Logical Tree Matching with Complete Answer Aggregates for Retrieving Structured Documents

Dissertation

zur Erlangung des akademischen Grades des
Doktors der Naturwissenschaften
an der Fakultät für Mathematik und Informatik
der Ludwig-Maximilians-Universität München

von

Holger Meuss

Tag der Einreichung: 12.4.2000

Meiner Frau Ruth

Abstract

The use of markup languages like SGML or XML for encoding the structure of documents has lead to many document databases where entries are adequately described as trees. In this context query formalisms are interesting that offer the possibility to refer both to textual content and logical structure of structured documents. Many query formalisms model answers to such queries as mappings from query variables to nodes in the document database. Confronted with a set of such answers the user may have problems to understand the inner structure and possible dependencies in this set. Due to the unstructured presentation of the query result in the form of a set of answers the user may even fail to locate the relevant pieces of information. This deficiency of mapping-based query formalisms in general serves as the major motivation for this work.

This work takes as a starting point the Tree Matching formalism for retrieving structured documents developed by Pekka Kilpeläinen. This mapping-based formalism has a simple and abstract model and is yet powerful in its expressivity. The problem of answer presentation and two common phenomena causing a combinatorial explosion in the number of answers are illustrated with Tree Matching, leading to the result that there are $\mathcal{O}(n^q)$ answers to a Tree Matching query, where n (q) is the size of the database (query). The basic Tree Matching model is reformulated in a language of first order predicate logic and extended in order to provide more flexibilty in query formulation.

The first central contribution of this work is the introduction of a data structure called "complete answer aggregate" that is presented to the user instead of an unstructured list of all answers. Complete answer aggregates provide an intuitive and visual overview over the set of all answers with their inner dependencies. Shared parts of distinct answers are only represented once in order to (1) support the users in their understanding of the query result, (2) reduce the size of the presented result, and (3) compute a result efficiently. The size of a complete answer aggregate is of order $\mathcal{O}(n \cdot h \cdot q)$ where h is the maximal length of a path in the database.

The second main contribution describes an algorithm computing the complete answer aggregate for a query and database in time $\mathcal{O}(n \cdot \log(n) \cdot h \cdot q)$. For a substantial class of queries and databases this algorithm runs even in time $\mathcal{O}(n \cdot \log(n) \cdot q)$. The algorithm makes use of dedicated index structures exploiting the peculiarities of structured documents.

The third central contribution describes how users can exploit the added value of complete answer aggregates in the retrieval process. An iterative retrieval model in the flavor of retrieval models developed in Information Retrieval is presented. In this model users define a "sphere of interest" with an initial query. The corresponding complete answer aggregate is modified with various manipulation and exploration techniques to further explore the sphere of interest and achieve new knowledge. The visual nature nature of complete answer aggregates makes this retrieval model intuitive and easy to use.

Danksagung

Diese Arbeit hätte nicht fertiggestellt werden können ohne die Hilfe von vielen Freunden und Kollegen, denen an dieser Stelle mein Dank ausgesprochen sei. Ich möchte mich bei François Bry bedanken für die Betreuung dieser Arbeit sowie für Hilfe und Hinweise, die mir oft eine neue Sicht auf die Dinge ermöglichten. Seiner Vermittlung habe ich auch zahlreiche interessante Kontakte und Diskussionen mit anderen Forschern zu verdanken. Mein besonderer Dank gilt Klaus Schulz, der mir während der letzten drei Jahre immer sehr freundschaftlich und außerordentlich kompetent beistand. Er hat mir die Augen für interessante Aspekte in diesem Gebiet geöffnet und mir immer wieder neue Motivation und Anregungen gegeben, wenn die Arbeit steckengeblieben ist. Christian Strohmaier und Slim Abdennadher haben Teile dieser Arbeit gelesen und mir wertvolle Hinweise für die Endkorrektur gegeben. Mein Dank gilt auch der Verso-Gruppe in Roquencourt, Frankreich, und der Information-Retrieval-Gruppe in Dortmund, für ihre Einladungen und die interessanten Einblicke in ihre Forschungsarbeiten. Natürlich danke ich auch dem Graduiertenkolleg "Sprache, Information, Logik" und der DFG für die angenehme Zusammenarbeit und die großzügige finanzielle Unterstützung.

Meine Freunde und meine Familie halfen mir, indem sie mir das Vertrauen und die Stärke gaben, die ich brauchte, um diese Arbeit zu schreiben. Mein größter Dank aber gilt meinen Kindern David, Esther und Nora, für das schöne Leben, das wir teilen, und meiner Frau Ruth für all ihre Liebe.

Contents

Chapter 1

Introduction

The information overload has become proverbial in the last years. More and more information is accessible, on the World Wide Web (WWW) and through more traditional communication channels, but finding relevant information becomes increasingly difficult. In a Reuters study in 1998 ([Reu99]) more than 40% of the questioned managers accounted difficulties in retrieving relevant information to the problem of information overload.[1] This problem shows up in one of its worst manifestations in the WWW. In the tenth WWW user survey conducted by the GVU ([GVU99]) 70% of the respondents stated that, when searching for information in the WWW, most of the time they search for specific information, but more than 40% stated they have problems in finding the information they are looking for.[2] The urge to support users in finding information in the WWW led to the development of enriching Web documents with structural information about the content, e.g. the Dublin Core framework ([Wei99]). Another result of these efforts was XML, the prospective language of the WWW, allowing to structure Web and other documents in an arbitrary way and to enrich them with information about their content. Industrial and academic research is now confronted with the task of finding mechanisms of exploiting the richer structure of XML documents in order to support the user in finding documents in the WWW. In parallel, research on Database Systems (DBS) was stimulated by problems accompanying the advent of the WWW, and techniques that have been developed in the field of Information Retrieval (IR) received suddenly a famous rebirth in search engines for the WWW. In the moment we can observe a tendency of amalgamation in the different fields concerned with retrieving information, DBS, IR, and the WWW, and a fruitful exchange of ideas and techniques between these fields. All these developments nourished the research on the retrieval of structured documents, that touches the three fields concerned with retrieving information. But before giving a more detailed introduction into the concepts of structured documents and their retrieval, we will examine some developments in academic and industrial research that had an impact on the synergetic effects we can observe in DBS, IR, and the WWW.

[1] In the fifth annual Reuters report on the aspects and consequences of information more than 1000 business executives worldwide had been questioned on their views and experiences of handling information. On the question "What do you think constrains you or your colleagues from obtaining the information you require?" 60% answered "Don't know how to get it," 59% "Time constraints," and 42% "Information Overload".

[2] More than 5000 Web users participated in the tenth WWW user survey conducted by the Graphic, Visualization & Usability Center (GVU) of the Georgia Institute of Technology in late 1998. The question "To what extent would you say you use the Internet to search for specific information?" was answered by 70% of the respondents with "Most times" and by 30% with "sometimes". For the question "What do you find to be the biggest problems in using the Web?" the respondents could chose one more items from a list of points. 45% answered with "Not being able to find the information I'm looking for".

1.1 Finding the Needle in the Haystack

In **Database Systems** there has been a tendency for the last two decades towards developing specialized data models dedicated to specific application domains, as opposed to universal data models. Examples for this development can be observed in the application domains of geography, biology, text documents, multimedia, etc.

Although there is this strong tendency towards divergence of data models, there are, on the other hand, many research efforts aiming at data integration. They are motivated by the developments that distinct databases are increasingly used together and that more and more data is migrated from one database to another. The migration is restrained by possibly different schemas of the databases. The impact of new global communication and data exchange structures like the WWW make this problem even more urging. In this setting the semistructured data model (e.g. [ABS99]) was developed, that (1) relaxed the strict notion of schema established in the field of DBS, allowing for data with irregular, unknown or changing structure, and where (2) the data describes its structure itself by the means of metadata, giving the data a "meaning" even outside of a database environment with a defining schema. These features make the model perfectly apt as a data exchange formalism. The observation that the semistructured data model is an abstract model for XML ([W3C98a]), the prospective language of the WWW and global data exchange, brought the semistructured data model into the focus of research on DBS and data exchange.

In **Information Retrieval** we can observe more attention towards the internal structure of textual documents beyond the concept of words. Beginning with the division of documents into a fixed number of fields more and more models begin to pay attention to a more flexible model for the structure of textual documents (e.g. [Fuh96]). A similar research direction breaks the monolithic view upon text documents and investigates the retrieval of passages of text instead of whole documents (e.g. [SAB93]).

Finally, the **World Wide Web (WWW)** with its overwhelming success as global communication medium, with its omnipresence in science, industry, and private life, led to an enormous interest from industrial and academic research. IR technologies that had been of interest for only a limited group of persons ten years earlier have become a standard part of every search engine in the WWW and are used by millions of people daily. With the introduction of XML ([W3C98a]) as the prospective language of the WWW by the W3C (World Wide Web Consortium) the exchange of information via the WWW becomes even easier as with the state of the art language HTML. Due to the flexible approach of XML to model the structure and layout of text documents it is also be used outside the WWW as a data representation and exchange language, e.g. in the fields of astronomy and biology. Since XML is a formalism for structured documents and there is an enormous interest in querying information on the WWW, the research on the retrieval of structured documents gained a major interest in industrial and academic research, as can be seen from the manyfold activities of the W3C on XML query languages, for example the first W3C workshop QL'98 ([W3C98b]) on query languages for the WWW with over 60 international contributions.

In general we can observe a tendency towards convergence of the three fields as can be seen from the general directions mentioned above and the numerous work on, for example, database systems with IR facilities, Web interfaces to database systems, integration of Web pages into database systems, Web search engines using IR techniques, etc.

1.2 Structured Documents

Structured documents, e.g. XML or SGML documents, are textual documents that bear an explicit, internal, commonly hierarchic structure. This structure is represented by markup that appears in the documents in the form of tags. Tags delimit structural elements inside the documents. These elements can be nested and arranged in any form, apart from the fact that they cannot overlap.[3] In most cases a collection of related structured documents has a common structure that can be described by a grammar (which is less restrictive than a schema in the sense of relational DBS). This grammar (or DTD: document type definition) determines the order and possible inclusion of elements in the documents. In Appendix B.3 a collection of XML documents can be found that describe movies, the corresponding DTD is found in Appendix B.1. In SGML and XML syntax, which we adapt in this work, tags are embedded in "<" and ">", e.g. "`<MOVIE>`" for an opening tag and "`</MOVIE>`" for the corresponding closing tag. An element's content is between the opening and closing tag. The document on page 137 for example describes a movie. The description consists of a title, information about the staff, the production year and some information about the content. The staff for example contains the name of the director and the cast, that is, the actors. On page 132 this hierarchical structure is depicted as a tree. The leaves of the tree contain the actual textual content, whereas the inner nodes are the structuring elements that bear labels describing the "representational function" of the element, e.g. "`movie`" or "`actor`". More comprehensive introductions into SGML or XML can be found in [Gol90, vH94, ABS99, Oas].

There are various formalisms to describe structured documents, notably the ISO standards SGML (Standard Generalized Markup Language: [ISO86, Gol90, vH94, Oas]) and ODA (Office Document Architecture: [ISO89]), as well as XML (eXtensible Markup Language: [W3C98a, ABS99, Oas]) which is under development by the W3C (World Wide Web Consortium) and has currently the status of a recommendation. The abstract model underlying all formalisms is that of an ordered, labeled tree describing the hierarchic structure of the documents with the leaves of the tree containing the actual textual content. Since we are not concerned with the syntactic peculiarities of the various representation formalisms for structured documents but with the abstract data model we will not go into further details of the formalisms here.

1.3 Retrieving Structured Documents

The development of representation formalisms for structured documents led to big repositories of structured documents. The use of XML for the WWW will increase the size and number of these already existing repositories by far. Research on the retrieval of structured documents is a reaction to the urging issue of how to support the user in the task of finding relevant documents for his or her information need. Retrieval of structured documents is a field in the intersection of DBS and IR: The user can access both the structure of the documents (what is a DBS aspect) and the textual content (an IR aspect) at the same time. The affiliation of structured document retrieval with both fields is additionally strengthened by converging tendencies in these fields as elaborated in Section 1.1. Aspects from DBS include persistent storage, efficient index structures, query evaluation and optimization. From IR techniques like inverted files, linguistic normalization, keyword extraction, relevance ranking and relevance feedback touch the retrieval of structured documents. The

[3] In XML terminology structured documents whose elements do not overlap are called well-formed.

challenge of structured document retrieval, additionally stimulated by the impact of the WWW on the DBS and IR communities, is to amalgamate these techniques in an appropriate way with new techniques specific to structured documents.

Query Languages

The efforts in supporting the user in the retrieval of structured documents led to the development of various query languages. In the query language the user specifies a pattern for the structure that also contains keywords that are required to occur in specific elements. An answer returns elements (nodes) from the document collection that match the query. We can distinguish the main features of query languages with the following (coarse) classification. A more comprehensive overview over query languages known in the literature, together with pointers to the literature and survey papers, can be found in Section 8.3.

One-Dimensional Versus Multi-Dimensional Formalisms

One-dimensional formalisms return as query result an unstructured set of document nodes that match the query, whereas *multi-dimensional* formalisms allow the definition of variables in the query and return as result a set of answers, where each answer is a mapping from the variables in the query to matching document nodes.[4] Informally, we can say that multi-dimensional formalisms pursue a more database-oriented approach and one-dimensional formalisms are more IR-oriented. One-dimensional formalisms can be evaluated quite efficiently due to their simple notion of answers, but are restricted in expressiveness, as elaborated in more detail in Section 8.3. And the lack of complex answers has the great disadvantage that we can not form queries like "*Give me all author-title pairs of articles containing the keyword 'IR'*".

Graphical Versus Text-Based Query Languages

Most query languages require the user to submit the query in a textual form. Some formalisms use the abstract tree model underlying structured documents as a guideline for the design of their query language. The pattern that is being searched for can be formulated graphically with the advantage of being easier to understand and formulate. Text-based formalisms are easier to access with automatically generated queries. In some cases a graphical query language has a text-based counterpart, and vice versa.

Retrieval and Manipulation

All query languages for structured documents have a retrieval (selection) part that describes how to locate and specify relevant documents and nodes. Some formalisms with a more database-oriented approach have in addition a manipulation (construction) part that allows to generate new documents out of the query result. With the latter class it is possible to update a document collection using the query language, i.e. to modify, remove or add documents.

Example Applications

There is a rich field of applications for the retrieval of structured documents, including retrieval of technical manuals, of correspondence, of scientific data, etc.

[4] In some multi-dimensional formalisms these mappings are only a transient, internal state used to produce a complex result in the form of one or more new documents.

This section illustrates some application areas of structured document retrieval on the basis of example queries. In the rest of this text we use as running example a document collection based on the Internet Movie Database (IMDB, [IMD00]) that stores data about movies in a set of documents. The small example collection consisting of ten documents can be found an Appendix B.3, a graphical representation of these documents in Appendix B.2, and a DTD in Appendix B.1. We will now introduce three domains with typical tasks that have to be performed on collections of structured documents. The first example illustrates an industrial application, whereas the second shows the use of structured document retrieval in academic research. The third example is the standard example in the literature on structured document retrieval. Other examples illustrating application areas can be found e.g. in [W3C00b].

Legal Documents

Legal documents are in general highly structured into laws, court decisions, paragraphs, sentences, dates, courts, etc. There already exist huge collections of legal documents in SGML or XML format, mainly in publishing companies for the publication on paper, CD-ROM, or the WWW. Typical queries to collection of legal documents by a lawyer or judge, referring to both structure and textual content at the same time, include:

- *Give me all titles of court decisions together with the names of the corresponding judges, that have been passed in Karlsruhe, Germany, between 1974 and 1978.*
- *Give me the names of all judges whose verdicts (being on the subject of copyright law) have been quashed.*
- *Find me all paragraphs that are referenced in court decisions where the sum in dispute was higher than 10.000 EUR.*

Computational Linguistics

In the recent years there have been several projects that established extensive databases of linguistic data describing the structure of phrases (e.g. [MSM93, OMM98, SLG95, SBKU98, HPS98, KN98b]). These databases are available as tagged corpora where the tags represent the (hierarchic) syntactical structure, sometimes with additional semantic information. They are used as a means for verification of linguistic hypotheses and provide an empiric overview over structural variety of phrases, context-dependent uses and associations of words and phrases, and other features of natural language. An appropriate query language can support the researcher in making explicit information that is only implicitly available in the corpus. We list some example queries.

- *What are the semantic categories of words that can be object of the verb 'assist'? Are they different from the respective semantic categories of the verb 'support'?*
- *Which words and/or parts of speech qualify as modifiers for the word "heart disease"?*
- *Show me examples for the use of the adjective 'hot' in different semantic contexts.*

Scientific Articles

Scientific articles have in most cases a common structure: An article has a title and authors, a publishing location (e.g. name of journal or conference), an abstract, a body, an appendix, a bibliography, etc. The following queries can be of interest:

- *Give me all titles and authors of articles containing the keyword 'Information Retrieval' published at a SIGMOD conference.*
- *Give me the names of authors with at least three SIGIR publications together with the titles and years of publication of the respective articles.*
- *Give me the author names of articles whose title contain the same keywords as the article I am actually working on.*

1.4 The Tree Matching Formalism

In 1992 Pekka Kilpeläinen developed in his PhD thesis a retrieval formalism for structured documents called Tree Matching ([Kil92]). This is a very elegant and simple, graphical and multi-dimensional formalism for retrieving (but not manipulating) structured documents that is based on mappings between pattern trees and the document trees. An answer to a query and a document collection is a mapping that preserves certain hierarchical and precedence relations. Kilpeläinen defined distinct classes of Tree Matching problems differing in the relations that have to be preserved by answers. Due to its conceptual simplicity, Tree Matching can be seen as an abstract model of many other query formalisms for structured documents. Summing up, Tree Matching is a very expressive retrieval formalism whose graphical approach to query formulation is very appealing and impresses with its simplicity.

On the negative side, Kilpeläinen showed that the decision problem whether there exists an answer to a given query and database is NP-complete for one of the most interesting (for structured document retrieval) classes of Tree Matching problems. Another deficiency of Tree Matching can be observed in all other multi-dimensional query formalisms as well: Due to the tuple structure of answers combinatorial effects come into effect resulting in polynomially many answers in the size of the database (number of nodes in the document tree). Apart from the fact that this effect makes computation even for the tractable classes of Tree Matching problems inefficient, these answers often share common subparts, making it for the user difficult to locate the relevant information in the list of all answers.

Regarding the retrieval process and query formulation, the strict division into different problem classes proved to be too inflexible for real-world applications. In addition, there exists no elaborated retrieval model, as for example in the field of IR, that describes and supports the user's exploration process and may involve query reformulation based on a closer inspection of the answers.

1.5 Contribution of this Work

In this work we will reformulate and extend the Tree Matching formalism and introduce a new concept of query result that is presented to the user. We will extend Tree Matching in the directions of (1) flexibility and efficiency provided in DBS, (2) the retrieval model developed in IR, and (3) special techniques for structured documents.

With minor changes in the notion of answers the NP-completeness result of Tree Matching can be avoided. The logical reformulation breaks the strict division of

the distinct problem classes and thus provides an extension towards more flexibility. In addition, the logical formulation introduces generic relations modeling features of structured documents like left-to-right order of elements, or attributes. These relations can be queried, thus making the reformulation more expressive than the original formalism. In parallel with the logical reformulation we will maintain the graphical approach of the Tree Matching formalism, thus enjoying the benefits of text-based and graphical query languages at the same time.

As the central contribution of this work we will introduce a data structure called "complete answer aggregate" that avoids the negative effects of the combinatorial explosion in the number of answers and can be presented to the user as a substitute for the list of all answers. Complete answer aggregates represent all answers in an intuitive and graphical way, giving the user a good overview over the set of all answers and its internal topology. Dependencies between distinct answers and parts of answers that are implicit in a list of all answers are made explicit with complete answer aggregates. The benefits coming from avoiding the combinatorial explosion with complete answer aggregates can be exploited in various multi-dimensional formalisms for retrieving structured documents.

We show that complete answer aggregates can be computed very efficiently. For a realistic class of queries and document databases even in time quadratic in the size of the document database and linear in the size of the query. The algorithm computing the complete answer aggregate for a query and document database uses new index structures that are discussed and explained.

We show how the elaborated retrieval model developed in IR research can be adapted to the retrieval of structured documents. This adaption relies heavily on complete answer aggregates as basic manipulation objects. The retrieval model to be developed in this work allows visual and interactive exploration of the information space represented by the document collection and involves the user's feedback in query reformulation. All operations provided by this retrieval model are on a graphical basis.

The central contributions of this work, which are elaborated in Chapters 4, 5 and 6, respectively, are:

- The introduction of complete answer aggregates as a unique representation and substitute for the set of all answers,
- an algorithm computing the complete answer aggregate for a query and document database efficiently, supported by dedicated index structures, and
- a model for an iterative retrieval process that involves query reformulation and constitutes the graphical exploration of the information space represented by the collection of structured documents.

1.6 Overview

In the next chapter we will review Kilpeläinen's Tree Matching formalism and point out its strengths and weaknesses. The primary observation will be that Tree Matching, as any other multi-dimensional formalism, suffers from a combinatorial explosion in the number of answers. Different answers have many subparts in common what makes it very difficult for the user to distinguish relevant from irrelevant answers.

In Chapter 3 we will provide a logical reformulation that extends the expressivity of Tree Matching to hybrid queries, i.e. queries that allow different matching requirements in the same query, and queries with constraints. Examples for constraints

are given in Appendix A. A particularly interesting constraint is the horizontal left-to-right ordering of nodes in documents. Queries involving these constraints form the class of partially ordered tree queries. In addition we will show how the original graphical formulation of Tree Matching is compatible with the logical formulation so that we can still enjoy the advantages of having a graphical query language.

In the following chapter the central notion of complete answer aggregates will be introduced for simple and partially ordered tree queries. We show that a complete answer aggregate is a natural and unique representation for exactly the set of all answers to the query. It is natural since it represents the internal topology, with the sharing of subparts, of the set of all answers in an intuitive way. In addition, the complete answer aggregate for a query is substantially smaller than the set of all answers.

Chapter 5 presents an algorithm computing the complete answer aggregate for a query in time $\mathcal{O}(q \cdot n \cdot h \cdot \log(n))$ where q is the size (number of nodes/variables) of the query, n the size (number of nodes) of the document collection, and h the height of the forest representing the document collection. For a realistic class of queries the algorithm runs even in time $\mathcal{O}(q \cdot n \cdot h)$. This chapter also elaborates on index structures used and the implementation of the algorithm in Java.

In Chapter 6 we introduce a retrieval model with the flavour of the iterative retrieval model used in IR that involves feedback of the user and query reformulation. The retrieval model supports the user with a set of graphical operations in the process of incrementally exploring a "sphere of interest".

The following chapter mentions points that couldn't be treated in this work and may serve as future work. These include extensions of the query language and the introduction of the notion of relevance, a central notion in IR, into the framework of logical Tree Matching and complete answer aggregates.

Chapter 8 reviews related work in the fields of DBS and IR and shows points of contact.

Appendix B contains an informal description of the example document collection, "Movie Collection", used as running example in this work, gives a DTD for the collection and lists the example documents in XML format as well as depicts their abstract tree structure.

Chapter 2

Kilpeläinen's Tree Matching

Pekka Kilpeläinen developed in his PhD thesis [Kil92] a simple and elegant model for a graphical query formalism to structured documents based on the notion of homomorphisms between trees. In this chapter we will outline informally the central features this model and discuss its strengths and weaknesses. In the following chapters we will develop a logical reformulation of Kilpeläinen's Tree Matching that shall overcome its weaknesses while preserving its strengths. In Section 3.4 we will discuss the differences between Kilpeläinen's Tree Matching model and this reformulation.

2.1 Kilpeläinens Query, Database and Query Evaluation Model

In the Tree Matching model a query is an ordered and labeled tree and the database is a set of ordered and labeled trees. Nodes of the query and the database can either be text nodes or structural nodes. Text nodes can only occur as leaves and bear the textual content of the document (for database nodes) or the search terms (for query nodes). Structural nodes impose a structure on the database and query, e.g. dividing the text into chapters, titles, paragraphs, etc. The query describes a pattern that is to be matched in the database. An answer to a query and a database is a mapping from the query nodes[1] to the database nodes that respects certain homomorphic requirements. One condition is that answers have to respect labels and textual content in the following sense: A mapping f from the nodes of a query to the nodes of a database is called *label-respecting* iff the image of every query node under f has the same label as the query node itself and the text of every text node in the query occurs as a substring in the text of its image node in the database (which has to be a text node itself). Depending on the nature of the additional requirements Kilpeläinen distinguishes ten classes of Tree Matching problems. The requirements are grouped into two orthogonal classes, one class concerning the order (ordered vs. unordered), the other concerning hierarchic relationships (tree vs. path vs. region vs. child vs. subtree inclusion). The ten problem classes derive from the combination of the two requirement classes. We will discuss the four most important (within the context of structured document retrieval) classes of Tree Matching problems: Ordered/unordered tree/path inclusion.

Hierarchical Requirements A hierarchical requirement determines what hierarchical relations a mapping f has to preserve in order to be called an answer.

[1] The formalism allows to restrict answers to certain query nodes of interest.

A mapping f from a query Q to a database T is called *tree-respecting* if for all nodes u, v in Q

- $f(u) = f(v)$ iff $u = v$,
- $\text{label}(u) = \text{label}(f(u))$ and
- u is an ancestor of v in Q iff $f(u)$ is an ancestor of $f(v)$ in T.

A mapping f from a query Q to a database T is called *path-respecting* if for all nodes u, v in Q

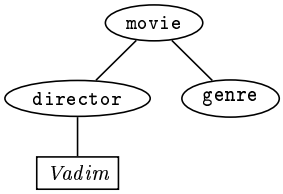
- u is the root in Q iff $f(u)$ is the root in T.
- $f(u) = f(v)$ iff $u = v$,
- $\text{label}(u) = \text{label}(f(u))$ and
- u is a parent of v in Q iff $f(u)$ is a parent of $f(v)$ in T.

Order Requirements We assume that we have total orders $<$ on the nodes of the query and the nodes of the database. These orders reflect the order on siblings and lifts it to all nodes in the standard way, i.e. $u < v$ iff u is an ancestor of v or u appears in a subtree on the left of v.

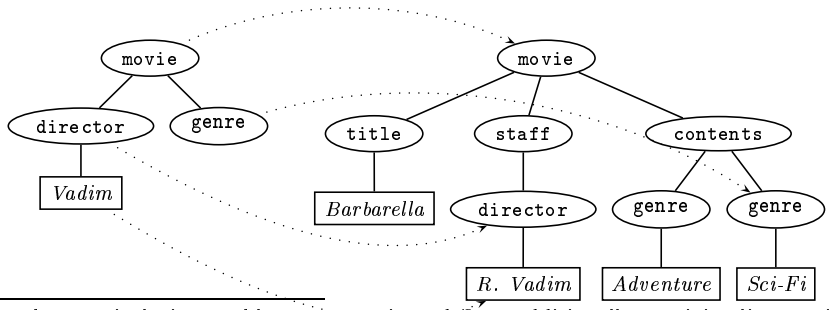
A mapping from a query Q to a database T is called *order-respecting* if for all nodes u, v in Q holds: $u < v$ iff $f(u) < f(v)$.

