Journal of Computer and System Sciences **66**(3) (2003) 549–566
91. Karvounarakis, G., Magkanaraki, A., Alexaki, S., Christophides, V., Plexousakis, D., Scholl, M., Tolle, K.: RQL: A functional query language for RDF. In Gray, P.M.D., Kerschberg, L., King, P.J.H., Poulovassilis, A., eds.: The Functional Approach to Data Management: Modelling, Analyzing and Integrating Heterogeneous Data. LNCS, Springer-Verlag (2004) 435–465
92. Kay, M.: Parsing in functional unification grammar. In Dowty, D., Karttunen, L., Zwicky, A., eds.: Natural Language Parsing: Psychological, Computational, and Theoretical Perspectives. Cambridge University Press, Cambridge (1985) 251–278
93. Kay, M.: Functional unification grammar: A formalism for machine translation. In: COLING-84, Stanford, CA (1984) 75–78
94. Kay, M.: XSL Transformations, Version 2.0. Recommendation, W3C (2007)
95. Kay, M.: XSL transformations (XSLT) version 2.0. W3C recommendation, W3C (January 2007) http://www.w3.org/TR/2007/REC-xslt20-20070123/.
96. Khare, R.: Microformats: The next (small) thing on the semantic web? IEEE Internet Computing **10**(1) (2006) 68–75
97. Khare, R., Çelik, T.: Microformats: a pragmatic path to the semantic web. In: WWW '06: Proceedings of the 15th international conference on World Wide Web, New York, NY, USA, ACM Press (2006) 865–866
98. Klaas, V.: Who's who in the world wide web: Approaches to name disambiguation. Diplomarbeit/diploma thesis, Institute of Computer Science, LMU, Munich (2007)
99. Klug, A.C.: On conjunctive queries containing inequalities. J. ACM **35**(1) (1988) 146–160
100. Klyne, G., Carroll, J.J., McBride, B.: Resource Description Framework (RDF): Concepts and Abstract Syntax. Recommendation, W3C (2004)
101. Knoblock, C.A., Minton, S., Ambite, J.L., Ashish, N., Modi, P.J., Muslea, I., Philpot, A., Tejada, S.: Modeling web sources for information integration. In: AAAI/IAAI. (1998) 211–218
102. Koch, C.: On the Complexity of Nonrecursive XQuery and Functional Query Languages on Complex Values. ACM Transactions on Database Systems **31**(4) (2006)
103. Kochut, K., Janik, M.: SPARQLeR: Extended SPARQL for semantic association discovery. In Franconi, E., Kifer, M., May, W., eds.: ESWC. Volume 4519 of Lecture Notes in Computer Science., Springer (2007) 145–159

104. Kolaitis, P.G., Papadimitriou, C.H.: Why not negation by fixpoint? In: PODS, ACM (1988) 231–239
105. Lenzerini, M.: Data integration: A theoretical perspective. (2002)
106. Manola, F., Miller, E.: RDF primer, W3C recommendation. Technical report, W3C (2004)
107. Manola, F., Miller, E., McBride, B.: Rdf primer. Recommendation, W3C (2004)
108. Marsh, J.: XML Base. Recommendation, W3C (2001)
109. Martínez, J.M.: Mpeg-7 overview. Technical Report ISO/IEC JTC1/SC29/WG11N6828, INTERNATIONAL ORGANISATION FOR STANDARDISATION (ISO) (2004)
110. Marx, M.: Conditional XPath, the first order complete XPath dialect. In: Proceedings of the twenty-third ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems, ACM New York, NY, USA (2004) 13–22
111. Marx, M.: Conditional XPath. ACM Transactions on Database Systems (TODS) **30**(4) (2005) 929–959
112. McBride, B.: Rdf vocabulary description language 1.0: Rdf schema (2004)
113. 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
114. Meuss, H., Schulz, K.U., Bry, F.: Towards Aggregated Answers for Semistructured Data. In: Proc. Intl. Conf. on Database Theory, Springer-Verlag (2001) 346–360
115. Milner, R.: An algebraic definition of simulation between programs. In: IJCAI. (1971) 481–489
116. Noy, N.F., Musen, M.A.: PROMPT: Algorithm and tool for automated ontology merging and alignment. In: AAAI/IAAI. (2000) 450–455
117. Olteanu, D.: SPEX: Streamed and Progressive Evaluation of XPath. IEEE Transactions on Knowledge and Data Engineering (2007)
118. Olteanu, D., Furche, T., Bry, F.: Evaluating Complex Queries against XML streams with Polynomial Combined Complexity. In: Proc. British National Conf. on Databases (BN-COD). (2003) 31–44.
119. 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.
120. Olteanu, D., Meuss, H., Furche, T., Bry, F.: Xpath: Looking forward. In Chaudhri, A.B., Unland, R., Djeraba, C., Lindner, W., eds.: EDBT Workshops. Volume 2490 of Lecture Notes in Computer Science., Springer (2002) 109–127
121. 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
122. Pepper, S.: The TAO of topic maps. (2000)
123. Pérez, J., Arenas, M., Gutierrez, C.: Semantics and complexity of SPARQL. In Cruz, I.F., Decker, S., Allemang, D., Preist, C., Schwabe, D., Mika, P., Uschold, M., Aroyo, L., eds.: International Semantic Web Conference. Volume 4273 of Lecture Notes in Computer Science., Springer (2006) 30–43
124. Pérez, J., Arenas, M., Gutierrez, C.: nSPARQL: A navigational language for RDF. In Sheth, A.P., Staab, S., Dean, M., Paolucci, M., Maynard, D., Finin, T.W., Thirunarayan, K., eds.: International Semantic Web Conference. Volume 5318 of Lecture Notes in Computer Science., Springer (2008) 66–81
125. Pérez, J., Arenas, M., Gutierrez, C.: nsparql: A navigational language for rdf. In: Proc. Int'l. Semantic Web Conf. (ISWC). (2008) 66–81
126. Polleres, A.: From sparql to rules (and back). In Williamson, C.L., Zurko, M.E., Patel-Schneider, P.F., Shenoy, P.J., eds.: WWW, ACM (2007) 787–796