Definition 2.1 A label-respecting mapping f from a query Q to a database T is called an *answer to the unordered tree inclusion (ordered tree inclusion, unordered path inclusion, ordered path inclusion, resp.) problem defined by Q and T* iff f is tree-respecting (order-respecting and tree-respecting, path-respecting, order-respecting and path-respecting, resp.).[2]

Example 2.2 Take as an example the tree[3] depicted at page 135 containing information about the movie "Barbarella"as a database (consisting of one tree only) and the following query asking for the genres of movies directed by Roger Vadim[4]:



In the following picture we see one possible answer for ordered and unordered tree inclusion to the query and the database[5] consisting of the "Barbarella"-document only:



[2] For the tree inclusion problems, Kilpeläinen defines additionally a minimality requirement that we do not consider here.

[3] Some parts of the tree are omitted for layout reasons.

[4] More exactly: someone with name "Vadim". In the examples to follow we will neglect this difference that occurs if two different people have the same family name.

[5] The document tree is shortened for layout reasons.

Note that there is no answer to the query and database for both path inclusion problems, since the `movie`-node in the document has no `director`-child.

If the user has exact knowledge of the structure of the database he or she may use path inclusion problems. This can also help to exclude undesired answers, since the query is interpreted in a more strict way. In the other case, tree inclusion problems help the user to retrieve nodes whose exact position with respect to their path from the root is not known or irrelevant. In any case, both classes of Tree Matching problems are undispensable for most real-life situations. Ordered Tree Matching problems allow the user to state that two nodes in the database shall appear in a given order. The fact that all formalisms for structured document retrieval known to the literature (as reviewed for example in [FSW99, BYN96, Loe94]) model concepts very similar to the four problem classes described above makes clear that the underlying concepts of these classes are essential to a system for structured document retrieval.

Query Evaluation Kilpeläinen describes bottom-up algorithms for evaluating the various classes of Tree Matching problems. These algorithms visit the database nodes in a bottom-up manner and mark every database node v that qualifies as a candidate for an answer mapping $f : u \mapsto v$ with all respective query nodes u. Since all descendants of a visited database node v have already been visited, it is easy to verify whether the subtree spanned by a query node u can be mapped homomorphically to the subtree of v, with meeting the respective requirements. If this process assigns the root of the query to a database node, a solution is found.

Table 2.1 shows the complexity results for the four classes of Tree Matching problems defined above. Note that these results do only refer to the decision problem, i.e. the problem whether there exists an answer to a query and a database. An algorithm computing the set of *all* answers may take more time than the decision problem alone, as we will point out in the following discussion.

	tree incl.	path incl.
unordered	NP-complete	$\mathcal{O}(q^{1.5} \cdot n)$
ordered	$\mathcal{O}(q \cdot n)$	

Table 2.1: Complexities of Tree Matching Problems

2.2 Discussion

The Tree Matching formalism for retrieving structured documents described in [Kil92] is a very elegant and simple formalism. We will discuss its advantages (+) and disadvantages (-) in comparison to alternative models for structured document retrieval as reviewed in [FSW99], [BYN96] and [Loe94]. The deficiencies of Tree Matching will serve as a motivation and starting point for the following chapters.

+ **Graphical Query Language and Easy Semantics** The Tree Matching model is very intuitive and easy to understand due to its graphical query language and easy semantics. Most other models have a query language based on operators. The semantics of a complex query expression in these formalisms are not comprehensible at first sight, in contrast to a Tree Matching query.

+ **Multi-Dimensional Formalism** An answer in Tree Matching is a mapping from nodes of the query tree to nodes of the database. This multi-dimensionality gives Tree Matching a database orientation, as opposed to

other formalisms. With Tree Matching, we can retrieve, say, pairs of author/title nodes of articles satisfying given conditions. Most other formalisms for structured documents (e.g. [RLS98, NBY97, JK99, CCB95]) are one-dimensional and can only retrieve unstructured sets of nodes.

\+ **Query Power** As [CM98] showed, many query formalisms (e.g. [NBY97, ST94, CCB95]) have two restrictions with respect to their expressive power. This concerns the unability to express childhood relationship (as opposed to descendant relationship) on the one hand and the simultaneous ancestorship of a node towards two nodes (e.g. "give me all chapters including both a figure and program code"). Tree Matching can express these classes of queries.

\+ **Grammar Independence** Tree Matching does not depend on the existence of a grammar. This fits to new developments in the modeling of structured documents, namely the concept of wellformed XML documents, that may exist without a describing grammar (DTD). Nonetheless grammars can, if provided, be exploited for query optimization as elaborated in [Kil92].

\- **NP-Completeness** Kilpeläinen showed that (the decision problem whether there exists a solution for) unordered tree inclusion is intractable. All other formalisms for structured document retrieval implement a very similar concept to unordered tree inclusion without being intractable. We will change the original problem definition of Tree Matching in this work slightly in order to avoid this intractability result. This change will be discussed in Section 3.4.

\- **Combinatorial Explosion** Even in the cases where Kilpeläinen showed that the decision problem can be solved in almost linear time, an enumeration of all answers may take exponential time in the size of the database, since there may be $\binom{n}{q}$ answers to a Tree Matching query (with n (q, resp.) being the number of nodes in the database (query, resp.)). This number holds for all four relevant Tree Matching problem classes. We illustrate two orthogonal phenomena of the combinatorial explosion in the number of answers in Examples 2.3 and 2.4.

Example 2.3 Take as example the query on the left in the following illustration consisting of a root v_0 with q children $v_1, \ldots, v_q$ ($q \geq 1$) and the database on the right consisting of one tree with root d and n children $d_1, \ldots, d_n$ ($n \geq 1, n > q$). All nodes have the same label.

There are $\binom{n}{q} \cdot q! = \frac{n!}{(n-q)!}$ answers to the query and database for the unordered path and tree inclusion problems and $\binom{n}{q} = \frac{n!}{q! \cdot (n-q)!}$ for the ordered path and tree inclusion problems. This can be seen as follows: Node v_0 has to be mapped to node d_0. We have to chose q distinct nodes $\{d_{i_1}, \ldots, d_{i_n}\}$ out of the set $\{d_1, \ldots, d_n\}$ as targets for the nodes in $\{v_1, \ldots, v_q\}$. There are $\binom{n}{q}$ different possibilities for chosing these q distinct nodes. In the case of ordered path or tree inclusion the mapping between the nodes $\{v_1, \ldots, v_q\}$ and $\{d_{i_1}, \ldots, d_{i_n}\}$ is determined by the order. In the case of unordered path or tree inclusion there are $q!$ distinct enumerations $\{d_{k_1}, \ldots, d_{k_n}\}$ of $\{d_{i_1}, \ldots, d_{i_n}\}$, where each enumeration defines a mapping $v_j \mapsto d_{k_j}$ for all $1 \leq j \leq q$. In this case this combines to a total of $\binom{n}{q} \cdot q!$ different answers.

Example 2.4 The following illustration depicts a sequence query (on the left side) and a sequence database of q nodes and n nodes, resp. ($n > q$)). The

query consists of nodes $v_1, \ldots, v_q$ ($q \geq 2$) and the database consists of nodes $d_1, \ldots, d_n$ ($n \geq 2$), so that v_i (d_j, resp.) is a child of v_{i-1} (d_{j-1}, resp.) for $2 \leq i \leq q$ ($2 \leq j \leq n$, resp.).

There is one answer to the query and database for both path inclusion problems and $\binom{n}{q} = \frac{n!}{q!\cdot(n-q)!}$ for both tree inclusion problems. The argumentation is basically the same as in Example 2.3 for the unordered tree or path inclusion problems.

This high number of answers caused by permutation of answer parts has two negative effects for the practical use of Tree Matching: The time for computing and presenting the set of answers is at least as high as the number of answers, i.e. $\binom{n}{q}$. And for the user a simple enumeration of the set of all answers is not satisfactory at all. Dependencies and permutation effects are not made explicit and in bigger examples the sheer number of answers (with taking into account that each answer consists of q variable bindings itself) can blur the information being searched for.

To avoid the combinatorial explosion in the number of answers is one of the main reasons for introducing the new data structure "Complete Answer Aggregates" in this work, more specifically in Chapter 4.

- **Inflexible Query Language** The user is bound to the disjoint problem classes. It is not possible to formulate "hybrid" queries where, say, one part of the query is evaluated according to the tree inclusion problem, whereas another part is evaluated according to the path inclusion problem. The same is true for ordered and unordered queries.

  The user has no possibilities to formulate additional constraints on the nodes like proximity, similarity, semantic comparisons, typing information, attribute values, etc. beyond the concepts of edges, labels and order.

  In Chapter 3 we will describe a notion of queries allowing for hybrid queries and additional constraints on nodes.

- **Closedness** For many database and Information Retrieval applications it is useful to have a closed algebra, i.e. a formalism where the result of a query can be conceived as a database that can be queried itself. In Information Retrieval this is for example necessary for efficient treatment of query refinement. In Database Systems closedness is a necessary condition for referential transparency, i.e. the possibility to compose complex queries out of subqueries in a consistent way. In Tree Matching the result of a query is a mapping, but not a tree. In Section 3.3.6 we will describe a simple possibility how to conceive answer mappings as trees. In Section 4.6 we will describe another way based on the new data model for answers introduced in this work.

As a conclusion of this discussion we can see that, if we want to use the advantages of the Tree Matching formalism for real-world applications, we have to overcome its deficiencies. In the following chapters we will develop a logical reformulation of the Tree Matching formalism, that allows for more flexibility in query formulation and avoids the intractability result for unordered tree inclusion. Then

we will define the notion of complete answer aggregates, a special data structure that can be computed efficiently and is presented to the user as an intuitive representation of the set of all answers. This avoids the two problems caused by the combinatorial explosion in the number of answers.

Chapter 3

Logical Tree Matching

This chapter will define a logical query and database model as a reformulation of Kilpeläinen's Tree Matching formalism. Since the introduced query formalism based on formulae of PL1 (first-order predicate logic) is in no way intended to be used by the end user and we do not want to drop the graphical query language with all its advantages, we will show how the logical formulation corresponds with Kilpeläinen's formulation based on mappings between trees. In Chapter 6 we will describe a retrieval environment based on the logical Tree Matching formalism developed in this chapter. Among other things, the reformulation will allow for hybrid queries that break the strict division of the distinct Tree Matching problem classes, and it shows how constraints beyond the concepts of edges, labels and order can be formulated. We will conclude the chapter with a discussion on how the proposed model fits with widespread standards like SGML or XML and a comparison of Kilpeläinen's Tree Matching model and the logical reformulation presented here.

3.1 Formal Preliminaries

In this section we provide some basic mathematical background that is needed later. As usual, if R denotes a binary relation on a set M, then R^* (resp. R^+) denotes the reflexive-transitive (resp. transitive) closure of R.

Definition 3.1 A (finite, unordered) *tree* is a pair (V, E) where V is a finite, nonempty set and E is a binary relation on V such that the following conditions are satisfied:

1. E is cycle-free, i.e., E^+ is irreflexive,
2. for each $v' \in V$ there exists at most one element $v \in V$ such that $\langle v, v' \rangle \in E$,
3. there exists exactly one element $v \in V$, called the *root* of (V, E), such that there exists no $v' \in V$ with $\langle v', v \rangle \in E$.

The elements of V are called *nodes*. If $\langle v, v' \rangle \in E$, then v' is a *child* of v, conversely v is called the *parent node* of v'. Condition 2) expresses that each node has at most one parent node. Distinct children of a common parent node are called *siblings*. If $\langle v, v' \rangle \in E^+$, then v' is a *descendant* of v, conversely v is called an *ancestor* of v'. If $\langle v, v' \rangle \in E^*$, then v' is a *reflexive descendant* of v, and v is a *reflexive ancestor* of v'. It follows from conditions 1) and 3) that every node $v \in V$ is a reflexive descendant of the root u, i.e. $\langle u, v \rangle \in E^*$ holds. A node is a *leaf* if it does not have any child, otherwise it is an *inner node*.

A *path* of (V, E) is a sequence $\langle v_0, \ldots, v_n \rangle \in V^n$ where v_0 is the root of (V, E), v_n is a leaf of (V, E) and $\langle v_i, v_{i+1} \rangle \in E$ for $0 \leq i \leq n-1$. A *partial path* of (V, E) is a prefix of a path of (V, E).

Definition 3.2

- A *forest* is a pair (V, E), where V is a finite nonempty set and E a binary relation on V so that E is cycle-free and for every node $v \in V$ there exists at most one element u so that $\langle u, v \rangle \in E$.
- A *sequence* is a tree (V, E) where every node u has at most one child v.

From the definitions it follows that trees are a special case of forests. We will use the terminology coined for trees in the same way for forests and sequences.

Definition 3.3 An *ordered tree* (*ordered forest*) is a tree (forest) (V, E), together with a strict partial order $<_{lr}^V$ on V, called *left-to-right ordering*, that has the following properties:

1. $<_{lr}^V$ relates two nodes $v_1 \neq v_2$ (in the sense that either $v_1 <_{lr}^V v_2$ or $v_2 <_{lr}^V v_1$) iff v_1 and v_2 have a common reflexive ancestor in (V, E) and neither v_1 is a descendant of v_2 in (V, E) nor vice versa, and
2. $v_1 <_{lr}^V v_2$ implies that $v_1' <_{lr}^V v_2'$ for all reflexive descendants v_1' and v_2' of v_1 and v_2, respectively.

If (V, E) is an ordered tree or forest, the union of descendant relationship with left-to-right ordering is called the *pre-order relationship* and denoted $<_p^V = E^+ \cup <_{lr}^V$. We write $u \leq_p^V v$ for $u <_p^V v \vee u = v$. The following lemmata are a trivial consequence of the definitions.

Lemma 3.4 *Let (V, E) be an ordered tree or forest, let $u, v \in V$. Then $u <_{lr}^V v$ iff there exists a node $u' \in V$ such that $u <_{lr}^V u'$ and $u' \leq_p^V v$.*

Proof. Let $u <_{lr}^V v$. Chose $u' = v$, then $u <_{lr}^V u'$ and $u' \leq_p^V v$ hold. Now let $u <_{lr}^V u'$ and $u' \leq_p^V v$ hold. Then either (1) $u' = v$, (2) $u' <_p^V v$, or (3) $u' \rightharpoonup^+ v$. In case (1) we are finished, and in caes (2) the claim follows from condition 2) of $<_{lr}$. In case 3) the claim follows from transitivity of $<_{lr}^V$. □

Lemma 3.5 *If (V, E) is a tree, then the relation $<_p^V$ is a strict total order on V.*

Proof. Totality follows from Condition 1) of Definition 3.3. Antisymmetry: Let $u <_p^V v$. Then v is a descendant of u or $u <_{lr}^V v$. In both cases the assumption $v <_p^V u$ leads to a contradiction. Next we show transitivity of $<_p^V$. Since E^+ and $<_{lr}^V$ are both transitive we only have to show that (a) $u <_{lr}^V v$ and $\langle v, w \rangle \in E^+$ imply $u <_p^V w$ and (b) $\langle u, v \rangle \in E^+$ and $v <_{lr}^V w$ imply $u <_p^V w$. In both cases it follows from the assumptions with definition of $<_{lr}^V$ that $u <_{lr}^V w$ and therefore $u <_p^V$ holds. □

3.2 Semantics: Modeling Documents as Tree Structures

As established by international standards like SGML [ISO86] or XML ([W3C98a]), we conceive documents as labeled and ordered trees where nodes and edges are used to model the logical structure of the document. The leaves of a document

tree contain the actual flat textual content of the document. Obviously, in order to represent databases with structured documents as trees we have to adapt this basic data structure to the special requirements of structured documents. As a first step we introduce a formal distinction between "structural" nodes and "textual" nodes. Depending on the application area it might be interesting to model additional relations on the nodes, and to make these relations available in the querying process. One important example that we discuss below is the left-to-right ordering between nodes. Since we do not want to restrict the discussion to a specific set of relations, other relations that might be relevant, such as e.g. inequality, semantic comparisons, typing information, attribute values, similarity, or proximity, are treated in a generic way as abstract constraints, leaving apart their precise nature. The resulting structures for modeling document databases will be treated as conventional structures of first-order predicate logic in the following chapters.

3.2.1 Document Structures

Let Σ and Γ denote two fixed disjoint alphabets, called *text alphabet* and *markup alphabet* respectively. We assume that the textual content of a document is modeled by elements of Σ^* (i.e., by strings over Σ) and the structural markup (labels of structural nodes) is modeled by symbols in Γ.

Definition 3.6 A *document structure* is a tuple $\mathcal{D} = (D_S, D_T, \rightharpoonup, Lab, Txt)$ where

1. $(D_T \cup D_S, \rightharpoonup)$ is a forest where D_T and D_S represent disjoint sets of nodes,
2. Lab is a function that assigns to each node $d \in D_S$ an element of Γ as label, and
3. Txt is a function that assigns to each node $d \in D_T$ a string $Txt(d) \in \Sigma^*$ as textual content.

The following condition must hold:

4. Each node in D_T is a leaf of $(D_T \cup D_S, \rightharpoonup)$.

In the sequel, the nodes in D_T and D_S are called *text nodes* and *structural nodes* respectively, and $D := D_T \cup D_S$ will always denote the joint set of nodes. The elements of D are called *document nodes* or *database nodes* and are identified by convention with (decorated) symbols d, e.

Example 3.7 In Appendix B.2 the reader may find a set of trees representing a document structure.

With a *document path* of $\mathcal{D}$ we mean a path of $(D_T \cup D_S, \rightharpoonup)$.

Recall also that with a tree or forest we mean an unordered tree or forest if not said otherwise. Ordered trees and forests can be modeled with the relations that we introduce now.

3.2.2 Relational Document Structures

We will now discuss relations that can be established on the document nodes and used in queries by the user. We introduce a generic relation model that allows to model arbitrary relations. But the focus lies on the left-to-right ordering of document nodes, which will be used in all of the following chapters. In Appendix A we will also discuss other useful relations as examples how to exploit the generic relation mechanism.

Let $\mathcal{R}$ denote a fixed (not necessarily finite) set of *relation symbols*, each equipped with a fixed arity.

Definition 3.8 A *relational document structure* is a tuple $\mathcal{D} = (D_S, D_T, \rightharpoonup, Lab, Txt, I)$, where $(D_S, D_T, \rightharpoonup, Lab, Txt)$ is a document structure and I is an interpretation function for $\mathcal{R}$, i.e., a mapping that assigns to every relation symbol $r \in \mathcal{R}$ of arity k a relation $I(r) \subseteq D^k$. (Recall that $D = D_S \cup D_T$.)

We will use sometimes an abbreviated infix notation for binary relation symbols and write urv instead of $\langle u, v\rangle \in I(r)$ if I is determined by the context.

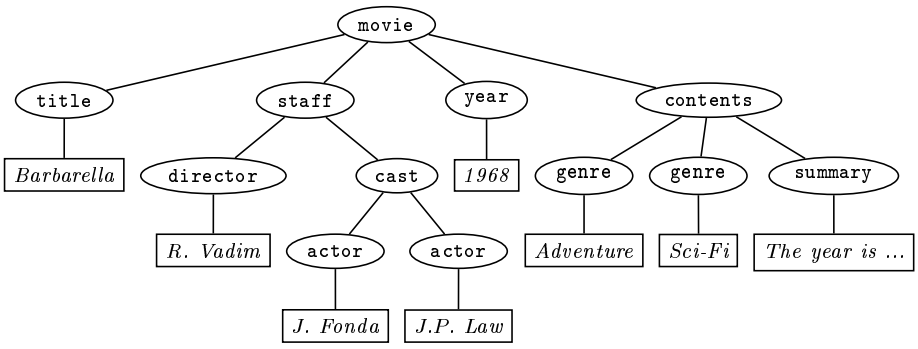
Examples for Other Relations

Since the user can only query those relations on the document nodes that are defined in the document structure, it is important to fix the set of relations for an application. In Appendix A we will outline, in addition to the order relation defined in the following, some common relations that are typical for structured documents.

3.2.3 Ordered Document Structures

Definition 3.9 A relational document structure $\mathcal{D} = (D_S, D_T, \rightharpoonup, Lab, Txt, I)$ is *ordered* if $\mathcal{R}$ contains a symbol $<_{lr}$ where $(D_T \cup D_S, \rightharpoonup)$ together with $<_{lr}^{D} := I(<_{lr})$ is an ordered forest.

Example 3.10 The following figure depicts an ordered relational document structure taken from the "Movie Collection" in Appendix B that represents a description of a movie.[1] Textual nodes are indicated by rectangles and structural nodes are indicated by oval nodes. The label or the textual content, respectively, is contained inside the nodes. The order $<_{lr}$ between the nodes is the left-to-right ordering suggested by the graphical presentation. If we refer to the nodes by their label or textual content, then, for example, *Barbarella*< `year`and *J. Fonda* < *J.P. Law* holds, but `staff` and `director` are incomparable with $<_{lr}$.



In the example above the definition of the relation $<_{lr}$ was depicted implicitly in the picture. This is not possible for all relations. In others cases, we may use explicit, additional edges (or hyperedges) that connect nodes in the relational document structure for representing a relation on the nodes. The reader may find examples for this in Appendix A.

Remark 3.11 When using markup languages like SGML or XML (see for example the XML representation of the example database "Movie Collection" in Appendix B.3), the logical structure of a document is encoded using a flat representation by means of delimiting tags. Various interesting aspects of the tree structure can be read off in a simple way:

[1] Some parts of the tree on page 135 are omitted for layout reasons.

1. two nodes d_1, d_2 stand in the pre-order relationship, $d_1 <_p^D d_2$, iff the opening tag for d_1 precedes the opening tag for d_2,

2. two nodes d_1, d_2 stand in the left-to-right relationship, $d_1 <_{lr}^D d_2$, iff the closing tag for d_1 precedes the opening tag for d_2,

3. node d_2 is a descendant of node d_1 iff both the opening and end tag for d_2 are between the two tags for d_1.

3.2.4 Modeling SGML and XML Documents as Relational Document Structures

XML ([W3C98a]) and SGML ([ISO86, Gol90]) are rich and widespread languages to describe structured documents and document classes. We will discuss what features of SGML and XML documents can be modeled with relational document structures and how this can be done. If the reader is not familiar with the XML or SGML standard, he or she may find introductions for example in [GP98, vH94, Oas]. The three central features in SGML and XML documents are elements, attributes and entities. Elements with their hierarchical containment are modeled as nodes of the document trees as seen for example in Appendix B.3. Attributes can be modeled with the help of relations as outlined in Appendix A.

Entities require some more considerations. Entities basically constitute a kind of string replacement mechanism for SGML and XML documents. In the most cases, entitities are used to represent special characters like ä or ç. In this case we can apply the replacement before considering the documents. Another use of entities is the inclusion of external entities, e.g. files etc. These external entities are possibly treated in a special way, for example pictures in JPG format. Obviously, the storage of these external entities in a document database based on our formalism is superfluous since their content can not be queried in a meaningful way with our query formalism. The treatment of multimedia objects requires a dedicated treatment (e.g. [BYRN99a, FGR98, OP98, KB96]) that is not part of our work. In some circumstances, entities are used as a means for modeling shared data. In the tree model underlying the logical Tree Matching shared data is not expressible. We can decide to neglect the aspect of information sharing and simply expand all entities, i.e. replace them by their defined value. This redundant modeling can be accompanied by notorious, undesired side-effects analyzed in the normalization theory for database systems (e.g. [Ull89]). These side-effects refer to anomalies evoked by changes to the stored data. Since this work does not cover changes to the stored data, we can ignore these anomalies.

Another feature of SGML and XML documents refers to links that are implemented with `ID/IDREF` attributes. With this feature, SGML and XML documents should be conceived more adequately as directed graphs. In our framework we have to model `ID/IDREF` attributes as normal attributes as discussed above. But then it is desirable to implement a join on attributes, i.e. a facility to query whether two attributes of two nodes have equal values. While this concept can be integrated into the document and query model easily, it is not sure wether it can be implemented in a way that performs efficiently.

Especially SGML, but also XML, offers a lot more sophisticated features like marked sections, processing instructions, multiple document structures, etc., most of which are only expressible indirectly in our model or not at all. Since these features do not belong to the core of XML or SGML we will not go deeper into this discussion.

As a conclusion we can see that we can cover with our model the central features of XML or SGML.

3.2.5 Modeling Semistructured Data as Relational Document Structures

The models for semistructured data (e.g. the OEM model described in [PGMW95]) are more general than relational document structures since they are based upon graphs rather than trees. In addition they allow for multiple labels on nodes and labels on edges. The latter point can be modeled with logical Tree Matching as the discussion in Appendix A shows. In [MS00] a framework is described that applies the methods introduced here to graph databases that are used for example in semistructured data. A method based on comparison of textual contents to simulate at least a DAG structure of the underlying database is discussed in Remark 6.11 in Section 6.

3.3 Syntax: Querying Structured Documents

In this section we define a logical query language $\mathcal{L}_\mathcal{R}$ for databases that are represented in the form of relational document structures. In principle the full first-order language associated with the given structure, presented in the previous section, may be used as query language. For practical use in IR-systems, the sublanguage of so-called "tree-queries" seems to be of particular relevance. This sublanguage, introduced in the second part of this section, will be studied in the following under various aspects.

Note that the sublanguage introduced now is not intended to be used by the end-user. Instead it is a formal basis for a structured document retrieval system. We will describe in Chapters 6 and 5.7 how this formal basis can be used for the design and implementation of a system and how the user can employ it.

3.3.1 The full first-order language

Recall that the alphabets Σ, Γ as well as the signature $\mathcal{R}$ are assumed to be fixed. In the sequel, let *Var* denote a countably infinite set of variables, denoted $x, y, z \ldots$, and let "$\propto$" ("contained in"), "$\lhd$" ("parent of") and "$\lhd^+$" ("ancestor of") denote three new binary symbols.

Definition 3.12 The set of *atomic $\mathcal{L}_\mathcal{R}$-formulae* contains all formulae of the form

- $x \lhd y$, for $x, y \in Var$ (*child formulae*),
- $x \lhd^+ y$, for $x, y \in Var$ (*descendant formulae*),
- $w \propto x$, for $x \in Var$ and $w \in \Sigma^+$ (*containment formulae*),
- $M(x)$, for $x \in Var$ and $M \in \Gamma$ (*labeling formulae*),
- $r(x_1, \ldots, x_k)$ where $r \in \mathcal{R}$ has arity k and $x_1, \ldots, x_k \in Var$ (*atomic constraints*).

Sometimes we will refer to child (descendant) formulae as *rigid* (*soft*) *edges*. *$\mathcal{L}_\mathcal{R}$-formulae* are inductively defined as follows:

- each atomic $\mathcal{L}_\mathcal{R}$-formula is an $\mathcal{L}_\mathcal{R}$-formula,
- if φ, φ_1 and φ_2 are $\mathcal{L}_\mathcal{R}$-formulae, then $\neg\varphi$, $(\varphi_1 \wedge \varphi_2)$, $(\varphi_1 \vee \varphi_2)$, and $(\varphi_1 \Rightarrow \varphi_2)$ are $\mathcal{L}_\mathcal{R}$-formulae,
- if φ is an $\mathcal{L}_\mathcal{R}$-formula and $x \in Var$, then $\exists x\varphi$ and $\forall x\varphi$ are $\mathcal{L}_\mathcal{R}$-formulae.

An $\mathcal{L}_\mathcal{R}$-formula φ is an *$\mathcal{L}$-formula* iff φ does not have a subformula that is an atomic constraint.

We write $x \lhd^{(+)} y$ when refering to formulae of the form $x \lhd y$ and $x \lhd^+ y$ at the same time.

Let $\mathcal{D} = (D_S, D_T, \rightharpoonup, Lab, Txt, I)$ be a relational document structure. With a *variable assignment in $\mathcal{D}$* we mean a total mapping $\nu : Var \to D$.

Definition 3.13 *Validity* of an atomic $\mathcal{L}_\mathcal{R}$-formula *in $\mathcal{D}$ under ν* is defined as follows[2]:

- $\mathcal{D} \models_\nu x \lhd y$ iff $\nu(x) \rightharpoonup \nu(y)$,
- $\mathcal{D} \models_\nu x \lhd^+ y$ iff $\nu(x) \rightharpoonup^+ \nu(y)$,
- $\mathcal{D} \models_\nu w \propto x$ iff $\nu(x)$ is a text node and $Txt(\nu(x))$ contains the word w,
- $\mathcal{D} \models_\nu M(x)$ iff $\nu(x)$ is a structural node and $Lab(\nu(x)) = M$,
- $\mathcal{D} \models_\nu r(x_1, \ldots, x_k)$ iff $\langle \nu(x_1), \ldots, \nu(x_k) \rangle \in r_\mathcal{D}$.

The validity relation is extended as usual to arbitrary $\mathcal{L}_\mathcal{R}$-formulae: Boolean connectives are just lifted to the meta-level, furthermore we define

- $\mathcal{D} \models_\nu \forall x \varphi$ iff for every node $d \in D$ we have $\mathcal{D} \models_{\nu'} \varphi$, where $\nu'(y) := \nu(y)$ for all variables $y \neq x$ and $\nu'(x) := d$,
- $\mathcal{D} \models_\nu \exists x \varphi$ iff there exists a node $d \in D$ such that $\mathcal{D} \models_{\nu'} \varphi$, where $\nu'(y) := \nu(y)$ for all variables $y \neq x$ and $\nu'(x) := d$.

An $\mathcal{L}_\mathcal{R}$-formula φ is *satisfiable* iff there exists a relational document structure $\mathcal{D}$ and an assignment ν such that $\mathcal{D} \models_\nu \varphi$. The set $fr(\varphi)$ of *free variables* of an $\mathcal{L}_\mathcal{R}$-formula φ is defined as usual. If $fr(\varphi)$ is given in a fixed order $\vec{x} = \langle x_1, \ldots, x_n \rangle$ and if $\vec{d} = \langle d_1, \ldots, d_n \rangle$ is a sequence of nodes of $\mathcal{D}$ we write $\mathcal{D} \models \varphi[d_1, \ldots, d_n]$ iff $\mathcal{D} \models_\nu \varphi$ for each variable assignment ν mapping x_i to d_i for $1 \leq i \leq n$.

According to Definition 3.13, atomic formulae of the form "$w \propto x$" refer only to textual nodes. Obviously, it is desirable to refer in a similar way to the part of the flat text that is dominated (indirectly) by a structural node. If x is a variable of a query that contains a structural condition of the form $M(x)$, we may write $\exists y(x \lhd^+ y \wedge w \propto y)$ to express that the word w must occur in the text associated with node x. If more comfort is needed we might add some "syntactic sugar" and introduce a new type of atomic formula that represents the above conjunction.

Definition 3.14 A *query* is a pair $Q = (\varphi, \vec{x})$ where φ is an $\mathcal{L}_\mathcal{R}$-formula and $\vec{x}$ is a fixed enumeration of $fr(\varphi)$. The set $fr(\varphi)$ is also called the set of free variables of Q and denoted in the form $fr(Q)$.

Note that we do not require that every variable x in a query has a corresponding labeling formula $M(x)$. Variables may remain unlabeled what makes the query language more flexible.

Definition 3.15 Let $\mathcal{D}$ be a relational document structure, let $Q = (\varphi, \vec{x})$ be a query where $\vec{x} = \langle x_1, \ldots, x_n \rangle$. A sequence $\vec{d} = \langle d_1, \ldots, d_n \rangle$ of nodes of $\mathcal{D}$ is an *answer* to Q in $\mathcal{D}$ iff $\mathcal{D} \models \varphi[d_1, \ldots, d_n]$. The set

$$A_\mathcal{D}(Q) := \{\langle d_1, \ldots, d_n \rangle \in D^n \mid \mathcal{D} \models \varphi[d_1, \ldots, d_n]\}$$

is called the *complete set of answers* for Q in $\mathcal{D}$.

[2] In the third condition we do not formally define what it means for a textual node to "contain" a given word w. This might mean "literal" containment, i.e., containment as a substring, or it might include more sophisticated notions of containment. See the discussion on text containment in Section 3.3.3.

Equivalently, an answer $\langle d_1, \ldots, d_n \rangle$ can be considered as a partial variable assignment mapping x_i to d_i for $i = 1, \ldots, n$. Both perspectives will be used in the sequel.

Example 3.16 Consider the relational document database "Movie Collection" in Appendix B.2 and the query (being a translation of the Tree Matching query in Example 2.2)

$$\begin{aligned} Q \quad = \quad & (m \lhd^+ d \wedge d \lhd v \wedge m \lhd^+ g \wedge \\ & \texttt{movie}(m) \wedge \texttt{director}(d) \wedge \text{"}\textit{Vadim}\text{"} \propto v \wedge \texttt{genre}(g), \\ & \langle m, d, v, g \rangle). \end{aligned}$$

This query has the following three answers in the database:

$$\{m \mapsto \texttt{e1}, d \mapsto \texttt{e5}, v \mapsto \texttt{e6}, g \mapsto \texttt{e15}\}$$
$$\{m \mapsto \texttt{g1}, d \mapsto \texttt{g5}, v \mapsto \texttt{g6}, g \mapsto \texttt{g21}\}$$
$$\{m \mapsto \texttt{g1}, d \mapsto \texttt{g5}, v \mapsto \texttt{g6}, g \mapsto \texttt{g23}\}$$

The efficient computation of the complete set of answers to a given query can be considered as the principal problem of an IR system. In this paper we concentrate on a particular subclass of queries.

3.3.2 Tree Queries

It is natural to assume that most queries aim to retrieve subtrees of a particular form from the database. As long as matters of universal quantification are ignored, the tree queries introduced in the following definition are a canonical choice for this retrieval task.

Definition 3.17 A *tree query* is a query $Q = (\psi \wedge c, \vec{x})$ where ψ is a conjunction of atomic $\mathcal{L}$-formulae and c is a conjunction of atomic constraints with $fr(c) \subseteq fr(\psi)$ such that the following condition is satisfied: there exists a variable $x \in fr(Q)$, called the *root* of Q, such that for each $y \in fr(Q)$ there exists a unique sequence of variables $x = x_0, \ldots, x_n = y$ $(n \geq 0)$ where ψ contains subformulae $x_i \lhd^{(+)} x_{i+1}$ for $0 \leq i \leq n-1$.

A *sequence query* is a tree query $Q = (\psi \wedge c, \vec{x})$ where $\vec{x} = \langle x_1, \ldots, x_q \rangle$ so that ψ contains formulae $x_i \lhd^{(+)} x_{i+1}$ for all $1 \leq i \leq q$.

The following lemma follows immediately from Definition 3.17.

Lemma 3.18 *Let $Q = (\psi \wedge c, \vec{x})$ be a tree query. Then*

1. *ψ does not have any $\lhd^{(+)}$-cycle, i.e., there is no sequence of variables $x_0, \ldots, x_n$ $(n \geq 0)$ such that ψ contains subformulae $x_i \lhd^{(+)} x_{i+1}$ for $0 \leq i \leq n-1$ and a subformula $x_n \lhd^{(+)} x_0$,*

2. *the root of Q is unique,*

3. *for each $y \in fr(Q)$ there exists at most one formula in ψ of the form $x \lhd^{(+)} y$, and*

4. *if Q is a sequence query for each $x \in fr(Q)$ there exists at most one formula in ψ of the form $x \lhd^{(+)} y$.*

Definition 3.19 A tree query $Q = (\psi \wedge c, \vec{x})$ is *inconsistent* if

1. φ contains a formula $w \propto x$, and a formula of the form $M(x)$ or $x \lhd^{(+)} y$ at the same time, or

2. if φ contains two formulae $M(x)$ and $M'(x)$ for $M' \neq M \in \Gamma$.

Clearly, inconsistent tree queries are unsatisfiable. Below we shall see that every consistent tree query is satisfiable if we do not restrict the interpretation of relation symbols in $\mathcal{R}$. Henceforth, with a tree query we always mean a consistent tree query. Since tree queries have a tree shape, standard notions from trees can be used for describing their structure:

Definition 3.20 Let $Q = (\psi \wedge c, \vec{x})$ be a tree query. A variable $y \in fr(Q)$ is called a *rigid* (resp. *soft*) *child* of $x \in fr(Q)$ in Q iff ψ contains a formula $x \lhd y$ (resp. $x \lhd^+ y$). In this situation, x is called a rigid (soft) *parent* of y in Q. A variable x that has no children is a *leaf* in Q. Two distinct children y_1, y_2 of a variable x in Q are said to be *siblings* in Q. A variable $y \in fr(Q)$ is a *reflexive descendant* of x in Q iff there exists a chain $x = x_0, \ldots, x_k = y$ of length $k \geq 0$ such that x_{i+1} is a child (of either type) of x_i in Q, for $i = 0, \ldots, k-1$. A *partial path* of Q is a sequence $\langle x_0, \ldots, x_k \rangle$ of length $k \geq 0$ starting at the root of Q such that x_{i+1} is a child (of either type) of x_i in Q for $i = 0, \ldots, k-1$. A *path* of Q is a maximal partial path. Let $x \in fr(Q)$. The *height* $h_Q(x)$ of x with respect to Q is defined as follows: $h_Q(x) := 0$ iff x does not have any child in Q, and $h_Q(x) := \max\{h_Q(y) + 1 \mid y \text{ is a child of } x \text{ in } Q\}$ otherwise.

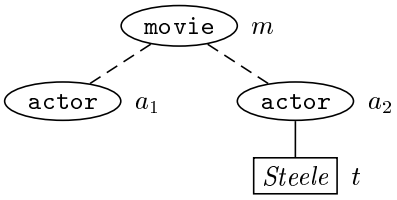
Note that x is always a reflexive descendant of x in Q, for each $x \in fr(Q)$.

From now on we will use a graphical notation for tree queries in parallel to the logical formulation. We will introduce this graphical representation here informally. A formal definition is given in Section 3.3.5. Every variable is represented as a node with the variable attached to it. The node contains a label if there is a labeling formula for the respective variable. Rigid edges are represented by solid lines between the nodes, soft edges by dashed lines. Constraints are drawn as arcs with the name of the constraint attached to. The following example will illustrate this convention. (For an example incorporating constraints see Example 3.26.)

Example 3.21 Consider the following query Q asking for actors having costarred with Barbara Steele:

$$\begin{aligned} Q \quad = \quad & (m \lhd^+ a_1 \wedge a_1 \lhd t \wedge m \lhd^+ a_2 \wedge \\ & \texttt{actor}(a_1) \wedge \texttt{actor}(a_2) \wedge \text{“}\textit{Steele}\text{”} \propto t, \\ & \langle m, a_1, t, a_2 \rangle) \end{aligned}$$

It can be represented as the following tree:



The variable m has two children, a_1 and a_2, both of which are soft children. The variables a_1 and t have no children and are leaves. The query has two paths: $\langle m, a_1 \rangle$ and $\langle m, a_2, t \rangle$. The height of the query is 2.

Existentially Quantified Tree Queries

The user may want to distinguish in the query between variables that he or she wants to see in the answer, and other variables that are only used to describe structural requirements in the query. We put this into practice with existential quantification, a mechanism used for this purpose in many branches of computer science, e.g. in Database Systems under the term projection. Variables that are existentially quantified are invisible in answers.

Example 3.22 Consider the tree query in Example 3.16. Since the user is only interested in the genres the following query seems more adequate:

$$\begin{aligned} Q \quad = \quad & (\exists m, d, v(\\ & m \lhd^+ d \wedge d \lhd v \wedge m \lhd^+ g \wedge \\ & \texttt{movie}(m) \wedge \texttt{director}(d) \wedge \text{“}\mathit{Vadim}\text{”} \propto v \wedge \texttt{genre}(g)), \\ & \langle m, d, v, g \rangle). \end{aligned}$$

This query has the following three answers in the relational document structure depicted in Appendix B.2:

$$\{g \mapsto \texttt{e15}\}, \{g \mapsto \texttt{g21}\} \text{ and } \{g \mapsto \texttt{g23}\}$$

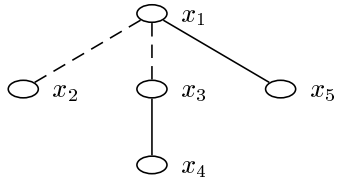
Since tree queries with existential quantifiers are treated in exactly the same way as tree queries without existential quantifiers, apart from the fact that some variables are not part of the answer mappings, we will not treat this class specifically in the rest of this work.

In the following sections three subclasses of tree queries (simple, local and partially ordered tree queries) will be considered, each obtained by restricting the classes of constraints that may be used in queries.

Simple and Local Tree Queries

Definition 3.23 Let $Q = (\psi \wedge c, \vec{x})$ be a tree query. A constraint $r(x_0, \ldots, x_n)$ of c is *Q-simple* iff either r is unary, or r is binary and x_1 (or x_0) is a child of x_0 (resp. x_1) in Q. A constraint $r(x_0, \ldots, x_n)$ of c is *Q-local* iff Q has a variable x with children $y_1, \ldots, y_h$ such that $\{x_0, \ldots, x_n\} \subseteq \{x, y_1, \ldots, y_h\}$. We say that Q is a *simple (local) tree query iff each constraint of c is Q-simple (Q-local).*

Example 3.24 Consider a tree query describing a structure as in the following illustration, i.e. $x_1 \lhd^+ x_2 \wedge x_1 \lhd^+ x_3 \wedge x_1 \lhd x_5 \wedge x_3 \lhd x_4$:



Then a constraint $c(x_1, x_2, x_5)$ is local whereas constraints $c(x_2, x_4)$ and $x(x_1, x_3, x_4)$ are not local. Both constraints are not simple, but $c(x_1, x_3)$ is a simple constraint.

It is important to note that "Q-simplicity" and "Q-locality" are purely syntactic concepts that just restrict the pairs of variables of Q that can be related in constraints.

An important class of Q-local constraints are the constraints $y_i <_{lr} y_j$ expressing that two sibling nodes stand in the left-to-right ordering relation $<_{lr}^{D}$ of $\mathcal{D}$. A formula $y_i <_{lr} y_j$ will be called a *left-to-right ordering constraint.*

Partially Ordered Tree Queries

Definition 3.25 A tree query $Q = (\psi \wedge c, \vec{x})$ is *partially ordered* if each constraint of c is either Q-simple or a left-to-right ordering constraint, and if the subset c_{lr} of left-to-right ordering constraints in c satisfies the following properties:

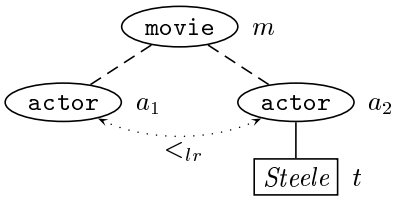
1. for each constraint $y_i <_{lr} y_j$ in c_{lr} the variables y_i and y_j are siblings with respect to Q,
2. c_{lr} does not have a cycle of the form $y_0 <_{lr} \cdots <_{lr} y_n <_{lr} y_0$.

The tree query $Q = (\psi \wedge c, \vec{x})$ is *linearly ordered* if the set of left-to-right constraints specifies a linear ordering for the set of children of each variable x of Q.

Example 3.26 Consider the following query Q, that is derived from Example 3.21 by adding an order constraint between nodes a_1 and a_2. It asks for actors having costarred with *Barbara Steele* but in a role more "important". (The importance of a role is reflected by the ordering of the actors.)

$$\begin{aligned} Q \quad = \quad & (m \lhd^+ a_1 \wedge m \lhd^+ a_2 \wedge a_2 \lhd t \wedge a_1 <_{lr} a_2 \wedge \\ & \texttt{actor}(a_1) \wedge \texttt{actor}(a_2) \wedge \text{“}\textit{Steele}\text{”} \propto t, \\ & \langle m, a_1, t, a_2 \rangle) \end{aligned}$$

Q can be represented as the following tree:



Q is a linearly ordered tree query.

Note that in particular each partially ordered tree query is a local tree query. The following lemma shows—in a sense to be made precise—that for local tree queries the possible instantiations of the reflexive descendants of a variable $x \in fr(Q)$ in answers only depend on the instantiation of x, but not on the instantiation of the ancestors of x in Q. The lemma will play an essential role for the techniques suggested in the following sections.

Lemma 3.27 *Let $Q = (\psi \wedge c, \vec{x})$ be a local tree query. Let y be a child of x in Q, let Y denote the set of proper descendants of y in Q, and let $Z = fr(Q) \setminus (Y \cup \{y\})$. Let $\langle y_1, \ldots, y_r \rangle$ and $\langle z_1, \ldots, z_s \rangle$ denote enumerations of Y and Z respectively. Assume that $\vec{x}$ has the form $\langle z_1, \ldots, z_s, y, y_1, \ldots, y_r \rangle$. If Q has two answers*

$$\begin{aligned} &\langle d_1, \ldots, d_s, \quad d \quad , e_1, \ldots, e_r \rangle \\ &\langle d'_1, \ldots, d'_s, \quad d \quad , e'_1, \ldots, e'_r \rangle \end{aligned}$$

that coincide on y, then

$$\begin{aligned} &\langle d_1, \ldots, d_s, \quad d \quad , e'_1, \ldots, e'_r \rangle \\ &\langle d'_1, \ldots, d'_s, \quad d \quad , e_1, \ldots, e_r \rangle \end{aligned}$$

are answers to Q as well.

Proof. This follows immediately from the fact that $\psi \wedge c$ does not have any atomic subformula that contains variables from Y and Z at the same time. □

3.3.3 Text Containment

In Definition 3.13 we did not exactly define in which cases the containment constraint $w \propto x$ holds. In the fields of Information Retrieval and Computational Linguistics there are many elaborated definitions of text containment that go far beyond literal containment as a substring. We will not tie ourselves down to one of the many notions of text containment, but will instead list the most common ideas here. An implementation has to chose a subset of these ideas that can be employed by the end-user. The following definitions are an (incomplete) list of meanings for "string w is contained in text node d".

Substring: w is a substring of the text associated with d.

Exact Match: w is exactly the text associated with d.

Regular expressions: w is a regular expression that can be matched with the text associated with d, e.g. "*R* Vadim*" matches "*Roger Vadim*" and "*R. Vadim*".

Distances: w contains descriptions like "words *ailment* and *cancer* separated by not more than 5 words". These descriptions are matched with the text of node d in an obvious way.

Fuzzy: w is contained in d if the text of d contains a substring that is similar to w. Depending on the notion of similarity, this notion is tolerant towards misspelling or phonetic ambiguities.

Thesaurus: w is contained in d if the text of d contains a synonym of w, a more general or a more narrow term. The user may specify in the query which of these interpretations he or she wants to use. In order to evaluate these expressions the query engine must be equipped with a thesaurus storing the semantical relations between terms.

Linguistic: The terms in the text nodes as well as the terms in the query may be subject to linguistic normalization techniques like lemmatization or stemming. Then it is possible to match, say, the term "*fly*" in a query with a substring "*... flew ...*" in the text of a document node.

3.3.4 Data-anchored Queries

We will describe in this section a special class of queries, data-anchored queries, on relational document structures that will show a very convenient behaviour in the following. For example, in Chapter 5 we will show that answers can be computer more efficiently for these queries.

Definition 3.28 Let $\mathcal{D} = (D_S, D_T, \rightharpoonup, Lab, Txt, I)$ be a relational document structure and $Q = (\phi, \vec{x})$ a query.

- A label $M \in \Gamma$ is called *periodic* in $\mathcal{D}$ iff there exists a path $\langle d_0, \ldots, d_k \rangle$ in $\mathcal{D}$ so that there are two nodes d_i and d_j $(0 \leq i, j \leq k, i \neq j)$ with $Lab(d_i) = Lab(d_j)$.
- $\mathcal{D}$ is called *periodic* iff Γ contains a periodic label.[3]
- Q is called *rigid* iff it contains no soft edges.
- Q is called *data-anchored* in $\mathcal{D}$ iff for every soft edge $x \lhd^+ y$ in Q there exists a labeling constraint $M(x)$ in Q so that M is not periodic in $\mathcal{D}$.

[3] Sometimes this property is called cyclicity or recursivity. We avoid these terms since they are already occupied with other notions in our context.

The following lemma is a trivial consequence of the definitions above:

Lemma 3.29

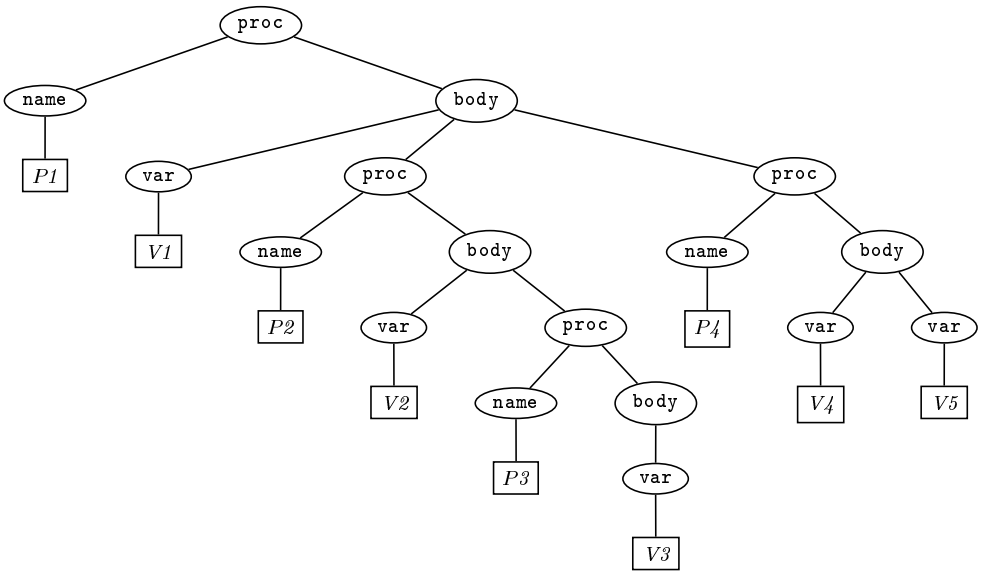
- *A rigid query is data-anchored in every relational document structure.*
- *Every query is data-anchored in a non-periodic relational document structure.*

We will later see that the combination of soft edges in a query and periodic relational document structures increase the number of answers and make computation of answers less efficient. With the class data-anchored queries we defined a class of queries on relational document structures that behave better.

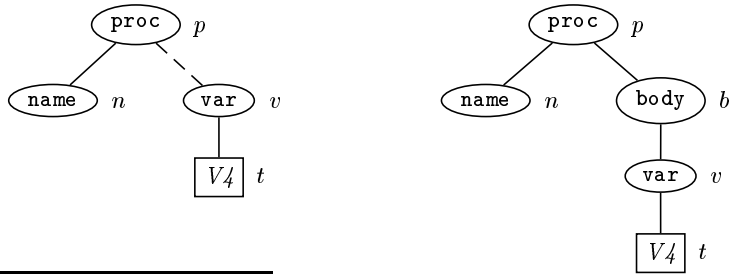
Remark 3.30 If we have a grammar describing the structure of the document database (as is the case for most XML and all SGML documents) the notions "periodic" and "data-anchored" can be specified with using the grammar only (i.e. without looking at the relational document structure itself).[4]

As a conclusion of this discussion we give a small example of a periodic document structure. Other examples include document structures that contain lists inside of lists, sections inside of sections, etc.

Example 3.31 The following document structure describes the basic structure of program code. A procedure has a name and a body. The body of a procedure may contain other procedures and variable declarations. This document structure is periodic since procedures may occur inside of procedures.



The user could pose the following queries to this relational document structure:



[4]Here we neglect the case that the grammar allows for a periodic document structure, but the concrete instance, the relational document structure, is not periodic.

The first query (on the left) searches for procedure names in whose scope variable *V4* has been defined, while the second query locates names of procedures whose body directly contains a variable definition for *V4*. The query on the left is trivially data-anchored in the relational document structure (since it is rigid), while the second is not data-anchored because the label `proc` is periodic in the relational document structure and its node has an outgoing soft edge.

3.3.5 Answers as Homomorphisms

We do not want to abandon the advantages of having a graphical query languages as the one proposed in the original Tree Matching formalism. Therefore we will present here a simple technique how to conceive tree queries as relational document structures[5]. Answers to queries correspond to homomorphisms from the relational document structure describing the query to the relational document structure representing the document database.

In order to have the same signature for relational document structures describing the query as well as relational document structures describing the document database, we have to make some slight technical changes to the definition of relational document structure. (In the rest of this work we will return to the view of relational document structures as defined in Definition 3.8.)

Definition 3.32 A *spare forest* is a triple $(V, \rightharpoonup, \overset{+}{\rightharpoonup})$ so that there exists a forest (V, E) with $\rightharpoonup \subseteq E$ and $\overset{+}{\rightharpoonup} \subseteq E^+$.

Now let Σ and Γ be fixed text and markup alphabets and let $\mathcal{R}$ be a fixed set of relation symbols. In the following discussion a relational document structure will be defined as tuples

$$\mathcal{D} = (D, \rightharpoonup, \overset{+}{\rightharpoonup}, Lab, Txt, I),$$

where $(D, \rightharpoonup, \overset{+}{\rightharpoonup})$ is a spare forest with D being composed of the disjoint sets of text and structural nodes, Lab and Txt are partial functions assigning labels in Γ to structural nodes and strings over Σ to text nodes, and I is the interpretation for the relation symbols. We now impose the additional restriction that text nodes may only contain one word[6]. This is no loss of generality, since a text node with several words can be modeled as a sequence of text nodes with one word each.

We can transform a relational document structure $(D_S, D_T, \rightharpoonup, Lab, Txt, I)$ in the sense of Definition 3.8 to a relational document structure $(D', \rightharpoonup', \overset{+}{\rightharpoonup}', Lab', Txt', I')$ according to the new definition in the following way: $D' = D_S \cup D_T, \rightharpoonup' = \rightharpoonup, \overset{+}{\rightharpoonup}' = \rightharpoonup^*, I' = I$. This transformation can be reversed since the domains of Lab and Txt identify the sets D_S and D_T.
The definition of an answer to a relational document structure remains the same apart from some small syntactical changes.