127. Polleres, A., Krennwallner, T., Kopecky, J., Akhtar, W.: Xsparql: Traveling between the xml and rdf worlds – and avoiding the xslt pilgrimage. In: Proc. European Semantic Web Conf. (ESWC). (2008)
128. Przymusinska, H., Przymunsinski, T.C.: Weakly stratified logic programs. Fundam. Inf. **13**(1) (1990) 51–65
129. Przymusinski, T.C.: On the declarative semantics of deductive databases and logic programs. In: Foundations of Deductive Databases and Logic Programming. Morgan Kaufmann (1988) 193–216
130. Recordon, D., Reed, D.: OpenID 2.0: a platform for user-centric identity management. In: DIM '06: Proceedings of the second ACM workshop on Digital identity management, New York, NY, USA, ACM (2006) 11–16
131. Ross, K.A.: Modular stratification and magic sets for DATALOG programs with negation. In: PODS, ACM Press (1990) 161–171
132. Sagonas, K.F., Swift, T., Warren, D.S.: The XSB programming system. In: Workshop on Programming with Logic Databases (Informal Proceedings), ILPS. (1993) 164
133. Schaffert, S.: Xcerpt: A Rule-Based Query and Transformation Language for the Web. PhD thesis, University of Munich (2004)
134. Schaffert, S.: Xcerpt: A Rule-Based Query and Transformation Language for the Web. Dissertation/doctoral thesis, University of Munich (2004)
135. Schenk, S., Staab, S.: Networked graphs: A declarative mechanism for sparql rules, sparql views and rdf data integration on the web. In: Proceedings of the 17th International World Wide Web Conference, Bejing, China (2008-04)
136. Schenkel, R., Theobald, A., Weikum, G.: HOPI: An Efficient Connection Index for Complex XML Document Collections. In: Proc. Extending Database Technology. (2004)
137. Schneider, P.P., Simeon, J.: The yin/yang web: Xml syntax and rdf semantics. Proceedings of the eleventh international conference on World Wide Web, ACM Press (2002) 11
138. Schwentick, T.: Xpath query containment. SIGMOD Record **33**(1) (2004) 101–109
139. Seaborne, A., Manjunath, G., Bizer, C., Breslin, J., Das, S., Davis, I., Harris, S., Idehen, K., Corby, O., Kjernsmo, K., Nowack, B.: SPARQL/Update A language for updating RDF graphs. W3C Member Submission, W3C (July 2008) `http://www.w3.org/Submission/2008/04/`.
140. Seaborne, A., Prud'hommeaux, E.: SPARQL query language for RDF. W3C recommendation, W3C (January 2008) http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/.
141. Siméon, J., Chamberlin, D., Florescu, D., Boag, S., Fernández, M.F., Robie, J.: XQuery 1.0: An XML query language. W3C recommendation, W3C (January 2007) http://www.w3.org/TR/2007/REC-xquery-20070123/.
142. Stickler, P.: Cbd - concise bounded description (2005)
143. Tamaki, H., Sato, T.: OLD resolution with tabulation. In Shapiro, E.Y., ed.: ICLP. Volume 225 of Lecture Notes in Computer Science., Springer (1986) 84–98
144. 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
145. Ullman, J.D.: Information integration using logical views. Theor. Comput. Sci. **239**(2) (2000) 189–210
146. van Gelder, A., Ross, K., Schlipf, J.: The well-founded semantics for general logic programs. Journal of the ACM **18** (1991) 620–650
147. van Gelder, A., Ross, K.A., Schlipf, J.S.: The well-founded semantics for general logic programs. Journal of the ACM (1991)
148. W3C: Gleaning resource descriptions from dialects of languages (GRDDL). W3c recommendation, W3C (September 2007)

149. Walsh, N., Muellner, L.: DocBook: The Definitive Guide. O?Reilly (1999)
150. 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
151. Wei, F., Lausen, G.: Containment of conjunctive queries with safe negation. In Calvanese, D., Lenzerini, M., Motwani, R., eds.: ICDT. Volume 2572 of Lecture Notes in Computer Science., Springer (2003) 343–357
152. 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