We now formulate the translation of tree queries to relational document structures.

Definition 3.33 Let $Q = (\psi \wedge c, \vec{x})$ be a tree query to a relational document structure $\mathcal{D} = (D, \rightharpoonup_D, \overset{+}{\rightharpoonup}_D, Lab_D, Txt_D, I_D)$. Then the *query tree* for Q is a relational document structure $\mathcal{Q} = (D_Q, \rightharpoonup_Q, \overset{+}{\rightharpoonup}_Q, Lab_Q, Txt_Q, I_Q)$ with the following components:

- D_Q is the set of variables in Q.

[5] We already used, in an informal way only, graphical representations of tree queries.
[6] A word is the smallest atomic entity in Σ^* that may be used in queries.

- $\rightharpoonup_Q$ is the set of pairs (x, y) so that Q contains a formula $x \lhd y$.
- $\overset{+}{\rightharpoonup}_Q$ is the set of pairs (x, y) so that Q contains a formula $x \lhd^+ y$.
- Lab_Q is a partial function $Lab_Q : Var \to \Gamma$ that assigns a label M to a variable x iff Q contains a formula $M(x)$.
- Txt_Q is a partial function $Txt_Q : Var \to \Sigma^*$ that assigns a string w to a variable x iff Q contains a formula $w \propto x$.
- I is a function assigning to every relation symbol in $r \in \mathcal{R}$ of arity k a relation $I(r) \subseteq D_Q$ so that $\langle x_1, \ldots, x_k \rangle \in I(r)$ iff Q contains a formula $r(x_1, \ldots, x_k)$.

It is easy to verify that the structure above is a relational document structure due to consistency of tree queries.

Example 3.34 The graphical representation of a query tree for a given tree query is exactly the informal graphic representation for this query as given in Examples 3.21 and 3.26.

Lemma 3.35 *Each tree query where relation symbols do not have a fixed interpretation is satisfiable.*

Proof. Let $Q = (\psi \wedge c, \vec{x})$ be a tree query. Let Q' be a tree query that is obtained from Q by replacing all soft edges by rigid edges. We may modify the query tree for Q' as follows. Each unlabeled node receives a fixed label $M \in \Gamma$. Obviously in this way a relational document structure $\mathcal{D}$ is obtained such that $\psi \wedge c$ holds in $\mathcal{D}$ under each variable assignment that maps each element of $\vec{x}$ to itself. □

Definition 3.36 Let Σ, Γ and $\mathcal{R}$ be fixed. A *document homomorphism* from a relational document structure $\mathcal{D}_1 = (D_1, \rightharpoonup_1, \overset{+}{\rightharpoonup}_1, Lab_1, Txt_1, I_1)$ to a relational document structure $\mathcal{D}_2 = (D_2, \rightharpoonup_2, \overset{+}{\rightharpoonup}_2, Lab_2, Txt_2, I_2)$ is a mapping $\nu : D_1 \to D_2$ so that the followings holds for all nodes $u_1, \ldots, u_k \in D$:

- If $u_1 \rightharpoonup_1 u_2$ then $\nu(u_1) \rightharpoonup_2 \nu(u_2)$.
- If $u_1 \overset{+}{\rightharpoonup}_1 u_2$ then $\nu(u_1) \overset{+}{\rightharpoonup}_2 \nu(u_2)$.
- If Lab_1 is defined for u_1 then $Lab_1(u_1) = Lab_2(\nu(u_1))$.
- If Txt_1 is defined for u_1 then $Txt_2(\nu(u_1))$ contains $Txt_1(u_1)$.
- If $\langle u_1, \ldots, u_k \rangle \in I_1(r)$ then $\langle \nu(u_1), \ldots, \nu(u_k) \rangle \in I_2(r)$.

The following lemma shows that homomorphisms and answers to a given tree query are equivalent notions.

Lemma 3.37 *Let $\mathcal{D}_Q = (X_Q, \rightharpoonup_Q, \overset{+}{\rightharpoonup}_Q, Lab_Q, Txt_Q, I_Q)$ be the query tree of the tree query Q, let $\mathcal{D} = (D, \rightharpoonup_D, Lab_D, Txt_D, I_D)$ be a relational document structure. A mapping $\nu : X_Q \to D$ is a document homomorphism from $\mathcal{D}_Q$ in $\mathcal{D}$ iff ν is an answer to Q in $\mathcal{D}$.*

Proof. Let $Q = (\psi \wedge c, \vec{x})$. We first show the direction "left to right":
Let $\nu : fr(Q) \to D$ be document homomorphism from $\mathcal{D}_Q$ to $\mathcal{D}$. We show for every atom φ in $\psi \wedge c$ that $\mathcal{D} \models_\nu \varphi$. All argumentations follow the same line of references, first to Definition 3.33, then to Definition 3.36, and finally to Definition 3.13:
$x \lhd y$**:** With Definition 3.33 it follows: $x \rightharpoonup_Q y$. With Definition 3.36 we have $\nu(x) \rightharpoonup_D \nu(y)$. Finally Definition 3.13 implies $\mathcal{D} \models_\nu x \lhd y$.

$x \lhd^+ y$**:** Analogously.
$w \propto x$**:** It follows $Txt_Q(x) = w$. Then we know that $Txt_D(\nu(x))$ contains w. Therefore $\mathcal{D} \models_\nu w \propto x$.
$M(x)$**:** Then $Lab_Q(x) = M$. Therefore $Lab_D(\nu(x)) = M$. This results in $\mathcal{D} \models_\nu M(x)$.
$r(x_1, \ldots, x_k)$**:** Then $\langle x_1, \ldots, x_k \rangle \in I_Q(r)$. It follows that $\langle \nu(x_1), \ldots, \nu(x_k) \rangle \in I_D(r)$, and therefore $\mathcal{D} \models_\nu r(x_1, \ldots, x_k)$.
For the inverse direction let ν be an answer to Q in $\mathcal{D}$, i.e. $\mathcal{D} \models_\nu \psi \wedge c$. We show that ν is document homomorphism by validating each case in Definition 3.36.
As in the other direction, every case in the analysis of Definition 3.36 follows the same line of arguments: First a reference to Definition 3.33, then to Definition 3.13:
$x \rightharpoonup_Q y$**:** Then $(x \lhd y) \in \psi$, and therefore $\nu(x) \rightharpoonup_D \nu(y)$.
$x \overset{+}{\rightharpoonup}_Q y$**:** Analogously.
$Lab_Q(x) = M$**:** We have $M(x) \in \psi$ and then $Lab_D(\nu(x)) = M$.
$w \in Txt_Q(x)$**:** Then $(w \propto x) \in \psi$, and therefore $Txt_D(\nu(x))$ contains w.
$\langle y_1, \ldots, y_k \rangle \in I_Q(r)$**:** It follows that c contains the atomic constraint $r(y_1, \ldots, y_k)$, and therefore $\langle \nu(y_1), \ldots, \nu(y_k) \rangle \in I_D(r)$. □

3.3.6 Answers as Relational Document Structures

As pointed out in Section 2.2 it is sometimes convenient to have a closed algebra, where the data a query works on and the answer to a query are structures with the same signature. In this case queries can be seen as filters returning just a view of a part of the database. We present in this section one simple way how to achieve this goal. In Section 4.6 we will show a more sophisticated way.

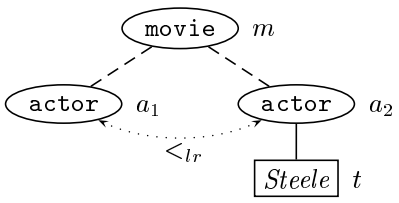
The idea presented in this section gives a definition how every single answer can be seen as a view on the database.

Definition 3.38 Let ν be an answer to a tree query $Q = (\psi \wedge c, \langle x_1, \ldots, x_q \rangle)$ and a relational document structure $\mathcal{D} = (D_S, D_T, \rightharpoonup_D, Lab_D, I_D)$. The *answer structure* $\mathcal{D}_\nu$ for ν is a relational document structure $(D_{(S,\nu)}, D_{(T,\nu)}, \rightharpoonup_\nu, Lab_\nu, Txt_\nu, I_\nu)$ defined as follows:

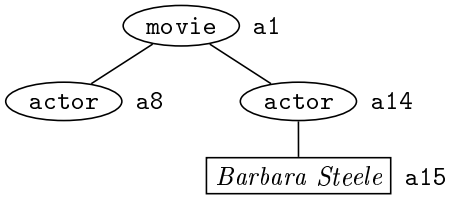
1. $D_{(S,\nu)}$ contains $\nu(x)$ for every x in $\vec{x}$ so that $\nu(x)$ is a structural node.
2. $D_{(T,\nu)}$ contains $\nu(x)$ for every x in $\vec{x}$ so that $\nu(x)$ is a text node.
3. $x \rightharpoonup_\nu y$ iff x is a (soft or rigid) child of y in Q.
4. $Lab_\nu(\nu(x)) = Lab_D(\nu(x))$ for every x in $\vec{x}$ so that $\nu(x)$ is a structural node.
5. $Txt_\nu(\nu(x)) = Txt_D(\nu(x))$ for every x in $\vec{x}$ so that $\nu(x)$ is a text node.
6. For all enumerations $\langle x_1, \ldots, x_k \rangle$ of k-ary subsets of $\vec{x}$ and every relation symbol c in $\mathcal{R}$: $\langle \nu(x_1), \ldots, \nu(x_k) \rangle \in I_\nu(c)$ iff $\langle \nu(x_1), \ldots, \nu(x_k) \rangle \in I_D(c)$.

It follows at once from this definition that an answer ν to a query Q and a relational document structure $\mathcal{D}$ is also an answer to Q and the answer structure $\mathcal{D}_\nu$.

Example 3.39 Take the following query of Example 3.26:



One answer to this query is $\nu = \{m \mapsto \texttt{a1}, a_1 \mapsto \texttt{a8}, a_2 \mapsto \texttt{a14}, t \mapsto \texttt{a15}\}$. The corresponding answer structure $\mathcal{D}_\nu$ is the following relational document structure. (The ordering relation $<_{lr}$ holds for the pairs $\langle \texttt{a8}, \texttt{a14} \rangle$ and $\langle \texttt{a8}, \texttt{a15} \rangle$.)



3.4 Comparison with Kilpeläinen's Tree Matching

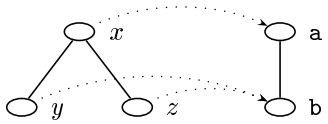
We mentioned in the introduction that the present formalism can be understood as a variant and generalization of Kilpeläinen's Tree Matching formalism [Kil92]. In this subsection we briefly comment on this point.

When we restrict Tree Matching to tree and path inclusion our formalism is more flexible than Tree Matching since in a partially ordered tree query we may specify an arbitrary partial ordering between the children of a query node, and we can also have rigid edges and soft edges at the same time. In this sense, the present formalism generalizes Tree Matching. However, there are also two subtle differences:

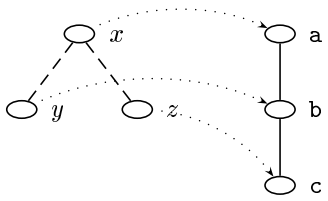
1. Kilpeläinen's homomorphic embeddings are always assumed to be injective, we do not impose such a restriction in our formalism.
2. A relation is "preserved" in Kilpeläinen's sense under a mapping h if it is preserved in both directions. For example, a mapping is said to preserve ancestorship if, for all nodes x, y of the pattern, x is an ancestor of y *if and only if* $h(x)$ is an ancestor of $h(y)$. We only demand the implication from left to right, i.e., the "only if" direction.

The following example illustrates the two differences.

Example 3.40 1. The following mapping is an answer in our formalism but not in the original Tree Matching formalism, since it is not injective.



2. The following mapping is an answer in our formalism but not in the original Tree Matching formalism (for the tree inclusion problem), since the image b of y is an ancestor of the image c of z, but y is no ancestor of z.



These innocent differences are responsible for the phenomenon that Kilpeläinen's unordered tree inclusion problem is NP-complete even in the decision version (cf. [Kil92]) whereas all the complexity results obtained here are polynomial. Since unordered tree inclusion problems represent the most natural variant of Tree Matching, the avoidance of the intractability result can be considered as a major advantage of the present formalism.

Chapter 4

Complete Answer Aggregates

This section is devoted to the problem of finding a suitable (re)presentation for the complete set of answers to a given tree query. As we will demonstrate below, the number of answers to a tree query Q may be exponential in the size of Q. Hence an explicit enumeration leads to exponential-time behaviour in the worst case. Quite generally a naive enumeration will also suffer from many redundancies since different answers may have several common sub-nodes. This makes it difficult to extract useful information from the sequence of answers.

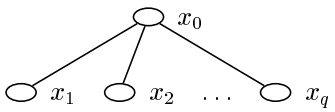
The question arises if there is a more compact and organized way of (re)presenting all answers. This includes two goals: A representation shall "contain" all answers so that they can be generated in a simple way out of this representation. On the other hand, sharing of information shall be exploited as much as possible, so that database nodes that occur as targets in different answers are only represented once if possible. This sharing of information shall give the user a good overview over the local dependencies that constrain, say, a variable y to a set $D_{(x,d,y)}$ of database nodes depending on the choice of mapping another variable x to a database node d. This means that the representation shall make explicit the answer to a reasoning like: "If I choose to map query node x to database node d what possibilities do I have for a child y of x?"

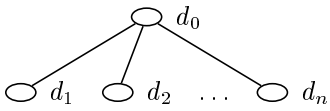
In the first part of this chapter we introduce the concept of a "complete answer formula" for a simple tree query. The "complete answer aggregates" that are introduced thereafter yield a physical representation of complete answer formulae. A complete answer aggregate yields a full representation of all answers that is quadratic in the size of the database for simple tree queries. For data-anchored queries the size is linear. In the end of this chapter these notions and results are extended to local tree queries and to ordered tree queries. Computational aspects are postponed to the next chapter.

4.1 Complete Answer Formulae for Simple Tree Queries

In order to introduce our representation technique we fix a simple tree query, Q, and a relational document structure $\mathcal{D}$ with set of nodes D. If Q has q variables, in the worst case the total number of answers to Q is of order $\mathcal{O}(|D|^q)$, even for rigid queries:

Example 4.1 Consider the rigid query Q (depicted on the left side) and the database $\mathcal{D}$ (on the right):

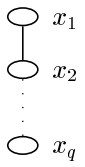
Query: 

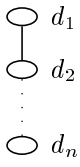
Database: 

The query Q has n^q answers in $\mathcal{D}$. The difference between this number and the number of answers for Example 2.3 is explained by the difference between the original Tree Matching formalism and our reformulation as elaborated in Section 3.4.

The example shows that nodes with a high branching degree may lead to an explosion of the number of answers. A similar example using data from the "Movie Collection" in the appendix can be found in Example 4.16. For queries with soft edges an orthogonal potential source of problems is deep nesting as already mentioned in Section 3.3.4:

Example 4.2 Consider the following query Q depicted on the left and the database $\mathcal{D}$ on the right:

Query: 

Database: 

The query Q has $\binom{n}{q} = \frac{n!}{q!\cdot(n-q)!}$ answers in $\mathcal{D}$. (The same number of answers as in Example 2.4.)

How can we avoid an enumeration of all answers? As a starting point, we take a logical perspective. As a first step we will represent the set of all answers with formulae of first order predicate logic. We use all nodes d of the relational document structure as constants d, the set *Var* as the set of variables and only one relation symbol, "=", that denotes the equality of objects. The complete set of answers to Q in $\mathcal{D}$ can then be represented as a formula in disjunctive normal form of size $\mathcal{O}(q \cdot n^q)$

$$\bigvee_{\langle d_1,\ldots,d_q\rangle \in A_{\mathcal{D}}(Q)} (\bigwedge_{i=1}^{q} x_i = d_i).$$

It is well-known that the size of the disjunctive normal form of a formula of propositional logic may be exponential in the size of the original formula. Even if we do not have anything like an "original" formula here, an obvious idea is to look for formulae that are logically equivalent to the disjunction of all answers but of smaller size, and to use a shared representation for multiple occurrences of the same subformula. Of course, from a practical point of view the formulae must offer a transparent view of all answers, and given the formula it should be possible to generate each particular answer without computational effort. Before we introduce a suitable class of formulae, let us illustrate the basic idea using the above examples.

Example 4.3 The set of all answers in Example 4.1 can be encoded as a formula of size $q \cdot n$ of the form

$$x_0 = d_0 \wedge \bigwedge_{i=1}^{q} (\bigvee_{j=1}^{n} y_i = d_j).$$

The set of all answers in Example 4.2 can be encoded as a formula of size $q \cdot \binom{n}{q}$ of the form

$$\bigvee_{i_1=1}^{n-q+1} (x_1 = d_{i_1} \wedge \bigvee_{i_2=d_{i_1}+1}^{n-q+2} (x_2 = d_{i_2} \wedge \bigvee_{i_3=d_{i_2}+1}^{n-q+3} (\ldots \bigvee_{i_q=d_{i_{q-1}}+1}^{n} (x_q = d_{i_q}) \ldots)))$$

Given this formula, each answer to Q can be immediately obtained in the following way. Select a possible value d_{i_1} for x_1 in the outermost disjunction. Each possible choice leads to a specific subdisjunction that gives possible values for x_2. Continuing in the same way, the choice of a value d_{i_k} for x_k $(k < q)$ always determines a new subdisjunction that determines a set of possible values for x_{k+1}. In more detail, a value $x_k = d_{i_k}$ always implies that the value for x_{k+1} can recruit from $\{d_{i_k+1}, \ldots, d_{n-q+k+1}\}$.

Although the second formula is not substantially smaller than the number of bindings $x \mapsto d$ in the set of all answers, its structure serves as a starting point for finding a compact representation for the set of all answers.

Let ϵ stand for the empty sequence $\langle\rangle$. The following definition generalizes the above type of representation, introducing a class of formulae that may be used for "dependent instantiation" of variables, given Q. The idea is to instantiate the variables of Q in a top-down manner, where the sets of possible values of descendant variables with respect to Q depend on the chosen instantiations of the ancestor variables.

Definition 4.4 Let $Q = (\psi \wedge c, \vec{x})$ be a simple tree query, let $x \in fr(Q)$. The set of *dependent Q-instantiation formulae for x* is inductively defined as follows. First assume that $h_Q(x) = 0$. For each non-empty set $D_x \subseteq D$, the formula

$$\Delta_{\epsilon}^{\langle x \rangle} :\equiv \bigvee_{d \in D_x} x = d$$

is a dependent Q-instantiation formula for x. Now assume that $h_Q(x) > 0$. Let $\emptyset \neq D_x \subseteq D$. For each $d \in D_x$ and each child y of x in Q, let $\Delta_{\langle d \rangle}^{\langle x,y \rangle}$ be a dependent Q-instantiation formula for y. Then

$$\Delta_{\epsilon}^{\langle x \rangle} :\equiv \bigvee_{d \in D_x} x = d \wedge \bigwedge_{y \text{ child of } x \text{ in } Q} \Delta_{\langle d \rangle}^{\langle x,y \rangle})$$

is a dependent Q-instantiation formula for x. There are no other dependent Q-instantiation formulae for x besides those defined above. The set D_x is called the set of *target candidates for x* in $\Delta_{\epsilon}^{\langle x \rangle}$.

Note that the notion of a dependent Q-instantiation formula is defined in a purely syntactical way and does not refer to an answer. In the sequel we use expressions of the form $\Delta_{\langle d_0,\ldots,d_{k-1} \rangle}^{\langle x_0,\ldots,x_k \rangle}$ for refering to subformulae of a dependent Q-instantiation formula $\Delta_{\epsilon}^{\langle x_0 \rangle}$. These subformulae are inductively defined as follows: Assume that $\Delta_{\langle d_0,\ldots,d_{k-1} \rangle}^{\langle x_0,\ldots,x_k \rangle}$ is a dependent Q-instantiation subformula of $\Delta_{\epsilon}^{\langle x_0 \rangle}$ of the form

$$\bigvee_{d \in D_{x_k}} (x_k = d \wedge \bigwedge_{y \text{ child of } x_k \text{ in } Q} \Delta_{\langle d \rangle}^{\langle x_k,y \rangle})$$

where $k \geq 0$. If x_{k+1} is a child of x_k in Q and $d_k \in D_{x_k}$, then $\Delta_{\langle d_0,\ldots,d_k \rangle}^{\langle x_0,\ldots,x_{k+1} \rangle}$ denotes $\Delta_{\langle d_k \rangle}^{\langle x_k,x_{k+1} \rangle}$. Furthermore each disjunct

$$x_k = d \wedge \bigwedge_{y \text{ child of } x_k \text{ in } Q} \Delta_{\langle d \rangle}^{\langle x_k,y \rangle}$$

is written in the form $\delta_{\langle d_0,\ldots,d_k \rangle}^{\langle x_0,\ldots,x_k \rangle}$. As an immediate consequence of these definitions we obtain

Remark 4.5 Modulo associativity and commutativity of "$\wedge$" and "$\vee$", each formula $\delta^{\langle x_0,\ldots,x_k\rangle}_{\langle d_0,\ldots,d_k\rangle}$ is uniquely determined by its subformulae of the form $\Delta^{\langle x_0,\ldots,x_k,y\rangle}_{\langle d_0,\ldots,d_k\rangle}$ where y is a child of x_k in Q. Similarly each subformula of the form $\Delta^{\langle x_0,\ldots,x_{k-1},x_k\rangle}_{\langle d_0,\ldots,d_{k-1}\rangle}$ is uniquely determined by its subformulae of the form $\delta^{\langle x_0,\ldots,x_k\rangle}_{\langle d_0,\ldots,d_k\rangle}$.

The following definition captures the notion of incrementally instantiating a dependent Q-instantiation formula in a top-down manner.

Definition 4.6 The set of *partial* (resp. *total*) *instantiations* of a dependent Q-instantiation formula $\Delta^{\langle x_0\rangle}_{\epsilon}$ is inductively defined as follows: the empty set "$\emptyset$" is a partial instantiation of $\Delta^{\langle x_0\rangle}_{\epsilon}$. Assume that ν is a partial instantiation of $\Delta^{\langle x_0\rangle}_{\epsilon}$. If $\Delta^{\langle x_0\rangle}_{\epsilon}$ has a subformula of the form $\delta^{\langle x_0,\ldots,x_k\rangle}_{\langle d_0,\ldots,d_k\rangle}$, if $\{\langle x_i,d_i\rangle \mid 1\leq i\leq k-1\}\subseteq\nu$ and ν does not have a pair of the form $\langle x_k,d\rangle$ $(d\in D)$, then $\nu\cup\{\langle x_k,d_k\rangle\}$ is a partial instantiation of $\Delta^{\langle x_0\rangle}_{\epsilon}$. There are no other partial instantiations besides those defined by the above rules. A partial instantiation ν of $\Delta^{\langle x_0\rangle}_{\epsilon}$ is a *total instantiation* iff ν contains a pair $\langle x,d\rangle$ for every $x\in fr(\Delta^{\langle x_0\rangle}_{\epsilon})$.

Lemma 4.7 *Let Q be a simple tree query, let $\Delta^{\langle x_0\rangle}_{\epsilon}$ be a dependent Q-instantiation formula. Then every partial instantiation of $\Delta^{\langle x_0\rangle}_{\epsilon}$ can be extended to a total instantiation of $\Delta^{\langle x_0\rangle}_{\epsilon}$. The set of total instantiations of $\Delta^{\langle x_0\rangle}_{\epsilon}$ is non-empty.*

The lemma can be proven by a trivial induction on $h_Q(x_0)$.

Lemma 4.8 *Let Q be a simple tree query, let $\Delta^{\langle x_0\rangle}_{\epsilon}$ be a dependent Q-instantiation formula for x_0, let $d_0,\ldots,d_k$ be elements of D. Then the following conditions are equivalent:*

1. *$\Delta^{\langle x_0\rangle}_{\epsilon}$ has a subformula $\Delta^{\langle x_0,\ldots,x_k\rangle}_{\langle d_0,\ldots,d_{k-1}\rangle}$ where d_k is a target candidate for x_k,*
2. *$\Delta^{\langle x_0\rangle}_{\epsilon}$ has a subformula of the form $\delta^{\langle x_0,\ldots,x_k\rangle}_{\langle d_0,\ldots,d_k\rangle}$,*
3. *Q has formulae $x_0\lhd^{(+)}x_1,\ldots,x_{k-1}\lhd^{(+)}x_k$ and $\{\langle x_i,d_i\rangle \mid 0\leq i\leq k\}$ is a partial instantiation of $\Delta^{\langle x_0\rangle}_{\epsilon}$.*

Proof. The equivalence "1 $\Leftrightarrow$ 2" follows immediately from the definition of these subformulae. To prove the implication "2 $\Rightarrow$ 3", let $\delta^{\langle x_0,\ldots,x_k\rangle}_{\langle d_0,\ldots,d_k\rangle}$ be a subformula of $\Delta^{\langle x_0\rangle}_{\epsilon}$. The definition of these formulae shows that x_{i+1} is a child of x_i, for $i=0,\ldots,k-1$. Hence Q has formulae $x_0\lhd^{(+)}x_1,\ldots,x_{k-1}\lhd^{(+)}x_k$. A trivial induction on k shows that $\{\langle x_i,d_i\rangle \mid 0\leq i\leq k\}$ is a partial instantiation of $\Delta^{\langle x_0\rangle}_{\epsilon}$. The inverse implication "3 $\Rightarrow$ 2" follows by a trivial induction on k. □

We now come to the central definition of this chapter.

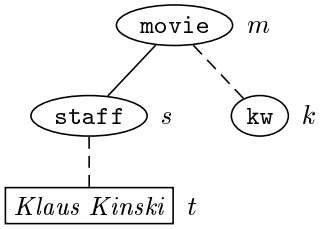
Definition 4.9 Let Q be a simple tree query with root x_0. A dependent Q-instantiation formula $\Delta^{\langle x_0\rangle}_{\epsilon}$ for x_0 is called a *complete answer formula* for Q iff each answer to Q is a total instantiation of $\Delta^{\langle x_0\rangle}_{\epsilon}$ and vice versa.

Complete answer formulae will be denoted in the form Δ_Q. Some of the following formulations become simpler when introducing the falsum "$\bot$" as an additional dependent Q-instantiation formula. By convention, "$\bot$" does not have any instantiation.

Example 4.10 Consider the query

$$\begin{aligned} Q \quad = \quad & (m \lhd s \wedge s \lhd^+ t \wedge m \lhd^+ k \wedge \\ & \texttt{movie}(m) \wedge \texttt{staff}(s) \wedge \mathit{Klaus\ Kinski} \propto t \wedge \texttt{kw}(k), \\ & \langle m, s, t, k \rangle) \end{aligned}$$

asking for keywords for the films where Klaus Kinski belonged to the staff.



If we restrict this query to the "Paganini"-document depicted at page 136 in the appendix, we have the following 12 answers:

$$\begin{array}{llll}
\{m \mapsto \texttt{i1}, & s \mapsto \texttt{i4}, & t \mapsto \texttt{i6}, & k \mapsto \texttt{i26}\} \\
\{m \mapsto \texttt{i1}, & s \mapsto \texttt{i4}, & t \mapsto \texttt{i6}, & k \mapsto \texttt{i28}\} \\
\{m \mapsto \texttt{i1}, & s \mapsto \texttt{i4}, & t \mapsto \texttt{i6}, & k \mapsto \texttt{i30}\} \\
\{m \mapsto \texttt{i1}, & s \mapsto \texttt{i4}, & t \mapsto \texttt{i6}, & k \mapsto \texttt{i32}\} \\
\{m \mapsto \texttt{i1}, & s \mapsto \texttt{i4}, & t \mapsto \texttt{i9}, & k \mapsto \texttt{i26}\} \\
\{m \mapsto \texttt{i1}, & s \mapsto \texttt{i4}, & t \mapsto \texttt{i9}, & k \mapsto \texttt{i28}\} \\
\{m \mapsto \texttt{i1}, & s \mapsto \texttt{i4}, & t \mapsto \texttt{i9}, & k \mapsto \texttt{i30}\} \\
\{m \mapsto \texttt{i1}, & s \mapsto \texttt{i4}, & t \mapsto \texttt{i9}, & k \mapsto \texttt{i32}\} \\
\{m \mapsto \texttt{i1}, & s \mapsto \texttt{i4}, & t \mapsto \texttt{i17}, & k \mapsto \texttt{i26}\} \\
\{m \mapsto \texttt{i1}, & s \mapsto \texttt{i4}, & t \mapsto \texttt{i17}, & k \mapsto \texttt{i28}\} \\
\{m \mapsto \texttt{i1}, & s \mapsto \texttt{i4}, & t \mapsto \texttt{i17}, & k \mapsto \texttt{i30}\} \\
\{m \mapsto \texttt{i1}, & s \mapsto \texttt{i4}, & t \mapsto \texttt{i17}, & k \mapsto \texttt{i32}\}
\end{array}$$

These answers can be represented as the following complete answer formula:

$$\begin{array}{l} m = \texttt{i1} \wedge \\ \qquad (s = \texttt{i4} \wedge \\ \qquad\qquad (t = \texttt{i6} \vee t = \texttt{i9} \vee t = \texttt{i17}) \end{array}) \wedge (k = \texttt{i26} \vee k = \texttt{i28} \vee k = \texttt{i30} \vee k = \texttt{i32})$$

We may now give the first kernel result.

Theorem 4.11 *For each simple tree query $Q = (\psi \wedge c, \vec{x})$ and each relational document structure $\mathcal{D}$ there exists a complete answer formula Δ_Q which is unique modulo associativity and commutativity of "$\wedge$" and "$\vee$".*

Proof. First assume that Q does not have any answer in $\mathcal{D}$. Then "$\bot$" is a complete answer formula for Q. It follows from Lemma 4.7 that Q does not have another complete answer formula. Assume now that Q has at least one answer. Since we later see how to compute a complete answer formula Δ_Q for Q (cf. Chapter 5) we only prove the uniqueness part here. Let x_0 be the root of Q, let $\Delta_\epsilon^{\langle x_0 \rangle}$ and $\Lambda_\epsilon^{\langle x_0 \rangle}$ be complete answer formulae for Q. Lemma 4.7 and Lemma 4.8 show that $\Delta_\epsilon^{\langle x_0 \rangle}$ has a subformula $\delta_{\langle d_0,\ldots,d_k \rangle}^{\langle x_0,\ldots,x_k \rangle}$ iff $\Lambda_\epsilon^{\langle x_0 \rangle}$ has a subformula $\lambda_{\langle d_0,\ldots,d_k \rangle}^{\langle x_0,\ldots,x_k \rangle}$. Starting at the subformulae with maximal k it is then trivial to prove by "inverse" induction

using Remark 4.5 that corresponding formulae $\delta^{\langle x_0,\ldots,x_k\rangle}_{\langle d_0,\ldots,d_k\rangle}$ and $\lambda^{\langle x_0,\ldots,x_k\rangle}_{\langle d_0,\ldots,d_k\rangle}$ are equal modulo associativity and commutativity of "$\wedge$" and "$\vee$". It follows that $\Delta^{\langle x_0\rangle}_{\epsilon}$ and $\Lambda^{\langle x_0\rangle}_{\epsilon}$ are equal modulo associativity and commutativity of "$\wedge$" and "$\vee$". □

Since we want to obtain a representation where multiple occurrences of the same subformula are shared, the following simple observation is crucial. The proof depends strongly on Lemma 3.27.

Lemma 4.12 *Let Δ_Q be a complete answer formula for the simple tree query Q. Then two subformulae of Δ_Q of the form $\delta^{\langle x_0,\ldots,x_{k-1},x_k\rangle}_{\langle d_0,\ldots,d_{k-1},d_k\rangle}$ and $\delta^{\langle x_0,\ldots,x_{k-1},x_k\rangle}_{\langle d'_0,\ldots,d'_{k-1},d_k\rangle}$ are always identical modulo associativity and commutativity of "$\wedge$" and "$\vee$".*

Proof. The proof of Theorem 4.11 shows that it suffices to verify the following: if $\delta^{\langle x_0,\ldots,x_{k-1},x_k\rangle}_{\langle d_0,\ldots,d_{k-1},d_k\rangle}$ has a subformula of the form $\delta^{\langle x_0,\ldots,x_{k-1},x_k,x_{k+1},\ldots,x_{k+r}\rangle}_{\langle d_0,\ldots,d_{k-1},d_k,d_{k+1},\ldots,d_{k+r}\rangle}$, then $\delta^{\langle x_0,\ldots,x_{k-1},x_k\rangle}_{\langle d'_0,\ldots,d'_{k-1},d_k\rangle}$ has a subformula of the form $\delta^{\langle x_0,\ldots,x_{k-1},x_k,x_{k+1},\ldots,x_{k+r}\rangle}_{\langle d'_0,\ldots,d'_{k-1},d_k,d_{k+1},\ldots,d_{k+r}\rangle}$ and vice versa.
Let $\delta^{\langle x_0,\ldots,x_{k-1},x_k\rangle}_{\langle d_0,\ldots,d_{k-1},d_k\rangle}$ have a subformula of the form $\delta^{\langle x_0,\ldots,x_{k-1},x_k,x_{k+1},\ldots,x_{k+r}\rangle}_{\langle d_0,\ldots,d_{k-1},d_k,d_{k+1},\ldots,d_{k+r}\rangle}$. By Lemma 4.8, Δ_Q has partial instantiations of the form $\{\langle x_i,d_i\rangle \mid 0\le i\le k+r\}$ and $\{\langle x_i,d'_i\rangle \mid 0\le i\le k\}$ where $d_k=d'_k$. By Lemma 4.7 there exist answers ν_1 (resp.ν_2) of Q that extend the former (latter) partial instantiation. By Lemma 3.27 there exists an answer ν_3 to Q that coincides with answer ν_1 on the set of reflexive descendants of x_k and with answer ν_2 on all other variables in $fr(Q)$. Answer ν_3 extends the partial instantiation $\{\langle x_i,d'_i\rangle \mid 0\le i\le k-1\}\cup\{\langle x_i,d_i\rangle \mid k\le i\le k+r\}$. Hence $\delta^{\langle x_0,\ldots,x_{k-1},x_k\rangle}_{\langle d'_0,\ldots,d'_{k-1},d_k\rangle}$ has a subformula of the form $\delta^{\langle x_0,\ldots,x_{k-1},x_k,x_{k+1},\ldots,x_{k+r}\rangle}_{\langle d'_0,\ldots,d'_{k-1},d_k,d_{k+1},\ldots,d_{k+r}\rangle}$. By symmetry the lemma follows. □

The following lemma is a simple consequence of Lemma 4.7, proven by structural induction on x, stating that a complete answer formula contains no superfluous information:

Lemma 4.13 *For every variable x in a simple tree query Q and every atom $x=d$ in a complete answer formula Δ_Q for Q and a relational document structure $\mathcal{D}$, there exists a total instantiation ν of with $\nu(x)=d$.*

4.2 Aggregates for Simple Tree Queries

Our next aim is to give a compact physical representation of complete answer formula.

Complete Answer Aggregates

Lemma 4.12 shows that for each pair (x_k,d_k) (where $x_k\in fr(Q)$ and $d_k\in D$) all subformulae of Δ_Q of the form $\delta^{\langle x_0,\ldots,x_{k-1},x_k\rangle}_{\langle d_0,\ldots,d_{k-1},d_k\rangle}$ are identical. We shall write them in the form $\delta_{x_k}(d_k)$. In the physical representation, all occurrences of a subformula $\delta_x(d)$ are shared and represented as a field $Agg_x[d]$ of a record[1] Agg_x assigned to the variable x.

Definition 4.14 Let $Q=(\psi\wedge c,\vec{x})$ be a simple tree query. An *aggregate* for Q is a family Agg_Q of records, $\{Agg_x \mid x\in\vec{x}\}$. Each record is composed of a finite

[1] In some contexts, the data structure that is used here, with an open number of fields that are accessed by arbitrary keys, are called "dictionaries" and distinguished from "records" (which have a fixed number of fields). Since the terminus "dictionary" is preoccupied to a certain extend in our context we prefer to ignore this difference here.

number of fields with indices $d \in D$, denoted $Agg_x[d]$. For each child y of x in Q, the field $Agg_x[d]$ contains a list of pointers, $Agg_x[d, y]$. Each pointer in a list $Agg_x[d, y]$ points to a field $Agg_y[e]$ of the record Agg_y. Distinct pointers of $Agg_x[d, y]$ point to distinct fields.

In the sequel we will sometimes with the terms "slot x or "slot Agg_x" refer to the record Agg_x and use the term link as a synonym for pointer.

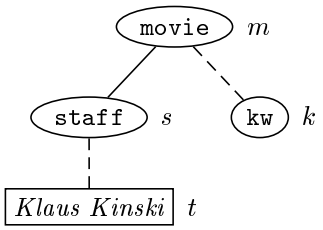
In Examples 4.16, 4.20 and 4.21 graphical representations for aggregates may be found. Since we are concerned in this section with the size of answers we will define the *size* of an aggregate as the number of pointers of the aggregate. Modulo a constant factor this value reflects the storage space needed for an aggregate.

Definition 4.15 Let $Q = (\psi \wedge c, \vec{x})$ be a simple tree query, let Δ_Q denote the complete answer formula for Q. A *complete answer aggregate* for Q is a Q-aggregate $Agg_Q = \{Agg_x \mid x \in \vec{x}\}$ that satisfies the following conditions:

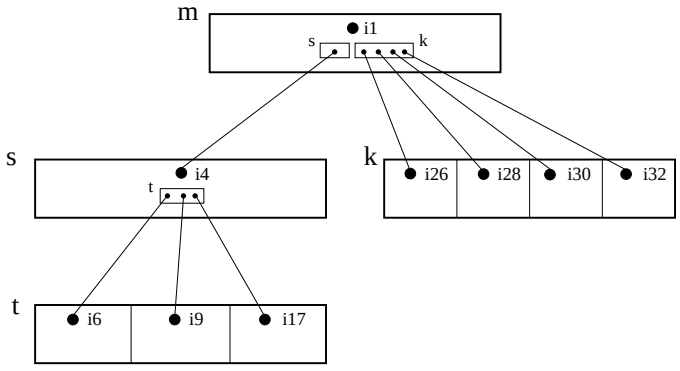
(CAA1) a record Agg_x has a field $Agg_x[d]$ iff Δ_Q has a subformula $\delta_x(d)$,

(CAA2) a list $Agg_x[d, y]$ has a pointer to a field $Agg_y[e]$ iff $\delta_x(d)$ contains a subformula of the form $\delta_y(e)$.

Example 4.16 Consider the query in Example 4.10 asking for keywords for films where Klaus Kinski belonged to the staff:



Restricting this query to the "Paganini"-document depicted at page 136 in the appendix, we have the following complete answer aggregate:



We can recognize the similarity to the original complete answer formula presented in Example 4.10. This complete answer aggregate stores 12 answers (see Example 4.10). (The complete answer aggregate for this query applied to all documents in Appendix B.2 can be found in Example 5.26.)

Remark 4.17 Ignoring the trivial case of an unsatisfiable query it is easy to see that a complete answer formula Δ_Q for a simple tree query Q uniquely determines the corresponding complete answer aggregate Agg_Q. Conversely, given a complete

answer aggregate Agg_Q we may reconstruct the complete answer formula Δ_Q in the following way: to obtain Δ_Q,

- read the record Agg_x of the root x of Q as the disjunction of the formulae associated with the fields $Agg_x[d]$,
- associate with each field $Agg_x[d]$ the conjunction of $x = d$ with the formulae associated with the pointer lists $Agg_x[d, y]$,
- associate with each list of pointers $Agg_x[d, y]$ of a field $Agg_x[d]$ the disjunction of the formulae associated with the address fields of the pointers.

With the help of the following definition we will see how the individual answers can be extracted from a complete answer aggregate:

Definition 4.18 A mapping $\nu : fr(Q) \to D$ is an *instantiation* of the aggregate $Agg_Q = \{Agg_x \mid x \in fr(Q)\}$ iff the following conditions are satisfied:

(INST1) Agg_x has a field $Agg_x[\nu(x)]$ for all $x \in fr(Q)$,

(INST2) if y is a child of x in Q, if $d = \nu(x)$ and $e = \nu(y)$, then e is an address node of a vertical pointer of a column $Agg_x[d, y[l, *]]$ of the array $Agg_x[d, y]$.

We say that each field $Agg_x[\nu(x)]$ *belongs to* the instantiation ν. Similarly each pointer column $Agg_x[d, y[l, *]]$ of the form described in (INST2) is said to belong to ν.

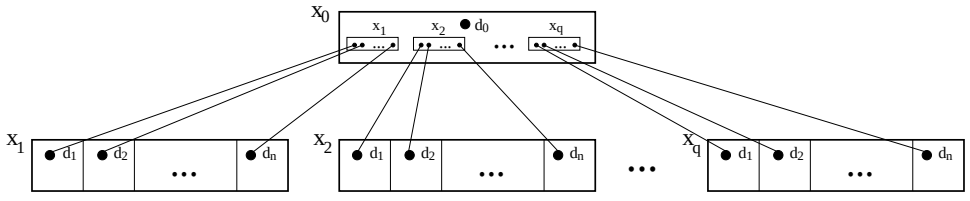
Instantiations can be formed out of an complete answer aggregate Agg_Q, similar to the complete answer formula Δ_Q, in a recursive top-down fashion by following the pointers.

From the definition of complete answer formulae and the correspondence between complete answer formulae and complete answer aggregates it follows that every instantiation of a complete answer aggregate for a simple tree query Q and a relational document structure $\mathcal{D}$ is an answer to Q and $\mathcal{D}$ and vice versa.

Example 4.19 We can verify in Examples 4.16 and 4.10 that every instantiation of the complete answer aggregate is an answer to the query and database, and vice versa.

To make the correspondence between the concepts of complete answer aggregates and complete answer formulae more obvious we add the following two examples.

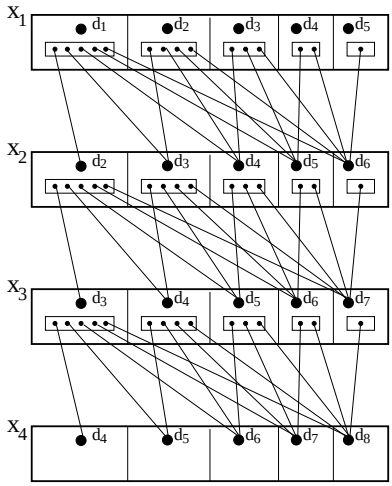
Example 4.20 The complete answer aggregate for the first formula in Example 4.3 (an encoding of all answers to the rigid tree query given in Example 4.1) can be depicted as follows.



The root variable x can only be instantiated with d_0. All other variables can be instantiated with each of the nodes $d_1, \ldots, d_n$.

In this example, the number of pointers of the aggregate is $q \cdot n$. Hence the size of the complete answer aggregate is of order $\mathcal{O}(q \cdot n)$.

Example 4.21 For the special case $q = 4$ and $n = 8$ the complete answer aggregate for the second formula in Example 4.3 (an encoding of the 70 answers to the tree query in Example 4.2) is of size 45, i.e. contains 45 pointers:



For arbitrary $n \geq q$, each record Agg_{x_q} contains $n - q + 1$ target candidates. The first target candidate in each record Agg_{x_i} (apart from the last record Agg_{x_q}, which contains no pointers at all) has $n-q+1$ pointers to target candidates in $Agg_{x_{i+1}}$, the last target candidate contains only one pointer. Hence each record Agg_{x_i} contains $1 + \ldots + (n - q + 1) = \frac{(n-q+1)(n-q+2)}{2}$ pointers, apart from the target candidates in the leaf record Agg_{x_q}. Therefore the total number of pointers in the aggregate is $(q-1) \cdot \frac{(n-q+1)(n-q+2)}{2}$.

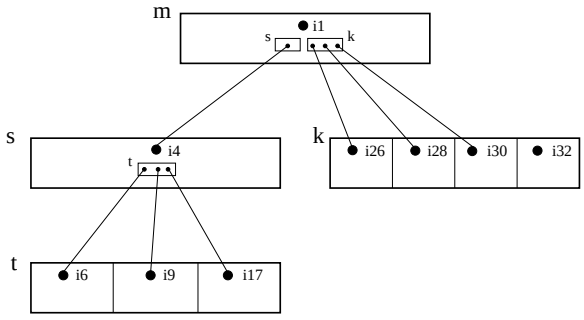
As the example shows, a field of a complete answer aggregate can serve as the address of several pointers if the query contains soft edges.

Minimality of Complete Answer Aggregates

We will now show that a complete answer aggregate is a minimal representation of the set of all answers in the following sense:

Definition 4.22 An aggregate Agg is called *minimal* iff all its fields and pointers belong to an instantiation of Agg.

Example 4.23 The following aggregate is not minimal because target candidate `i32` does not belong to an instantiation:



Lemma 4.24 *The complete answer aggregate Agg for a simple tree query Q and a relational document structure $\mathcal{D}$ is minimal.*

Proof. Since Agg directly corresponds to the complete answer formula Δ_Q for Q and $\mathcal{D}$ the claim follows directly from Lemma 4.13. □

Due to the correspondence of instantiations of complete answer aggregates and answers, the above lemma states that the complete answer aggregate contains no superfluous nodes or pointers, i.e. no nodes or pointers that do not contribute to an answer.

Size of Complete Answer Aggregates

We show now that the size of a complete answer aggregate for a data-anchored simple tree query Q is linear both in the size of the query and the database. In the sequel, let $|Q|$ denote the number of symbols of Q. This means in particular that the number of variables of Q and the number of atomic constraints of Q is bounded by $|Q|$. With $|D|$ we denote the cardinality of D.

Lemma 4.25 *Let Q be a data-anchored simple tree query in a relational document structure $\mathcal{D}$ and Agg the complete answer aggregate for Q and $\mathcal{D}$. For every field $Agg_y[e]$ in Agg there exists maximally one pointer with target $Agg_y[e]$.*

Proof. Each pointer with target $Agg_y[e]$ starts at a field of the form $Agg_x[d]$ where x is the parent of y in Q: the definition of the complete answer aggregate implies that the complete answer formula, Δ_Q, has a formula $\delta_x(d)$ with subformula $\delta_y(e)$. Lemma 4.8 and Lemma 4.7 show that Δ_Q has a total instantiation ν mapping x to d and y to e. If y is a rigid child of x in Q it follows that $d = \nu(x)$ is the unique parent of $e = \nu(y)$ in $\mathcal{D}$, since ν is an answer to Q. If y is a soft child of x in Q then Q contains a labeling constraint $M(y)$ so that M is not periodic in $\mathcal{D}$, since Q is data-anchored. In this case $d = \nu(x)$ is the unique ancestor of $e = \nu(y)$ with label M in $\mathcal{D}$. □

Theorem 4.26 *Let $\mathcal{D}$ be a relational document structure and let Q be a data-anchored simple tree query in $\mathcal{D}$. Then the size of the complete answer aggregate for Q is of order $\mathcal{O}(|Q| \cdot |D|)$.*

Proof. The complete answer aggregate Agg for Q contains $\leq |Q|$ records Agg_x, the total number of fields $Agg_x[d]$ is bounded by $|Q| \cdot |D|$. Lemma 4.25 shows that for a fixed field $Agg_y[e]$ there is at most one pointer ending at $Agg_y[e]$. Therefore the total number of pointers is bounded by $|Q| \cdot |D|$ as is the number of fields in Δ_Q. It follows that the total size of Agg is of order $\mathcal{O}(|Q| \cdot |D|)$. □

Theorem 4.26 depends on the fact that each target candidate in an aggregate is pointed at only once. If we conceive the aggregate as a graph, with the target candidates as nodes and the pointers as edges, then the resulting graph is a tree or forest if the query is data-anchored in the relational document structure. For arbitrary tree queries and databases, every target can be pointed at by more than one pointer. The graph induced by the aggregate is not necessarily a tree and the maximal size also depends on the maximal length of a document path, denoted $h_{\mathcal{D}}$:

Theorem 4.27 *Let $\mathcal{D}$ be a relational document structure and let Q be a simple tree query. Then the size of the complete answer aggregate for Q is of order $\mathcal{O}(|Q| \cdot |D| \cdot h_{\mathcal{D}})$.*

Proof. In this case a field $Agg_y[e]$ can serve as the address of at most $h_{\mathcal{D}}$ pointers: in fact all pointers with address $Agg_y[e]$ start from some field $Agg_x[d]$ where x is the parent of y in Q and d is an ancestor of e. There are at most $h_{\mathcal{D}}$ ancestors of e, and for a fixed field $Agg_x[d]$ there is at most one pointer from $Agg_x[d]$

to $Agg_y[e]$. Hence the total number of pointers is bounded by $|Q| \cdot |D| \cdot h_{\mathcal{D}}$. It follows that the total size of Agg_Q is of order $\mathcal{O}(|Q| \cdot |D| \cdot h_{\mathcal{D}})$. □

It should be noted that in both bounds we could replace the size $|Q|$ of the query by the number of variables occurring in Q.

4.3 Complete Answer Aggregates and Local Dependencies

One motivation of this chapter was the search for a representation technique that allows to generate all answers in a simple way and that makes explicit local dependencies.

We saw how instantiations, and thus answers, can be obtained from complete answer aggregates in a simple way. With the notion of instantiations we could see implicitly that complete answer aggregates also meet the motivation of making local dependencies explicit. We will now describe an alternative way of characterizing a complete answer aggregates being based on dependencies between bindings in answers that should make the last point more clear. With this point of view complete answer aggregates can be conceived as "folded" complete answer sets.

Definition 4.28 A *local dependency aggregate* for a simple tree query $Q = (\psi \wedge c, \vec{x})$ and a relational document structure $\mathcal{D}$ is an aggregate Agg so that

(LDA1) Agg_x contains a field $Agg_x[d]$ iff there exists an answer ν to Q and $\mathcal{D}$ with $\nu(x) = d$ and

(LDA2) $Agg_x[d]$ contains for a child y of x a pointer to a field $Agg_y[e]$ iff there exists an answer ν to Q and $\mathcal{D}$ with $\nu(x) = d$ and $\nu(y) = e$.

From this definition follows that there exists exactly one local dependency aggregate for a given query and relational document structure.

Lemma 4.29 *Let Q be a simple tree query and $\mathcal{D}$ a relational document structure. Then the complete answer aggregate for Q and $\mathcal{D}$ is the local dependency aggregate for Q and $\mathcal{D}$.*

Proof. Let Agg be the complete answer aggregate for Q and $\mathcal{D}$. We show that Agg is the local dependency aggregate for Q and $\mathcal{D}$ and begin with condition (LDA1): Let x be a query variable and $Agg_x[d]$ a field in Agg_x. Due to minimality of complete answer aggregates (Lemma 4.24) $Agg_x[d]$ belongs to an instantiation and therefore there exists an answer ν with $\nu(x) = d$. Now let ν be an answer with $\nu(x) = d$. Since ν can be obtained as an instantiation of Agg, the record Agg_x contains a field $Agg_x[d]$.
We show now condition (LDA2): Let x and y be query variables so that y is a child of x. Let $Agg_x[d]$ and $Agg_y[e]$ be fields in Agg_x and Agg_y so that $Agg_x[d]$ contains a pointer for y to $Agg_y[e]$. Due to minimality of complete answer aggregates, we can form an instantiation ν with $\nu(x) = d$ and $\nu(y) = e$. With ν we found an answer mapping x to d and y to e. Now let ν be an answer with $\nu(x) = d$ and $\nu(y) = e$. Since this answer can also be obtained from an instantiation of Agg, we know that Agg_x and Agg_y, respectively, contain fields $Agg_x[d]$ and $Agg_y[e]$ so that $Agg_x[d]$ has a pointer for y to $Agg_y[e]$.
Since the local dependency aggregate for Q and $\mathcal{D}$ is unique, the claim is proven now. □

Example 4.30 If we compare the complete answer set to the query and relational document structure in Example 4.10 with the resulting complete answer aggregate in Example 4.16, we can observe that the complete answer aggregate is the local dependency aggregate for the query. We can also see how a complete answer aggregate could be constructed out of the complete answer set, following exactly Definition 4.28, what supports the view of complete answer aggregates being "folded" complete answer sets.

In [MS00] complete answer aggregates were defined for more general query classes like DAG queries or graph queries via the notion of local dependency aggregates. For these more general query classes the goal of having a representation out of which the set of all answers can be generated in a simple way was not achieved: When we try to form an instantiation of a local dependency aggregate, backtracking, i.e. asearch in the complete answer aggregates, can be required. The reason for this is that Lemma 3.27, and therefore Lemma 4.7, do not hold for DAG or cyclic graph queries.

The discussion above should make clear that trees are a natural limit for structural complexity of queries that still allow to achieve both goals, making explicit local dependencies and obtaining all answers in a simple way, with the means of data structures similar to complete answer aggregates.

4.4 Complete Answer Formulae for Local Tree Queries

So far, we have introduced complete answer aggregates for the restricted class of simple tree queries only. In this section we briefly discuss how the same concept can be used for more general classes of queries. One important characteristics of the notion of a complete answer aggregates is the principle that the administrational information that is stored in a field $Agg_x[d]$ only concerns the possible instantiations of the children of the variable x in the query Q. This restriction can be interpreted as a form of locality. Since we do not want to give up the principle, the class of Q-local constraints seems to represent a natural limit for representation techniques based on the idea of a complete answer formulae. The characterizations of complete answer formulae for local tree queries obtained in this section will be used later when treating the special case of partially ordered tree queries.

Let Q be a local tree query. Suppressing all constraints of Q that are not Q-simple we obtain a simple tree query Q_s. Let Δ_{Q_s} be the unique complete answer formula for Q_s (cf. Theorem 4.11). Each subformula $\delta_x(d)$ of Δ_{Q_s} describes the set of possible instantiations of the descendants of x under the hypothesis that x is mapped to d. These instantiations respect Q-simple constraints, but not necessarily the suppressed Q-local constraints. To circumvent this problem we add a new *restrictor condition* to each formula $\delta_x(d)$ that guarantees that the instantiation of the children $y_1, \ldots, y_h$ of x_k in Q satisfies the Q-local constraints imposed on $\langle x_k, y_1, \ldots, y_h \rangle$ in Q. In principle the syntactic form of restrictor conditions is arbitrary, as long as they correctly encode Q-local constraints. For the sake of specificity we use an explicit enumeration of admissible instantiation tuples for $\langle y_1, \ldots, y_h \rangle$ in the following definitions. The following definition captures the syntactical form of an appropriate class of formulae, while the notion complete answer formula defined afterwards captures the semantics.

Definition 4.31 Let $Q = (\psi \wedge c, \vec{x})$ be a local tree query, let $x \in fr(Q)$. The set of *dependent Q-instantiation formulae for x* is inductively defined as follows. Assume

that $h_Q(x) = 0$. For each non-empty set $D_x \subseteq D$, the formula

$$\Delta_\epsilon^{\langle x \rangle} :\equiv \bigvee_{d \in D_x} x = d$$

is a dependent Q-instantiation formula for x. Now assume that $h_Q(x) > 0$. Let $\emptyset \neq D_x \subseteq D$. Let $\{y_1, \ldots, y_h\}$ denote the set of children of x in Q. For each $d \in D_x$ and each child y_i, let $\Delta_{\langle d \rangle}^{\langle x, y_i \rangle}$ be a dependent instantiation formula for y_i with set of target candidates $D_d^x(y_i)$. If $R_d^x(y_1, \ldots, y_h)$ is a non-empty subset of $D_d^x(y_1) \times \cdots \times D_d^x(y_h)$ such that for all $i = 1, \ldots, h$ and all $d_i \in D_d^x(y_i)$ there exists a tuple in $R_d^x(y_1, \ldots, y_h)$ where the i-th component is d_i ("contribution obligation"), then

$$\Delta_\epsilon^{\langle x \rangle} :\equiv \bigvee_{d \in D_x} (x = d \wedge \langle y_1, \ldots, y_h \rangle \in R_d^x(y_1, \ldots, y_h) \wedge \bigwedge_{i=1}^{h} \Delta_d^{\langle x, y_i \rangle})$$

is a dependent Q-instantiation formula for x. Besides the above formulae, there are no other dependent Q-instantiation formulae for x.

In the sequel, $R_d^x(y_1, \ldots, y_h)$ will be called the *restrictor set* of the subformula $x = d \wedge \langle y_1, \ldots, y_h \rangle \in R_d^x(y_1, \ldots, y_h) \wedge \bigwedge_{i=1}^{k} \Delta_d^{\langle x, y_i \rangle}$. The condition that restrictor sets are always non-empty ensures that partial instantiations of dependent instantiation formulae can be extended to total instantiations (see below). The second condition on restrictor sets, which will be called "*contribution obligation*" for the sake of reference, ensures that no target candidate $d_i \in D_d^x(y_i)$ is isolated, i.e. every target candidate $d_i \in D_d^x(y_i)$ contributes to at least one answer. As in the case of simple tree queries we use expressions $\Delta_{\langle d_0, \ldots, d_{k-1} \rangle}^{\langle x_0, \ldots, x_k \rangle}$ and $\delta_{\langle d_0, \ldots, d_k \rangle}^{\langle x_0, \ldots, x_k \rangle}$ for refering to subformulae of a dependent Q-instantiation formula $\Delta_\epsilon^{\langle x_0 \rangle}$.

Definition 4.32 Let Q be a local tree query. The set of *partial (total) instantiations* of a dependent Q-instantiation formula $\Delta_\epsilon^{x_0}$ is inductively defined as follows: for each subformula $\delta_{d_0}^{x_0}$ the mapping $\{\langle x_0, d_0 \rangle\}$ is a a partial instantiation of $\Delta_\epsilon^{x_0}$. Let $\delta_{\langle d_0, \ldots, d_k \rangle}^{\langle x_0, \ldots, x_k \rangle}$ be a subformula of $\Delta_\epsilon^{x_0}$ of the form

$$x_k = d_k \wedge \langle y_1, \ldots, y_h \rangle \in R_{d_k}^{x_k}(y_1, \ldots, y_h) \wedge \bigwedge_{i=1}^{h} \Delta_{d_k}^{\langle x_k, y_i \rangle}$$

(where $y_1, \ldots, y_h$ is the sequence of children of x_k in Q). Assume that ν is a partial instantiation of $\Delta_\epsilon^{x_0}$ such that $\{\langle x_i, d_i \rangle \mid i = 1, \ldots, k\} \subseteq \nu$ and ν does not instantiate any child y_i of x_k. For each tuple $\langle e_1, \ldots, e_h \rangle \in R_{d_k}^{x_k}(y_1, \ldots, y_h)$ the mapping $\nu \cup \{\langle y_i, e_i \rangle \mid 1 \leq i \leq h\}$ is a partial instantiation of $\Delta_\epsilon^{x_0}$. There are no other partial instantiations besides those defined by the above rules. A partial instantiation ν of $\Delta_\epsilon^{x_0}$ is a (total) *instantiation* iff ν contains a pair $\langle x, d \rangle$ for every $x \in fr(\Delta_\epsilon^{x_0})$.

Definition 4.33 Let Q be a local tree query, with root x_0. A dependent Q-instantiation formula $\Delta_\epsilon^{\langle x_0 \rangle}$ for x_0 is called a *complete answer formula* for Q iff each answer to Q is a total instantiation of $\Delta_\epsilon^{\langle x_0 \rangle}$ and vice versa.

The following theorem and its proof is a simple extension of Theorem 4.11:

Theorem 4.34 *For each local tree query Q a complete answer formula Δ_Q is unique modulo associativity and commutativity of "$\wedge$" and "$\vee$".*

Before we can prove Theorem 4.34 some preparation is needed.

Lemma 4.35 *Let Q be a local tree query, let $\Delta_\epsilon^{x_0}$ be a dependent Q-instantiation formula. Then every partial instantiation of $\Delta_\epsilon^{x_0}$ can be extended to a total instantiation of $\Delta_\epsilon^{x_0}$. The set of total instantiations of $\Delta_\epsilon^{x_0}$ is non-empty.*

Proof. Follows immediately from the fact that restrictor sets for variables x with $h_Q(x) > 0$ as well as arbitrary sets of target candidates are always non-empty. □

Lemma 4.36 *Let Q be a local tree query, let $\Delta_\epsilon^{\langle x_0\rangle}$ be a dependent Q-instantiation formula for x_0, let $d_0, \ldots, d_k$ be elements of D. Then the following conditions are equivalent:*

1. *$\Delta_\epsilon^{\langle x_0\rangle}$ has a subformula $\Delta_{\langle d_0,\ldots,d_{k-1}\rangle}^{\langle x_0,\ldots,x_k\rangle}$ where d_k is a target candidate for x_k,*
2. *$\Delta_\epsilon^{\langle x_0\rangle}$ has a subformula of the form $\delta_{\langle d_0,\ldots,d_k\rangle}^{\langle x_0,\ldots,x_k\rangle}$,*
3. *Q has formulae $x_0 \lhd^{(+)} x_1, \ldots, x_{k-1} \lhd^{(+)} x_k$ and $\{\langle x_i, d_i\rangle \mid 0 \le i \le k\}$ is a subset of a partial instantiation of $\Delta_\epsilon^{\langle x_0\rangle}$ that does not instantiate any child of x_k in Q.*

Proof. The equivalence "1 $\Leftrightarrow$ 2" is trivial. Implication "2 $\Rightarrow$ 3" follows from the contribution obligation mentioned after Definition 4.31 using induction on k. The converse direction "3 $\Rightarrow$ 2" is trivial. □

Proof of Theorem 4.34. Let $\Delta_\epsilon^{\langle x_0\rangle}$ and $\Lambda_\epsilon^{\langle x_0\rangle}$ be complete answer formulae for Q. Assume that $\Delta_\epsilon^{\langle x_0\rangle}$ has a subformula $\delta_{\langle d_0,\ldots,d_k\rangle}^{\langle x_0,\ldots,x_k\rangle}$. If x_k is a leaf of Q it follows from Lemmata 4.35 and 4.36 that $\Lambda_\epsilon^{\langle x_0\rangle}$ has a corresponding subformula $\lambda_{\langle d_0,\ldots,d_k\rangle}^{\langle x_0,\ldots,x_k\rangle}$. Assume now that $y_1, \ldots, y_h$ (for $h \ge 1$) denotes the set of children of x_k in Q. Let $\langle e_1, \ldots, e_h\rangle$ be an element of the restrictor set of $\delta_{\langle d_0,\ldots,d_k\rangle}^{\langle x_0,\ldots,x_k\rangle}$. Using Lemmata 4.36 and 4.35 we see that Q has an answer ν that maps x_i to d_i for $i = 1, \ldots, k$ and y_j to e_j for $j = 1, \ldots, h$. But then $\Lambda_\epsilon^{\langle x_0\rangle}$ must have a subformula $\lambda_{\langle d_0,\ldots,d_k\rangle}^{\langle x_0,\ldots,x_k\rangle}$ where $\langle e_1, \ldots, e_h\rangle$ is an element of the restrictor set, since otherwise answer ν could not be obtained as an instantiation of $\Lambda_\epsilon^{\langle x_0\rangle}$. By symmetry it follows that $\Delta_\epsilon^{\langle x_0\rangle}$ and $\Lambda_\epsilon^{\langle x_0\rangle}$ have corresponding subformulae of the form $\delta_{\langle d_0,\ldots,d_k\rangle}^{\langle x_0,\ldots,x_k\rangle}$ respectively $\delta_{\langle d_0,\ldots,d_k\rangle}^{\langle x_0,\ldots,x_k\rangle}$ with identical restrictor sets. Starting at the subformulae with maximal k it is then simple to prove by "inverse" induction that corresponding formulae $\delta_{\langle d_0,\ldots,d_k\rangle}^{\langle x_0,\ldots,x_k\rangle}$ and $\lambda_{\langle d_0,\ldots,d_k\rangle}^{\langle x_0,\ldots,x_k\rangle}$ are equal modulo associativity and commutativity of "$\wedge$". It follows that $\Delta_\epsilon^{\langle x_0\rangle}$ and $\Lambda_\epsilon^{\langle x_0\rangle}$ are equal modulo associativity and commutativity of "$\wedge$" and "$\vee$". □

The following lemma and its proof are again simple variants of the corresponding Lemma 4.12.

Lemma 4.37 *Let Δ_Q be a complete answer formula for the local tree query Q. Then two subformulae of Δ_Q of the form $\delta_{\langle d_0,\ldots,d_{k-1},d_k\rangle}^{\langle x_0,\ldots,x_{k-1},x_k\rangle}$ and $\delta_{\langle d'_0,\ldots,d'_{k-1},d_k\rangle}^{\langle x_0,\ldots,x_{k-1},x_k\rangle}$ are always identical modulo associativity and commutativity of "$\wedge$" and "$\vee$".*

Proof. It suffices to verify the following: if $\delta_{\langle d_0,\ldots,d_{k-1},d_k\rangle}^{\langle x_0,\ldots,x_{k-1},x_k\rangle}$ has a subformula of the form $\delta_{\langle d_0,\ldots,d_{k-1},d_k,d_{k+1},\ldots,d_{k+r}\rangle}^{\langle x_0,\ldots,x_{k-1},x_k,x_{k+1},\ldots,x_{k+r}\rangle}$, then $\delta_{\langle d'_0,\ldots,d'_{k-1},d_k\rangle}^{\langle x_0,\ldots,x_{k-1},x_k\rangle}$ has a subformula of the form $\delta_{\langle d'_0,\ldots,d'_{k-1},d_k,d_{k+1},\ldots,d_{k+r}\rangle}^{\langle x_0,\ldots,x_{k-1},x_k,x_{k+1},\ldots,x_{k+r}\rangle}$ and vice versa. Moreover, if $h_Q(x_{k+r}) > 0$, then the restrictor sets of both formulae are identical.

Let $\delta^{\langle x_0,\ldots,x_{k-1},x_k,x_{k+1},\ldots,x_{k+r}\rangle}_{\langle d_0,\ldots,d_{k-1},d_k,d_{k+1},\ldots,d_{k+r}\rangle}$ be a subformula of $\delta^{\langle x_0,\ldots,x_{k-1},x_k\rangle}_{\langle d_0,\ldots,d_{k-1},d_k\rangle}$. By Lemma 4.8, Δ_Q has partial instantiations that extend the mappings $\{\langle x_i,d_i\rangle \mid 0\leq i\leq k+r\}$ and $\{\langle x_i,d'_i\rangle \mid 0\leq i\leq k\}$ where $d_k=d'_k$. By Lemma 4.7 there exists an answer ν (resp. μ) of Q that extends the former (latter) partial instantiation. By Lemma 3.27 there exists an answer σ of Q that coincides with answer ν on the set of reflexive descendants of x_k and with answer μ on all other variables in $fr(Q)$. Answer σ extends the mapping $\{\langle x_i,d'_i\rangle \mid 0\leq i\leq k-1\}\cup\{\langle x_i,d_i\rangle \mid k\leq i\leq k+r\}$. But then $\delta^{\langle x_0,\ldots,x_{k-1},x_k\rangle}_{\langle d'_0,\ldots,d'_{k-1},d_k\rangle}$ must have a subformula of the form $\delta^{\langle x_0,\ldots,x_{k-1},x_k,x_{k+1},\ldots,x_{k+r}\rangle}_{\langle d'_0,\ldots,d'_{k-1},d_k,d_{k+1},\ldots,d_{k+r}\rangle}$. By symmetry the first condition mentioned above follows.
Assume that that both subformulae exist and that $h_Q(x_{k+r})>0$. Let $y_1,\ldots,y_h$ be the sequence of children of x_{k+r} and let $\langle e_1,\ldots,e_h\rangle$ be an element of the restrictor set of $\delta^{\langle x_0,\ldots,x_{k-1},x_k,x_{k+1},\ldots,x_{k+r}\rangle}_{\langle d_0,\ldots,d_{k-1},d_k,d_{k+1},\ldots,d_{k+r}\rangle}$. It follows from Lemma 4.8 and Lemma 4.7 that Q has an answer τ that extends the mapping $\{\langle x_i,d_i\rangle \mid 0\leq i\leq k+r\}\cup\{\langle y_i,e_i\rangle \mid i=1,\ldots,h\}$. Lemma 3.27 shows that there exists an answer ϵ of Q that extends the mapping $\{\langle x_i,d'_i\rangle \mid 0\leq i\leq k-1\}\cup\{\langle x_i,d_i\rangle \mid k\leq i\leq k+r\}\cup\{\langle y_i,e_i\rangle \mid i=1,\ldots,h\}$. But then $\langle e_1,\ldots,e_h\rangle$ must be an element of the restrictor set of $\delta^{\langle x_0,\ldots,x_{k-1},x_k,x_{k+1},\ldots,x_{k+r}\rangle}_{\langle d'_0,\ldots,d'_{k-1},d_k,d_{k+1},\ldots,d_{k+r}\rangle}$. By symmetry it follows that both subformulae mentioned above have the same restrictor set. □

On the basis of the lemma we may write subformulae $\delta^{\langle x_0,\ldots,x_{k-1},x\rangle}_{\langle d_0,\ldots,d_{k-1},d\rangle}$ in the form $\delta_x(d)$ and subformulae of the form $\Delta^{\langle x_0,\ldots,x_{k-1},x,y_i\rangle}_{\langle d_0,\ldots,d_{k-1},d\rangle}$ in the form $\Delta_{x,y_i}(d)$.

Constraints as Restrictor Conditions

At the end of this section we want to show that the restrictor conditions of a complete answer formula for a local tree query Q are equivalent to the Q-local (non Q-simple) constraints imposed on the respective variables in Q. A definition is needed before.

Definition 4.38 Let Q be a local tree query and let $y_1,\ldots,y_h$ be the children of a query variable x. A sequence of nodes $\langle e_1,\ldots,e_h\rangle$ *satisfies* a constraint $r(x,y_{i_1},\ldots,y_{i_r})$ (where $\{y_{i_1},\ldots,y_{i_r}\}\subseteq\{y_1,\ldots,y_h\}$) relative to d iff $r_{\mathcal{D}}(d,e_{i_1},\ldots,e_{i_r})$ holds in $\mathcal{D}$.

Lemma 4.39 *Let $Q=(\psi\wedge c,\vec{x})$ be a tree query, $\delta_x(d)$ a subformula of the complete answer formula Δ_Q for Q and $y_1,\ldots,y_h$ be the children of x. In the situation of Definition 4.38, let R denote the restrictor set of $\delta_x(d)$, for $i=1,\ldots,h$ let D_i be the set of target candidates for y_i in $\Delta_{x,y_i}(d)$. Then R is the set of all tuples $\langle e_1,\ldots,e_h\rangle\in D_1\times\cdots\times D_h$ where $\langle e_1,\ldots,e_h\rangle$ satisfies all non Q-simple constraints $r(x,y_{i_1},\ldots,y_{i_r})$ (where $\{y_{i_1},\ldots,y_{i_r}\}\subseteq\{y_1,\ldots,y_h\}$) of c relative to d.*

Proof. Let $\langle e_1,\ldots,e_h\rangle\in R$. By definition, $\langle e_1,\ldots,e_h\rangle\in D_1\times\cdots\times D_h$. It follows from Lemma 4.35 and Lemma 4.36 that Q has an answer that extends the mapping $\{\langle x_i,d_i\rangle \mid i=1,\ldots,k\}\cup\{\langle y_i,e_i\rangle \mid i=1,\ldots,h\}$. This shows that $\langle d_k,e_1,\ldots,e_h\rangle$ satisfies all constraints $r(x_k,y_{i_1},\ldots,y_{i_r})$ in c where $\{y_{i_1},\ldots,y_{i_r}\}\subseteq\{y_1,\ldots,y_h\}$.
Conversely let $\langle e_1,\ldots,e_h\rangle\in D_1\times\cdots\times D_h$, assume that $\langle d_k,e_1,\ldots,e_h\rangle$ satisfies all non Q-simple constraints $r(x_k,y_{i_1},\ldots,y_{i_r})$ in c where $\{y_{i_1},\ldots,y_{i_r}\}\subseteq\{y_1,\ldots,y_h\}$. Assume, to get a contradiction, that $\langle e_1,\ldots,e_h\rangle\notin R$. Replacing R with $R\cup\{\langle e_1,\ldots,e_h\rangle\}$ we would get a dependent Q-instantiation formula with a larger set of instantiations where still each instantiation is an answer to Q. In fact, since we did not modify any set of target candidates the new Q-instantiation formula leads to instantiations that satisfy all Q-simple constraints and $\mathcal{L}$-formulae of the query. This would mean that Δ_Q is not a complete answer formula. □

Lemma 4.39 shows that we may use the Q-local constraints itself as restrictor formulae. Therefore it is easy to see that a complete answer formula for a given

local tree query always exists if the query has at least one answer. For the special case of ordered tree queries we shall give an algorithm for computing a complete answer formula in the next section.

Though this form of representation of restrictor formulae with the Q-local constraints itself seems natural and yields a compact representation it has the disadvantage that it might be far from obvious which tuples of target candidates for the children variables actually satisfy the relevant set of Q-local constraints. On the other hand, a naive enumeration of all elements of the restrictor set might lead to serious space problems, something that we wanted to avoid with the use of answer aggregates. Since the optimal representation of restrictor sets depends on the concrete type of Q-local constraints that are used we do not continue the discussion on this general level. Instead we treat the special case of ordered tree queries in more detail.

4.5 Complete Answer Aggregates for Partially Ordered Tree Queries

Partially ordered tree queries represent a special subclass of local tree queries, hence all results of the previous section can be applied. We may now give our second major result.

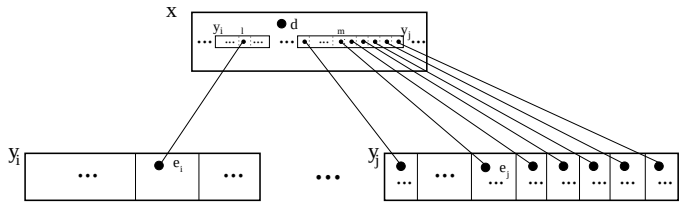
Theorem 4.40 *For each partially ordered tree query Q there exists a complete answer formula Δ_Q which is unique modulo associativity and commutativity of "$\wedge$" and "$\vee$".*

Proof. The uniqueness part is a special instance of Theorem 4.34. In Section 5 we give an algorithm that computes a complete answer formula for Q. □

It remains to find a suitable representation for restrictor sets that can be used to immediately enumerate possible instantiations and leads to reasonable space requirements.

Consider a partially ordered tree query Q. Let Q_s denote the simple tree query that is obtained by suppressing all left-to-right ordering constraints and let Agg_{Q_s} be the complete answer aggregate for Q_s. We assume that the fields $Agg_y[e]$ of each record Agg_y are ordered via pre-order relation $<_p^D$ of the nodes e. Similarly pointer lists of the form $Agg_x[d, y]$ are ordered following the ordering of their address fields. These assumptions will help to find a simple encoding for left-to-right ordering constraints.

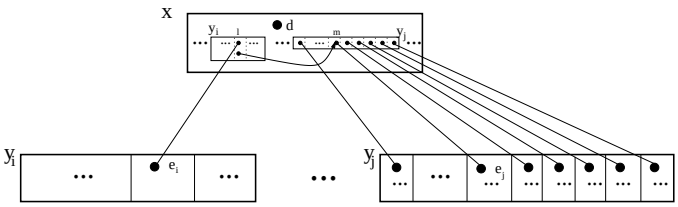
To illustrate the idea, consider a pointer $Agg_x[d, y_i[l]]$ of Agg_{Q_s} pointing to a field $Agg_{y_i}[e_i]$ as indicated in the figure below. Assume that Q has a constraint $y_i <_{lr} y_j$. Now let $Agg_x[d, y_j[m]]$ be the left-most pointer in $Agg_x[d, y_j]$ with an address field $Agg_{y_j}[e_j]$ such that $e_i <_{lr}^D e_j$.[2]



In this situation, all pointers $Agg_x[d, y_j[m']]$ with index $m' \geq m$ have address fields $Agg_{y_j}[e_{m'}]$ such that $e_i <_{lr}^D e_{m'}$, and these are the only pointers of $Agg_x[d, y_j]$ with

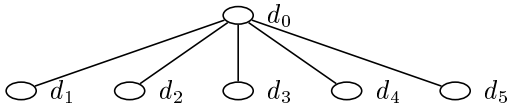
[2] For the sake of simplicity we assume that such a pointer exists. The discussion of the other case, where we have to erase $Agg_x[d, y_i[l]]$, is postponed to Section 5.3.

an address field satisfying this condition. In fact, by our ordering convention for fields we have $e_j \leq_p^D e_{m'}$ for each such m' and, since $e_i <_{lr}^D e_j$, Lemma 3.4 shows that $e_i <_{lr}^D e_{m'}$. This shows that all pointers have the required property. By choice of m, no other pointer can satisfy the condition. Hence, in order to encode the left-to-right ordering constraint $y_i <_{lr} y_j$ subject to the choices $x = d$ and $y_i = e_i$ it suffices to introduce a "horizontal" pointer from $Agg_x[d, y_i[l]]$ to $Agg_x[d, y_j[m]]$ as indicated in the following figure.



The pointer is interpreted in the following way. When instantiating x with d and y_i with e_i, we may use exactly the pointers $Agg_x[d, y_j[m]]$, $Agg_x[d, y_j[m+1]], \ldots$ for instantiating y_j. Of course, when we proceed in this way we have to introduce horizontal pointers for all possible instantiation values of variables and all left-to-right ordering constraints. We illustrate the complete picture with an example:

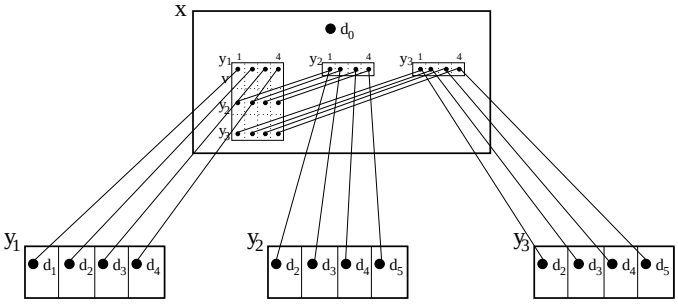
Example 4.41 Let $\mathcal{D}$ have the following form (we ignore labels and textual contents) where the left-to-right ordering between the children of d_0 is as depicted in the figure.



The complete answer aggregate for the partially ordered tree query Q of the form

$$(x \lhd y_1 \wedge x \lhd y_2 \wedge x \lhd y_3 \wedge y_1 <_{lr} y_2 \wedge y_1 <_{lr} y_3, \langle x, y_1, y_2, y_3 \rangle)$$

is the following object.



Since there are two left-to-right ordering constraints for y_1, $y_1 <_{lr} y_2$ and $y_1 <_{lr} y_3$, with each vertical pointer of $Agg[d_0, y_1]$ (row "v") we associate two horizontal pointers (rows "y_2" and "y_3"). When instantiating y_1 with d_2, for example, we may instantiate y_2 using the pointers to d_3, d_4 or d_5, and similarly for y_3.

Definition 4.42 Let $Q = (\psi \wedge c, \vec{x})$ be a partially ordered tree query. An *aggregate* for Q is a family Agg_Q of records, $\{Agg_x \mid x \in \vec{x}\}$. Each record Agg_x is composed of an ordered sequence of fields $Agg_x[d]$, the ordering is given by the pre-order relationship of nodes d in $\mathcal{D}$. For each child y_i of x in Q, the field $Agg_x[d]$ contains a two-dimensional array $Agg_x[d, y_i]$. With $Agg_x[d, y_i[l, *]]$ we denote the l-th column.

1. The first entry $Agg_x[d, y_i[l, v]]$ of $Agg_x[d, y_i[l, *]]$ is a "vertical" pointer, i.e., a pointer to a field of the form $Agg_{y_i}[e_i]$. Node e_i is called the *address node* of $Agg_x[d, y_i[l, *]]$. Address nodes of distinct columns are distinct.

2. For each left-to-right ordering constraint $y_i <_{lr} y_j$ of Q there is one additional entry (*horizontal pointer*) $Agg_x[d, y_i[l, y_j]]$ in $Agg_x[d, y_i[l, *]]$ that represents a pointer to the first column $Agg_x[d, y_j[m, *]]$ with an address node e_j such that $e_i <^D_{lr} e_j$. There are no other entries in $Agg_x[d, y_i[l, *]]$.

Definition 4.43 Let $Q = (\psi \wedge c, \vec{x})$ be a partially ordered tree query, let Δ_Q denote the complete answer formula for Q. A *complete answer aggregate* for Q is an aggregate $\{Agg_x \mid x \in \vec{x}\}$ for Q that satisfies the following conditions.

(CAA1) Agg_x has a field $Agg_x[d]$ iff Δ_Q has a subformula $\delta_x(d)$,

(CAA2) an array $Agg_x[d, y_i]$ has a vertical pointer with address field $Agg_{y_i}[e]$ iff $\delta_x(d)$ has a subformula $\delta_{y_i}(e)$.

Ignoring the trivial case of an unsatisfiable query it is again easy to see that a complete answer formula Δ_Q for a partially ordered tree query Q uniquely determines the corresponding complete answer aggregate Agg_Q. Conversely, given a complete answer aggregate Agg_Q we may reconstruct the complete answer formula Δ_Q in the following way: to obtain Δ_Q

- read the record Agg_x of the root x of Q as the disjunction of the formulae associated with the fields $Agg_x[d]$,
- associate with each field $Agg_x[d]$ the conjunction of $x = d$ with the *horizontal pointer condition* (see below) and the formulae associated with the lists of pointers,
- associate with each list of pointers $Agg_x[d, y]$ of a given field $Agg_x[d]$ the disjunction of the formulae associated with the address fields of the pointers.

Assume that $Agg_x[d]$ has the pointer arrays $Agg_x[d, y_1], \ldots, Agg_x[d, y_h]$ for the children $y_1, \ldots, y_h$ of x in Q. The *horizontal pointer condition* has the form $(y_1, \ldots, y_h) \in R^x_d(y_1, \ldots, y_h)$ where $R^x_d(y_1, \ldots, y_h)$ contains all tuples $(e_1, \ldots, e_h)$ that satisfy the following conditions:

1. there exist pointer columns $Agg_x[d, y_1[l_1, *]], \ldots, Agg_x[d, y_h[l_h, *]]$ where vertical pointers have address nodes $e_1, \ldots, e_h$,

2. for each horizontal pointer $Agg_x[d, y_i[l_i, y_j]]$ with address $Agg_x[d, y_j[k, *]]$ we have $k \leq l_j$.

Clearly, the sets $R^x_d(y_1, \ldots, y_h)$ are exactly the restrictor sets defined in Definition 4.31.

We define the notion of instantiations of complete answer aggregates similar to simple tree queries:

Definition 4.44 A mapping $\nu : fr(Q) \to D$ is an *instantiation* of the aggregate $Agg_Q = \{Agg_x \mid x \in fr(Q)\}$ iff the following conditions are satisfied:

(INST1) Agg_x has a field $Agg_x[\nu(x)]$ for all $x \in fr(Q)$,

(INST2) if y is a child of x in Q, if $d = \nu(x)$ and $e = \nu(y)$, then e is an address node of a vertical pointer of a column $Agg_x[d, y[l, *]]$ of the array $Agg_x[d, y]$,

(INST3) if $\nu(x) = d$ and Agg_Q has a horizontal pointer $Agg_x[d, y_i[l, y_j]]$ with address $Agg_x[d, y_j[m, *]]$, if $e_i := \nu(y_i)$ is the address node of the vertical pointer $Agg_x[d, y_i[l, v]]$, then $e_j := \nu(y_j)$ is the address node of a vertical pointer $Agg_x[d, y_j[m', v]]$ such that $m' \geq m$.

We say that each field $Agg_x[\nu(x)]$ *belongs to* the instantiation ν. Similarly each pointer column $Agg_x[d, y[l, *]]$ of the form described in (INST2) is said to belong to ν.

As was the case for simple tree queries, every instantiation of a complete answer aggregate for a partially ordered tree query Q and a relational document structure $\mathcal{D}$ is an answer to Q and $\mathcal{D}$.

Minimality of Complete Answer Aggregates

Complete answer aggregates for partially ordered tree queries are minimal representations of the set of all answers in the in the same sense as complete answer aggregates for simple tree queries:

Definition 4.45 An aggregate Agg is called *minimal* iff all its fields and pointers belong to an instantiation of Agg.

Lemma 4.46 *The complete answer aggregate Agg for a partially ordered tree query Q and a relational document structure $\mathcal{D}$ is minimal.*

Proof. Since Agg directly corresponds to the complete answer formula Δ_Q for Q and $\mathcal{D}$ the claim follows directly from Lemma 4.47. □

The following lemma (corresponding to Lemma 4.13 in the simple tree query case) is a simple consequence of Lemma 4.35 and needed for the proof of Lemma 4.46.

Lemma 4.47 *For every variable x in a simple tree query Q and every atom $x = d$ in a complete answer formula Δ_Q for Q and a relational document structure $\mathcal{D}$, there exists a total instantiation ν of with $\nu(x) = d$.*

Before we show how to compute a complete answer aggregate for a partially ordered tree query we want to give an upper bound for the size.

Size of Complete Answer Aggregates

Remark 4.48 Let $Q = (\psi \wedge c, \vec{x})$ be a partially ordered tree query, let Agg denote the complete answer aggregate for Q. To each pointer $p = Agg_x[d, y_i[l, s]]$ of Agg we assign a unique triple $(L_1(p), L_2(p), L_3(p))$ as follows.

- $L_1(p)$ is the address node of the vertical pointer $Agg_x[d, y_i[l, v]]$ of the column $Agg_x[d, y_i[l, *]]$ of p.
- We define $L_2(p) := d$. Recall that d is always an ancestor of $L_1(p)$.
- $L_3(p)$ is the following atomic subformula/constraint of Q: if $s = v$ (p is a vertical pointer) then $L_3(p) := x \lhd^{(+)} y_i$ is the formula of Q that expresses that y_i is a child of x.
  If $s = y_j$, then $L_3(p)$ is the left-to-right ordering constraint $y_i <_{lr} y_j$.

Clearly distinct pointers are mapped to distinct triples. Hence the total number of pointers of Agg is bounded by the number of possible triples. This yields a general bound $|D| \cdot h_{\mathcal{D}} \cdot |Q|$ for tree queries, and a bound $|D| \cdot |Q|$ for data-anchored queries in relational document structures. This can be seen as follows. There are $|D|$ possibilities for $L_1(p)$. $L_2(p)$ must be an ancestor (a parent for rigid queries) of

$L_1(p)$. If $L_1(p)$ is fixed, there are $h_{\mathcal{D}}$ possibilities for $L_2(p)$, and just one possibility for data-anchored queries. Since $L_3(p)$ is an atomic subformula of Q, there are $|Q|$ possibilities for $L_3(p)$.

It follows that the bounds for the size of a complete answer aggregate that we obtained for simple tree queries hold for partially ordered tree queries as well:

Theorem 4.49 *Let $\mathcal{D}$ be an ordered relational document structure and let Q be a partially ordered tree query. Then the size of the complete answer aggregate for Q is of order $\mathcal{O}(|D| \cdot h_{\mathcal{D}} \cdot |Q|)$. If Q is data-anchored in $\mathcal{D}$, then the size of the complete answer aggregate for Q and $\mathcal{D}$ is of order $\mathcal{O}(|D| \cdot |Q|)$.*

4.6 Complete Answer Aggregates as Relational Document Structures

As pointed out in the discussion in Section 2.2 it is useful to have a formalism where answers to queries can be conceived as relational document structures themselves. In Section 3.3.6 we already presented a very simple mechanism to conceive answers as relational document structure. In this section we will present a more sophisticated way that makes use of the graph structure present in complete answer aggregates. We restrict this discussion to data-anchored queries in relational document structures. For the full class of tree queries it is not possible to achieve a simple result since in this case the complete answer aggregates do not necessarily have a tree or forest structure.

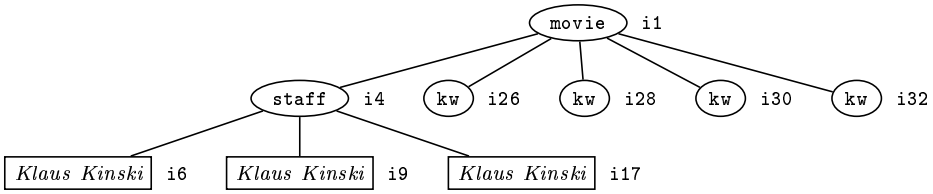
Consider the aggregate in Example 3.31. We can observe that the pointers impose a forest structure on the set of target candidates. Lemma 4.25 raises this observation to a formal proposition. We can therefore define an aggregate structure in the following way:

Definition 4.50 Let Q be a data-anchored query in a relational document structure $(\mathcal{D} = D_S, D_T, \rightharpoonup, Lab, Txt, I)$ and Agg be the complete answer aggregate for Q and $\mathcal{D}$. Then the *aggregate structure* for Agg $\mathcal{D}_{Agg} = (D_{(S,Agg)}, D_{(T,Agg)}, \rightharpoonup_{Agg}, Lab_{Agg}, Txt_{Agg}, I_{Agg})$ is a relational document structure defined in the following way:

- $D_{(S,Agg)}$ contains all structural nodes $d \in D_S$ so that $Agg_x[d]$ is a target candidate $Agg_x[d]$ in Agg.
- $D_{(S,Agg)}$ contains all text nodes $d \in D_T$ so that $Agg_x[d]$ is a target candidate $Agg_x[d]$ in Agg.
- $\langle d, e\rangle \in \rightharpoonup_{Agg}$ iff there exist target candidates $Agg_x[d]$ and $Agg_y[e]$ in Agg so that there is a vertical pointer from $Agg_x[d]$ to $Agg_y[e]$.
- $Lab_{Agg}(d) = Lab(d)$ and $Txt_{Agg}(d) = Txt(d)$ for all nodes $d \in D_{S,Agg} \cup D_{T,Agg}$.
- $\langle d, e\rangle \in I_{Agg}(r)$ iff $\langle d, e\rangle \in I(r)$.

The notion of aggregate structures is well-defined since Lemma 4.25 guarantees that $\rightharpoonup_{Agg}$ imposes a forest structure upon $D_{(S,Agg)} \cup D_{(T,Agg)}$.

Example 4.51 The complete answer aggregate in Example 4.16 defines, according to the above definition, the following aggregate structure:



Analogously to the idea elaborated in Section 4.3 of complete answer aggregates being folded complete answer sets, we can conceive aggregate structures as folded answer structures as defined in Section 3.3.6.

A more interesting perspective presents the aggregate structure as a view upon the relational document structure: The aggregate structure contains a subset of the nodes in the relational document structure and inherits all relations. If soft edges are used in the query, a sequence of edges in the relational document structure may be collapsed into a single edge in the aggregate structure.

As for answer structures we have a similar kind of idempotence for aggregate structures: All answers to a data-anchored query Q in a relational document structure $\mathcal{D}$ are also answers to Q and $\mathcal{D}_{Agg}$. If Q is rigid, the answer aggregate Agg for Q and $\mathcal{D}$ is also the complete answer aggregate for Q and $\mathcal{D}_{Agg}$.

Chapter 5

Computing Complete Answer Aggregates

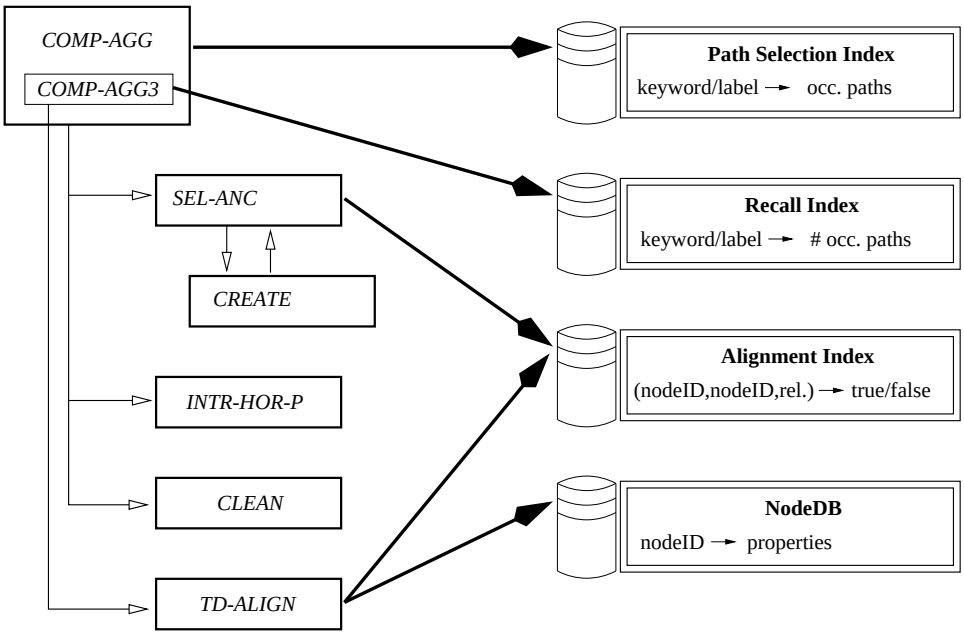
In this section we show how the complete answer aggregate for a partially ordered tree query and a relational document structure can be computed and prove the following central result:

Theorem 5.1 *Let Q be a partially ordered tree query, let $\mathcal{D}$ be a relational document structure. Then it is possible to compute a complete answer aggregate for Q in time $\mathcal{O}(|Q| \cdot |D| \cdot h_{\mathcal{D}} \cdot log(|D|))$ and space $\mathcal{O}(|Q| \cdot |D| \cdot h_{\mathcal{D}})$. If Q is data-anchored in $\mathcal{D}$ then we can compute the complete answer aggregate in time $\mathcal{O}(|Q| \cdot |D| \cdot log(|D|))$ and space $\mathcal{O}(|Q| \cdot |D|)$.*

Before we describe the algorithm we give in the first section a short overview over the involved index structures and the main procedures. The algorithm uses special index structures for accessing the relational document structure that we describe in more detail in the second section. In the next section we characterize fields and pointer columns of aggregates that can not contribute to any instantiation. In the algorithm, these fields and pointer columns will be eliminated by a dedicated sub-procedure. Hereafter, the algorithm itself together with its sub-procedures is described, followed by the soundness and completeness proof. In the last part of this chapter we discuss complexity and implementation issues and make some remarks on related work.

5.1 Overview

This section will provide a short overview over the algorithm and index structures used in the algorithm. The details are to be found in the following sections. The basic algorithm for evaluating partially ordered tree queries on relational document structures as described in Section 5.4.2 consists of five procedures: *COMP-AGG*, *SEL-ANC*, *CREATE*, *INTR-HOR-P*, and *CLEAN*. For the optimizations described in Section 5.4.3 a sixth procedure, *TD-ALIGN* is used. The algorithm uses four index structures for accessing the relational document structure, the path selection index, the alignment index, and in the optimized version additionally the recall index and the node database. The overall dependency structure of the algorithm can be seen in the following illustration:



The procedure *COMP-AGG*[1] triggers the bottom-up query evaluation with an empty aggregate. It decomposes the query into paths and calls for the keywords or labels in the leaf of each query path the path selection index. The path selection index returns a set of paths in the relational document structure that lead to a database node matching the keyword or label. In order to match the returned database paths with the respective query path the procedure *SEL-ANC* is called for every query path. This procedure enters matching database nodes into the respective slots of the aggregate (that are constructed on the fly if necessary). In order to detect pairs of nodes that do not meet the requirements imposed by the query the alignment index is queries by *SEL-ANC*. The procedures *SEL-ANC* and *CREATE* wander through the query path and database paths in parallel in a double-recursive way. If all query paths are treated this way a call to the procedure *INTR-HOR-P* adds horizontal pointers to the fields in the aggregate, if order constraints are used. As a last step of the basic algorithm the procedure *CLEAN* removes ("isolated") target candidates and pointers from the aggregate that can not contribute to solutions. These were identified in the previous steps. The removal of target candidates may trigger removal of further target candidates. The resulting aggregate is the complete answer aggregate for the query and relational document structure.

In the optimized version the procedure *COMP-AGG3* calls the recall index in order to detect query paths where a top-down evaluation is more efficient, since the database path sets returned by the path selection index would be too big. Database path sets small enough are treated in the way described above. If a database path set corresponding to a query path is too big, it is not used (not even loaded into main memory) and a top-down alignment is triggered by a call to the procedure *TD-ALIGN*. This procedure matches the query path with the nodes in the relational document structure and fills the slots of the aggregate in a similar way as the procedures *SEL-ANC* and *CREATE*. The differences are that alignment begins at the root and wanders downwards in the query path and relational document structure, and that the node database (a persistent data structure storing information about nodes, e.g. parent and children) has to be queried several times in order to

[1] Most procedures of the algorithm are presented in different versions, e.g. *COMP-AGG1* or *COMP-AGG2*. If we use the name of the procedure without a number we refer to all versions of the procedure.

follow the paths, since in this case we have no explicit set of database paths that shall be matched. If all query paths are treated with either *SEL-ANC/CREATE* or *TD-ALIGN*, the procedures *INTR-HOR-P* and *CLEAN* are called as described in the basic version.

We will now describe the used index structures in more detail.

5.2 Index architecture

The algorithm for computing a complete answer aggregate for a given query uses two basic index structures for $\mathcal{D}$: the path selection index and the alignment index. For the optimized version another index structure, the recall index, and a persistent data structure, the node database, storing information about the document nodes is used. In order to facilitate the index access we shall assume that $\mathcal{D}$ is an *ordered* relational document structure. Even if such an ordering does not exist a priori, an artificial ordering can always be imposed on $\mathcal{D}$. This turns out to be advantageous.

Each formula of the form $w \propto x$ ($w \in K$), $M(x)$ ($M \in \Gamma$) or $r(x)$ ($r \in \mathcal{R}$) will be called a *unary index formula*, formulae of the form $r(x, y)$ ($r \in R$) are called *binary index formulae*[2]. When we abstract from the variable $x \in X$ that is used we talk about (unary resp. binary) *index predicates*. Note that formulae of the form $x \lhd y, x \lhd^+ y$ as well as left-to-right ordering constraints are *not* treated as index formulae. The motivation for this distinction is the following: we assume that the information that describes the tree structure of the database (i.e., the actual set of nodes, children relationship, left-to-right ordering) is stored separately in a node database from.

Overall the algorithm accesses three persistent data structures that will be described here: a node database, a path selection index for unary predicates, and an alignment index for binary predicates. For some optimizations, a fourth persistent data structure, the recall index, is used.

5.2.1 Node Database

The node database stores the relational document structure in a table. It implements mappings from nodes (i.e. their identifiers) to their parents, children, labels, textual contents, and unary relations like attributes. The pre-order relation of nodes is encoded via node identifiers, cf. Remark 5.18. This means that the node database establishes an exhaustive (apart from binary relations, that are stored in the alignment index only) representation of the relational document structure in the form of a relational database. Nonetheless, to make query evaluation perform efficiently, we need additional index structures. The node database will be used in the algorithm for *navigational* tasks only, when the path selection index cannot perform well.

5.2.2 Path Selection Index

The path selection index is a *constructive* index, mapping appropriate unary predicates to sets of occurrences. It implements a mapping for each kind of unary predicate: $w \propto \_$ for terms $w \in \Sigma^*$, unary relations r for $r \in \mathcal{R}$, and M for labels $M \in \Gamma$.

Occurrences

In Information Retrieval an occurrence (describing the occurrence of a search term) is defined as a pair of document and offset (of the respective term inside the doc-

[2]Theorem 5.1 and the algorithm to be described below refer to partially ordered tree queries only, therefore we can restrict ourselves to unary and binary formulae.

ument). This definition of occurrence is insufficient for our purpose, since we need a structural rather than a flat notion of occurrence. Therefore we define an *occurrence* as the path leading from the leaf containing the search term to the root of the respective document, plus the offset. E.g. an occurrence of the term *Marceau* in the relational document structure in Appendix B.2 is $\langle \texttt{g1}, \texttt{g4}, \texttt{g7}, \texttt{g14}, \texttt{g15} \rangle : 2$, where $\langle \texttt{g1}, \texttt{g4}, \texttt{g7}, \texttt{g14}, \texttt{g15} \rangle$ is the path from the root of the respective document to the leaf containing the term *Marceau* and 2 indicates that *Marceau* is the second term in that leaf. Of course, this increases the size of the path selection index a lot, but lets the system perform more efficiently.[3]
A alternative approach would be to define an occurrence as a pair of the leaf containing the search term and the offset inside the leaf.Our algorithm compares the path leading to an occurrence returned by the path selection index to a path pattern in the query. This means that the node database has to be used to navigate from the leaf (the returned occurrence) to the root in order to obtain the corresponding path. We can see that this approach leads to a high number of page accesses to the node database for retrieving the paths from the leaves to the root, resulting in a bad efficiency due to the high I/O costs.

The following definition of an inverted partial document path captures the notion of occurrences.

Definition 5.2 An *inverted partial document path* is a non-empty sequence of nodes $\langle d_i, d_{i-1}, \ldots, d_0 \rangle$ such that $\langle d_0, \ldots, d_i \rangle$ is a partial document path. The *initial node* of an inverted partial document path $\langle d_i, d_{i-1}, \ldots, d_0 \rangle$ is the bottom-most node d_i. An *inverted query path* has the form $\langle x_k, \ldots, x_0 \rangle$ where $\langle x_0, \ldots, x_k \rangle$ is a path of Q.

An inverted partial document path will simply be called an *inverted document path*. The path selection index contains for each unary index predicate p a list Π_p of inverted document paths. Π_p is assumed to be ordered via pre-order relationship of initial nodes. The lists Π_p are assumed to be "sound" and "complete" in the following sense:

Remark 5.3 A node $d \in D$ is an initial node of an inverted document path in Π_p for a unary index predicate p if and only if d satisfies the predicate p in $\mathcal{D}$, i.e. $\mathcal{D} \models p[d]$.

Clearly the number of inverted document paths is bounded by the number n of nodes in the relational document structure. For distinct unary index predicates p and p' the intersection of the lists Π_p and $\Pi_{p'}$ can be computed in time $\mathcal{O}(n)$ using a simultaneous traversal of Π_p and $\Pi_{p'}$ along $<_p^D$. This can be generalized to finite intersections.

Lemma 5.4 *Let $p_1, \ldots, p_r$ be unary index predicates. The intersection $\Pi_{p_1} \cap \ldots \cap \Pi_{p_k}$ can be computed in time $\mathcal{O}(r \cdot n)$.*

Given the query Q, each call to the path selection index will be triggered by an inverted query path $\langle x_k, \ldots, x_0 \rangle$. In order to simplify the presentation of the following algorithm we shall assume that the index access directly yields the intersection $\Pi_{p_1} \cap \ldots \cap \Pi_{p_r}$ where $p_1(x_k), \ldots, p_r(x_k)$ is the complete list of unary index formulae for variable x_k in Q. From Lemma 5.4 we get

Lemma 5.5 *Let q be the number of atoms in a query Q and n the number of nodes in a relational document structure $\mathcal{D}$. The total time-complexity for the access to the path selection index of $\mathcal{D}$ for a query Q is bounded by $\mathcal{O}(q \cdot n)$.*

[3] In order to limit the increase of index growth we use a two step storage: Every path has an identifier and is stored separately in a table mapping the identifier to the actual database nodes. An occurrence is now a pair of path identifier and offset.

In real-life situations we will not call the path selection index for every unary index predicate. As we will see in the description of the algorithm, the path selection index will only be used in appropriate cases, when the number of paths returned is small enough.

5.2.3 The Alignment Index

The algorithm will check if an inverted document path π_D that results from the call to the path selection index for an inverted query path $\pi_Q = \langle x_k, \ldots, x_0 \rangle$ is conform with the conditions that are imposed on the variables $x_0, \ldots, x_k$ in Q. The check is organized as a bottom-up alignment process that removes document paths π_D not conforming to the requirements imposed by the query. This latter process is supported by the alignment index. We call it therefore a *destructive* index.

Remark 5.6 For theoretical estimation of the time-complexity of the algorithm to be described in the following section the following assumption will be made.

1. for each node d and each unary index predicate $p \in \mathcal{R}$ it is possible to check in constant time if $\mathcal{D} \models p[d]$, and

2. for each pair of nodes $d_i \rightharpoonup^+ d_j$ and each binary index predicate $r \in \mathcal{R}$ it is possible to check in constant time if $\mathcal{D} \models r[d_i, d_j]$.

We say that the pair $\langle d_i, d_{i-1} \rangle$ satisfies the index formulae for $\langle y, x \rangle$ iff d_{i-1} satisfies the unary index formulae for x (i.e. labeling and unary constraints) and $\langle d_i, d_{i-1} \rangle$ satisfy the binary constraints for $\langle y, x \rangle$. For unary index predicates p the required test can be implemented by assigning to p an array L_p of all nodes of D that satisfy p. From a theoretical point of view, a similar approach is possible for binary index predicates r, just using two-dimensional arrays, L_r.

Surely this strategy is not optimal in concrete cases. However, since all binary index predicates are "generic" relations $r \in \mathcal{R}$ is seems hard to suggest a better approach that works in full generality.

Since we assume that the number of distinct index predicates is finite and constant the above assumptions lead to the following result.

Lemma 5.7 *Given a pair of document nodes (d_i, d_j) where d_j is an ancestor of d_i, and a set of index predicates, P, it is possible to check in constant time if d_j satisfies all unary predicates in P and if (d_i, d_j) satisfies all binary index predicates in P.*

5.2.4 The Recall Index

As pointed out in Section 5.2.2, we do not use the path selection index if the set Π_p of paths returned by the path selection index for a unary index predicate p is too big. In order to detect these situations, we use a third index structure that helps to speed up query evaluation. The recall index returns for every unary index predicate p the size of the set Π_p containing all inverted document paths so that Remark 5.3 holds. The recall index is a *supporting index* since it supports the algorithm in efficiently using the other index structures. As for the path selection index, we generalize this and say that the recall index returns for an inverted query path $\langle x_k, \ldots, x_0 \rangle$ the size of the intersection $\Pi_{p_1} \cap \ldots \cap \Pi_{p_r}$ where $p_1(x_k), \ldots, p_r(x_k)$ is the complete list of unary index formulae for variable x_k in Q. Obviously the recall index is not an independent index on its own but can be implemented as a simple additional functionality within the path selection index.

5.2.5 Implementation Aspects

The index structures have been described so far as abstract mappings without paying attention to their physical implementation. In this section we will make some remarks on the physical implementation of the index structures and the storage space needed. We will not cover the recall index on its own, since it can implemented as an additional functionality of the path selection index. In this section we propose an implementation on top of a relational database system. Other implementations are possible as well, for example using a dedicated text indexing system like Altavista's Search Intranet Developer's Kit (Altavista SDK: [Alt99]).

The path selection index maps terms and labels to sets of inverted document paths. We propose a two-level storage, giving each path an identifier on its own. The path selection index maps terms or labels now to ordered sets of path identifiers and the actual paths have to be retrieved from a path table using the path identifiers. This technique saves storage space, since there may be many occurrences of distinct terms in the same leaf (with the same database path). If we would store the paths explicitly for every term the path index would grow directly proportional with the number of terms in the document collection. In addition we store in every database path for every node (apart from text nodes) not only the node identifiers but also the according labels, e.g. $\langle \texttt{g1} : \texttt{movie}, \texttt{g4} : \texttt{cast}, \texttt{g7} : \texttt{staff}, \texttt{g14} : \texttt{actor}, \texttt{g15} \rangle : 2$. This makes query evaluation more efficient since we main effort in the algorithm is comparing the labels in order to match the query paths, but does not consume much additional storage space, since the labels can be encoded as integer numbers.

The size of the path table is of order $\mathcal{O}(h_{\mathcal{D}} \cdot n)$ since there are maximally n path in the relational document structure and each path has maximally $h_{\mathcal{D}}$ nodes, where $h_{\mathcal{D}}$ is the maximal height of the trees in the relational document structure. If we assume for path and node identifiers a length of 4 bytes, and for label encodings a length of 2 bytes, we have a total storage space for the path table of $n \cdot (4 + h_{\mathcal{D}} \cdot (4+2))$ bytes, i.e. approximately $6 \cdot n \cdot h_{\mathcal{D}}$.

The size of the path selection index itself depends on the number t of terms and labels indexed in the relational document structure. We can assume that t is bigger than n since every leaf contains many terms. Index structures for terms ("inverted files") have been thoroughly investigated in the field of Information Retrieval. There are many techniques how to organize inverted files, like Tries, Patricia Trees, or PAT arrays. The size of an index structure is between 30% and 40% of the original text size ([BYRN99b], pages 192 ff.) but can be reduced with sophisticated compression techniques. Since all structural information is stored in the path table, we can assume the same size $(0.3 \cdot t)$ for the path selection index in our case.

The alignment index depends strongly on the relations that are used. As shown for the example of the left-to-right order a skillful integration of a relation in the framework can save a huge amount of storage space and computational effort. If no relations are defined in addition to the left-to-right order the size of the alignment index is 0, since we already integrated the label information into the path selection index.

The size of the node database depends on the unary relations defined, e.g. attributes etc. The textual content of text nodes is stored with offsets pointing into flat text files storing the documents in their original form. If no additional relations are defined, the size of the node database depends on the width w of the relational document structure, i.e. the maximal number of children a node can have, and is of order $\mathcal{O}(n * w)$. If we assume a storage space of 4 bytes for every node identifier, we have a maximal total storage space for the node database of $(4 \cdot w) + 4 + 4) \cdot n \approx 4 \cdot w \cdot n$ for associating the node identifier of a node with the node identifiers of its parent and children.

The figures given in this section for the size of the index structures do always

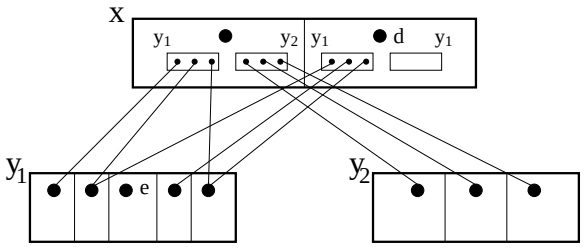
refer to the worst case. In average we can expect much smaller index structures, since, for example, the number of paths is smaller than the number of nodes in the relational document structure, not all paths have the same maximal length $h_{\mathcal{D}}$, and not all nodes have the same number w of children.

5.3 Isolated fields and pointer columns

Given a partially ordered tree query Q, our algorithm first tries to compute a complete answer aggregate for the modified query Q_s that is obtained by suppressing all left-to-right ordering constraints of Q. In this situation the algorithm will sometimes introduce "isolated" fields that do not contribute to answers.

Definition 5.8 A field $Agg_x[d]$ of an aggregate for Q is *upwards isolated* if x is not the root of Q and if there does not exist any vertical pointer with address field $Agg_x[d]$. A field $Agg_x[d]$ is *downwards isolated* if for some child y of x the array $Agg_x[d, y]$ is empty. A field is *isolated* if it is upwards isolated or downwards isolated.

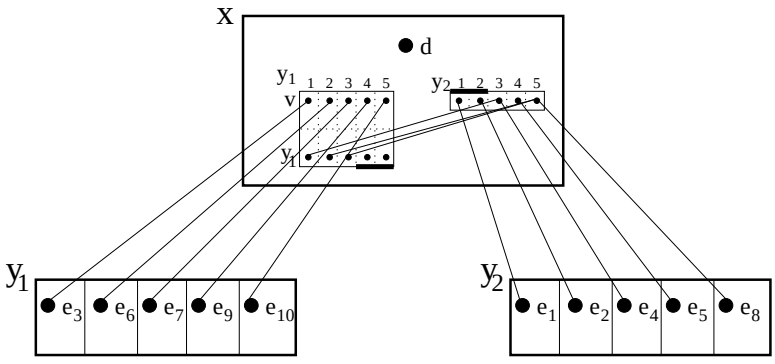
As an illustration consider the following aggregate.



Field $Agg_{y_1}[e]$ is upwards isolated. There is no value for x that would allow for an instantiation of y_1 with e. Field $Agg_x[d]$ is downwards isolated. An instantiation of x with d cannot be completed since there exists no possible instantiation for y_2 in this case. A similar problem may arise with pointer columns in connection with horizontal pointers:

Definition 5.9 A pointer column $Agg_x[d, y_i[l, *]]$ is *right isolated* if the address node of its vertical pointer $Agg_x[d, y_i[l, v]]$ is a node e_i such that for an ordering constraint $y_j <_{lr} y_i$ there is no address node e_j of y_j in $Agg_x[d, y_j]$ such that $e_i <_{lr} e_j$. A pointer column $Agg_x[d, y_j[l, *]]$ with address field $Agg_{y_j}[e_j]$ is *left isolated* if there exists a constraint $y_i <_{lr} y_j$ in Q and if the left-most horizontal pointer $Agg_x[d, y_i[1, y_j]]$ points to a column $Agg_x[d, y_j[k, *]]$ such that $k > l$.

In this situation, an instantiation of y_i (y_j) with e_i (resp. e_j) cannot contribute to a successful instantiation of Agg. As an example, consider the following record, where Q is assumed to have a constraint $y_1 <_{lr} y_2$.



Assume that the indices of e_i reflect the left-to-right ordering, i.e. $e_i <_{lr} e_j$ iff $i < j$. In this case an instantiation of y_1 with e_9 or e_{10} cannot be completed with a suitable instantiation of y_2 such that the constraint $y_1 <_{lr} y_2$ is satisfied. The columns $Agg_x[d, y_1[4, *]]$ and $Agg_x[d, y_1[5, *]]$ are right-isolated. We say that $Agg_x[d, y_1[4, y_2]]$ and $Agg_x[d, y_1[5, y_2]]$ are "dangling" pointers with value $\perp$. On the other hand, it is clear that an instantiation of y_2 with e_1 or e_2 cannot be completed with a corresponding instantiation of y_1. The pointer columns $Agg_x[d, y_2[1, *]]$ and $Agg_x[d, y_2[2, *]]$ are left-isolated.

In the description of the algorithm we shall use dangling pointers with value $\perp$. The adaption of the definition of an aggregate for Q is straightforward.

5.4 Algorithm for Evaluating Tree Queries

In this section we will describe an algorithm for evaluating partially ordered tree queries on relational document structures. In order to make the description of the algorithm more clear, we will first describe a simplified algorithm, that evaluates simple sequence queries on relational document structures. Then we will extend the basic procedures developed for the sequence case to the general case for partially ordered tree queries on ordered relational document structures. As a last step, we will extend this algorithm to the full version that makes more sophisticated use of the index structures and special cases in the query structure.

In all versions the basic procedure will remain the same: In the first phase we will construct an aggregate that possibly contains isolated fields. These fields are removed in a second phase, yielding the complete answer aggregate for the query and relational document structure. The overall structure of the procedure *CLEAN* used in the second phase is very similar to a part of an algorithm described in [MH86]. We will discuss this point in more detail in Section 5.8.2.

The line numbering used in the different versions of the algorithm is consistent, i.e. lines that remain the same have equal line numbers in the different versions. Therefore the line numbering in the first two versions is not always consecutively.

5.4.1 Simple Sequence Queries on Relational Document Structures

The following procedure *COMP-AGG*1 computes a complete answer aggregate for a given sequence query and a relational document structure. As stated in the lines following Lemma 5.4, we assume that the path selection index returns for an inverted query path $\langle x_k, \ldots, x_0 \rangle$ directly the intersection of all path selection index results for all unary index formulae for the leaf variable x_k. The procedure's core is a loop (lines 20-23) where we try to map the query leaf x_k to leaves d_m of the paths returned by the path selection index. For each inverted database path $\langle d_m, \ldots, d_0 \rangle$ returned by the path selection index, the main procedure *SEL-ANC*1 is called in the loop. This procedure yields a Boolean value indicating if d_m is a possible image for x_k. It also computes an aggregate *Agg*. This aggregate may contain some superfluous document nodes that do not contribute to answers. These nodes are marked red. As a final step all red nodes are removed using the procedure *CLEAN*1. The resulting structure *Agg* is the complete answer aggregate for the query Q and database *db*.

The procedure *COMP-AGG*1 contains a call to the path selection index in line 18.

```
 1    procedure Aggt COMP-AGG1(Query q, Database db) {
 2      Aggt Agg := ∅;
16      let π_Q := ⟨x_k, ..., x_0⟩ be the inverted path of Q
```

```
17      Agg := Agg ∪ {Agg_{x_k}} where Agg_{x_k} is an empty record;
18      Π := list of inv. doc. paths obtained from path sel. index for db and π_Q;
19
20      for each path π = ⟨d_m, d_{m-1}, ..., d_0⟩ of Π do {
21         introduce a field Agg_{x_k}[d_m] with empty pointer array for child y of x;
22         SEL-ANC1(Agg, x_k, ⟨d_m, d_{m-1} ..., d_0⟩);
23      }
31      CLEAN1(Agg);
32      return Agg;
      }
```

Phase 1: Construction

*COMP-AGG*1 triggers for each database path a recursive chain of calls to the procedures *SEL-ANC*1 and *CREATE*1 which try to align the query path and a database path, beginning with the leaves and climbing up via parents and ancestors.

In order to avoid the case that pairs $\langle x, d\rangle$ are treated more than once by *SEL-ANC*1, every pair is added to the aggregate (by inserting field $Agg_x[d]$ into record Agg_x) the first time it is treated. If it later turns out that node d is not a possible image of x, the field $Agg_x[d]$ will be colored red.

*SEL-ANC*1$(Agg, y, \langle d_i, \ldots, d_0)$ tries to map a query node y to a database node d_i and yields a respective Boolean value. At first, lines 103-105 verify whether the alignment process is already finished, since it reached the root of the query (successful) or the root of the database path (unsuccessful). If there are nodes left in the query and database path, two cases are distinguished for the parent x of y. (1) If the edge between y and x is rigid (lines 108-112), the parent d_{i-1} of d_i is tried whether it is a possible image for x. This is the case if the pair $\langle d_i, d_{i-1}\rangle$ satisfies the index formulae for $\langle y, x\rangle$ (line 109) and if the call to *CREATE*1 was successful, i.e. the rest of the query and database path could be aligned successfully. (2) If the edge between y and x is a soft edge (lines 113-119), all ancestors $d_{i-1}, \ldots, d_0$ of d_i undergo the same process as in case (1).

If during the execution of *SEL-ANC*1 at least one call to *CREATE*1 was successful, the node d_i is a possible image for y. In the other case the field $Agg_y[d]$ is colored red for later removal (line 121).

*CREATE*1$(x, d_j, Agg, y, \langle d_i, \ldots, d_0\rangle)$ creates a field $Agg_x[d_j]$ (lines 133 or 161) and links it to field $Agg_y[d_i]$ if the call to *SEL-ANC*1 was successful. If necessary a record Agg_x is created beforehand (lines 131-140).

An important feature that leads to the good time-complexity of the algorithm is that *CREATE*1 only calls *SEL-ANC*1 if the field $Agg_x[d_j]$ did not already exist (line 151). The intuitive justification for this is that all the structure (pointers, fields and colors) that we would obtain by continuation of the alignment process has already been included in the aggregate. If the call to *SEL-ANC*1 was successful or if the field $Agg_x[d_j]$ did already exist and wasn't colored red, then vertical pointers from $Agg_y[d_i]$ to $Agg_x[d_j]$ are introduced (lines 152-156 or 162-166).

Summing up, the recursive upwards alignment of the query and database path stops in the following situations: If the root of either path is reached in *SEL-ANC*1, if the index formulae for the two actual query nodes x and y are not satisfied for the actual database nodes, or if *CREATE*1 treats a database node that has already been inserted in the respective record of the aggregate.

The procedure *SEL-ANC*1 contains calls to the alignment index in lines 109 and 115.

```
101    procedure Bool SEL-ANC1(Aggt Agg, QNode y, DBNodes ⟨d_i, ..., d_0⟩) {
102      Bool Node-Found := false;
103      if y = root of Q or i = 0 then {
104        if y is root of Q then Node-Found := true;
105      }
```

```
106       else {
107         x := parent of y in Q;
108         if y is a rigid child of x in Q then {
109           if ⟨d_i, d_{i-1}⟩ satisfy index formulae for ⟨y, x⟩ in Q then {
110             if CREATE1(x, d_{i-1}, Agg, y, ⟨d_i, ..., d_0⟩) then Node-Found := true;
111           }
112         }
113         else {
114           for each node d_j in {d_{i-1}, ..., d_0} do {
115             if ⟨d_i, d_j⟩ satisfy index formulae for ⟨y, x⟩ in Q then {
116               if CREATE1(x, d_j, Agg, y, d_i, ⟨d_i, ..., d_0⟩) then Node-Found := true;
117             }
118           }
119         }
120       }
121       if not Node-Found then COLOR-RED(Agg, y, d_i);
125       return Node-Found;
126     }
127
128
129     procedure Bool CREATE1(QNode x, DBNode d_j,Aggt Agg,
130                            QNode y, DBNodes ⟨d_i, ..., d_0⟩) {
131       if record Agg_x does not exist in Agg then {
132         introduce empty record Agg_x;
133         introduce field Agg_x[d_j] with empty pointer array for the child y of x;
134         if SEL-ANC1(Agg, x, ⟨d_j, ..., d_0⟩) {
135           add new pointer column Agg_x[d_j, y[1, *]] to Agg_x[d_j, y];
136           introduce vertical pointer from Agg_x[d_j, y[1, v]] to Agg_y[d_i];
137           return true;
138         }
139         else return false;
140       }
141       else {
151         if field Agg_x[d_j] exists then {
152           if Agg_x[d_j] is not red then {
153             add new pointer column Agg_x[d_j, y[k, *]] to Agg_x[d_j, y];
154             introduce vertical pointer from Agg_x[d_j, y[k, v]] to Agg_y[d_i];
155             return true;
156           }
157           else
158             return false;
159         }
160         else {
161           introduce a field Agg_x[d_j] with empty pointer array for the child y of x;
162           if SEL-ANC1(Agg, x, ⟨d_j, ..., d_0⟩) {
163             add new pointer column Agg_x[d_j, y[1, *]] to Agg_x[d_j, y];
164             introduce vertical pointer from Agg_x[d_j, y[1, v]] to Agg_y[d_i];
165             return true;
166           }
167           else return false;
168         }
169       }
170     }
```

Phase 2: Cleaning

In the case of sequence queries, the cleaning procedure is remarkably simple: All red fields are removed from the aggregate. We will see later that the situation will become more complicated for tree queries.

```
    procedure Void CLEAN1(Aggt Agg) {
      for all fields Agg_x[d] in Agg do {
        if Agg_x[d] is red then
          remove Agg_x[d] with all its pointer columns;
      }
    }
```

5.4.2 Partially Ordered Tree Queries on Relational Document Structures

We will now treat the full scope of queries covered in this chapter: partially ordered tree queries on ordered document structures. We will use the basic computation scheme of sequence queries based on alignment of paths. Compared with the case of simple sequence queries we can observe three new phenomena: (1) The occurrence of order constraints, (2) the possibility that a field that was entered successfully during the alignment of one query path must be removed since it cannot contribute to the alignment of another query path, and finally (3) the fact that removal of fields during cleaning may trigger removal of other, depending, fields. The main procedure *COMP-AGG*2 now decomposes the query into paths that are handed to *SEL-ANC*2. This procedure takes another argument, *Isol-F*, a stack collecting all fields being colored red during the construction process. In order to treat observation (3), *Isol-F* will guide the cleaning process together with a collection *Isol-PC* containing all pointer columns that must be removed. The construction process of *SEL-ANC*2 does not consider order constraints. The representational and computational information about them is introduced with the procedure *INTR-HOR-P*, treating observation (1). In order to cope with observation (2), records are marked as old or new (lines 17 and 24). Records that have been introduced newly during the treatment of the actual query path are marked as new, all others as old. This information will be used in procedure *CREATE*2.

```
 1     procedure Aggt COMP-AGG2(Query Q, Database db) {
 2       Aggt Agg := ∅;
 3       IsolFields Isol-F := ∅;
16       for all inverted paths π_Q := ⟨x_k, ..., x_0⟩ of Q do {
17         Agg := Agg ∪ {Agg_{x_k}} where Agg_{x_k} is an empty record marked new;
18         Π := list of inv. doc. paths obtained from path sel. index for π_Q and db;
19
20         for each path π = ⟨d_m, d_{m-1}, ..., d_0⟩ of Π do {
21           introduce a field Agg_{x_k}[d_m] with empty pointer array for child y of x;
22           SEL-ANC2(Agg, x_k, ⟨d_m, d_{m-1} ..., d_0⟩, Isol-F);
23         }
24         mark all new records as old;
25       }
29       IsolCols Isol-PC := ∅;
30       INTR-HOR-P(Q, Agg, Isol-F, Isol-PC)
31       CLEAN2(Agg, Isol-PC, Isol-F);
32       return Agg;
33     }
```

Phase 1: Construction

The structure and recursive dependency scheme of the two procedures *SEL-ANC*2 and *CREATE*2 is roughly the same as for *SEL-ANC*1 and *CREATE*1. *SEL-ANC*2 differs from *SEL-ANC*1 only in the fact that fields that are colored red are in addition added to the stack *Isol-F* of isolated fields (line 122). The structure of *CREATE*2 is enriched by another case, distinguishing between new and old records. New records are treated in the same way as in *CREATE*1. In old records no new fields can be inserted and it is therefore checked whether a field already exists (lines 142-149). If this is not the case, node d_j is not entered and *CREATE*2 returns `false`. The intuition behind this is that query node x has already been treated as part of another query path (because the record Agg_x is marked old). In this treatment, d_j did not qualify as an image of x. Therefore d_j cannot qualify as image of x globally, even if it does qualify now in the local treatment of the actual query path. $Agg_x[d_j]$ would necessarily be a downwards isolated field, that had to be detected and removed in the rest of the algorithm. The rest of *CREATE*2 is the

same as *CREATE*1.

The new pointer columns that are introduced in lines 144 and 153 have to be inserted at a position so that the order of pointer columns (reflecting the order of the target candidates being pointed at by the vertical pointers) is respected: In a pointer array a pointer column precedes another pointer column iff the vertical pointer of the former points to a field preceding the field pointed at by the vertical pointer of the latter.

The new procedure *INTR-HOR-P* serves two goals: (1) Detect downwards isolated fields in *Agg* and add them to *Isol-F* (lines 176-180), and (2) compute and insert administrative information for managing the order constraints and detecting pointer columns not respecting the order constraints (lines 181-201). In order to achieve these goals, *INTR-HOR-P* visits every array $Agg_x[d, y_i]$. If this array is empty, the field is downwards isolated and therefore colored red and added to *Isol-F* (lines 177-179) for later removal. In the other case, all relevant left-to-right ordering constraints $y_i <_{lr} y_j$ are considered. At this point the order of pointer columns reflecting the order of their vertical pointers becomes important. For each pointer column the procedure tries to introduce the appropriate horizontal pointer (lines 185-193). If this is not possible (i.e., for right isolated pointer columns) a "dangling" pointer $\perp$ is introduced and the pointer column is added to the stack *Isol-PC* of isolated pointer columns and colored yellow for later removal (lines 194-198). When treating the first pointer column $Agg_x[d, y_i[1, *]]$ we also check if $Agg_x[d, y_j]$ contains left isolated pointer columns (lines 187-191). These are added to the stack of isolated pointer columns.

```
101   procedure Bool SEL-ANC2(Aggt Agg,QNode y,DBNodes ⟨d_i,...,d_0⟩,IsolFields Isol-F) {
102     Bool Node-Found := false;
103     if y = root of Q or i = 0 then {
104       if y is root of Q then Node-Found := true;
105     }
106     else {
107       x := parent of y in Q;
108       if y is a rigid child of x in Q then {
109         if ⟨d_i, d_{i-1}⟩ satisfy index formulae for ⟨y, x⟩ in Q then {
110           if CREATE2(x, d_{i-1}, Agg, y, ⟨d_i,...,d_0⟩) then Node-Found := true;
111         }
112       }
113       else {
114         for each node d_j in {d_{i-1},...,d_0} do {
115           if ⟨d_i, d_j⟩ satisfy index formulae for ⟨y, x⟩ in Q then {
116             if CREATE2(x, d_j, Agg, y, d_i, ⟨d_i,...,d_0⟩) then Node-Found := true;
117           }
118         }
119       }
120     }
121     if not Node-Found then {
122       add Agg_y[d_i] to Isol-F;
123       COLOR-RED(Agg,y,d_i);
124     }
125     return Node-Found;
126   }
127
128
129   procedure Bool CREATE2(QNode x, DBNode d_j, Aggt Agg,
130                          QNode y, DBNodes ⟨d_i,...,d_0⟩, IsolFields Isol-F) {
131     if record Agg_x does not exist in Agg then {
132       introduce empty record Agg_x marked new;
133       introduce field Agg_x[d_j] with empty pointer arrays for the children of x;
134       if SEL-ANC2(Agg, x, ⟨d_j,...,d_0⟩) {
135         add a pointer column Agg_x[d_j, y[1,*]] to Agg_x[d_j, y];
136         introduce vertical pointer from Agg_x[d_j, y[1, v]] to Agg_y[d_i];
137         return true;
138       }
```

```
139         else return false;
140       }
141       else {
142         if Agg_x is marked old then {
143           if field Agg_x[d_j] exists and is not marked red then {
144             add a new pointer column Agg_x[d_j, y[k,*]] to Agg_x[d_j];
145             introduce a vertical pointer from Agg_x[d_j, y[k,v]] to Agg_y[d_i];
146             return true;
147           }
148           else return false;
149         }
150         else {
151           if field Agg_x[d_j] exists then {
152             if Agg_x[d_j] is not marked red then {
153               add a pointer column Agg_x[d_j, y[k,*]] to Agg_x[d_j, y];
154               introduce vertical pointer from Agg_x[d_j, y[k,v]] to Agg_y[d_i];
155               return true;
156             }
157             else
158               return false;
159           }
160           else {
161             introduce field Agg_x[d_j] with empty pointer arrays for children of x;
162             if SEL-ANC2(Agg, x, ⟨d_j, ..., d_0⟩) {
163               add a pointer column Agg_x[d_j, y[1,*]] to Agg_x[d_j, y];
164               introduce vertical pointer from Agg_x[d_j, y[1,v]] to Agg_y[d_i];
165               return true;
166             }
167             else return false;
168           }
169         }
170       }
171     }
172
173
174     procedure Void INTR-HOR-P(Query Q, Aggt Agg,
175                               IsolFields Isol-F, IsolCols Isol-PC) {
176       for each array Agg_x[d, y_i] of Agg do {
177         if Agg_x[d, y_i] = empty array then {
178           add Agg_x[d] to Isol-F;
179           COLOR-RED(Agg, x, d);
180         }
181         else {
182           for each constraint y_i <_lr y_j of Q do {
183             for each column Agg_x[d, y_i[l,*]] of Agg_x[d, y_i] do {
184               e_i := address node of Agg_x[d, y_i[l,v]];
185               if exists k := minimal number s.th. address node e_j
186                         of Agg_x[d, y_j[k,v]] satisfies e_i <_lr e_j then {
187                 if l = 1 and k > 1 then
188                   for all k' < k do {
189                     add Agg_x[d, y_j[k',v]] to Isol-PC;
190                     COLOR-YELLOW(Agg, x, d, y_j, k');
191                   }
192                 introduce hor. pointer Agg_x[d, y_i[l, y_j]] pointing to Agg_x[d, y_j[k,*]];
193               }
194               else {
195                 set pointer Agg_x[d, y_i[l, y_j] to ⊥;
196                 add Agg_x[d, y_i[l,*]] to Isol-PC;
197                 COLOR-YELLOW(Agg, x, d, y_i, l);
198               }
199             }
200           }
201         }
202       }
203     }
```

Phase 2: Cleaning

In phase 2 the pointer columns in *Isol-PC* and the fields in *Isol-F* are removed from the aggregate. All pointer columns in *Isol-PC* are marked yellow in the beginning.

Basically the following procedure is very simple. We take the eliminable elements from the stacks and erase them by calls to the procedures *ELIM-F*/ and *ELIM-PC*. Since the erasure of an isolated element may lead to new isolated elements the process has to be organized in a recursive way. If during this process a record Agg_x becomes empty, then the procedure stops (*Solution-Possible* = `false`). Otherwise it continues until all isolated fields and pointer columns are erased.

In the presence of left-to-right ordering constraints, the strategy has to be modified. We do not immediately erase an isolated pointer column that possibly serves as the address of horizontal pointers. The reason is that we have to re-address these horizontal pointers, using the column that represents the right neighbour as the new address. With the naive strategy this process possibly would have to be iterated, we would end up with a quadratic complexity.

Hence, when treating yellow pointer columns that possibly serve as the target of horizontal pointers we proceed in two steps. Instead of erasing the column, we only eliminate the (vertical and horizontal) pointers departing from the column and colour the column "red" afterwards. Horizontal pointers to red columns are only re-addressed once, after all yellow elements have been treated (lines 316-321).

The procedure *ELIM-PC* eliminates/colors isolated pointer columns. We say that the address field $Agg_y[e]$ of a vertical pointer $Agg_x[d, y[l, v]]$ is *upwards isolated up to* $Agg_x[d, y[l, v]]$ iff $Agg_x[d, y[l, v]]$ is the only vertical pointer with address field $Agg_y[e]$. During a call to *ELIM-PC* the vertical pointer of the column is erased (line 335), adding the target field to *Isol-F* if the field becomes upwards isolated (lines 329-333). When treating the last non-red column of an array (337-340) we know that after the final removal of red columns the actual field will be downwards isolated. Hence the field is added to *Isol-F*. In the situation of lines 346-349 the columns $Agg_x[d, y_i[k', *]]$ are necessarily left-isolated when removing red columns. In the situation of lines 358-361 the columns $Agg_x[d, y_i[k', *]]$ are necessarily right-isolated when removing red columns.

The procedure *ELIM-F* simply removes isolated fields by first removing all vertical pointers pointing at them (lines 377-381) and then removing all pointers of the field itself (line 390). This latter removal can cause other fields to become upwards isolated (lines 386-389).

```
301     procedure Bool CLEAN2(Aggt Agg, IsolCols Isol-PC, IsolFields Isol-F) {
302       Bool Solution-Possible := true;
303       repeat until Solution-Possible = false or Isol-PC = Isol-F = ∅ {
304         if Isol-F ≠ ∅ then {
305           let Agg_x[d] ∈ Isol-F;
306           if not ELIM-F(Agg, x, d, Isol-PC, Isol-F) then
307              Solution-Possible=false;
308         }
309         else {
310           let Agg_x[d, y[l, *]] ∈ Isol-F;
311           ELIM-PC(Agg, x, d, y, l, Isol-PC, Isol-F);
312         }
313       }
314       if not Solution-Possible return false;
315
316       for each red pointer column Agg_x[d, y[l, *]] of Agg do {
317         select minimal l' > l where column Agg_x[d, y[l', *]] is not red;
318         re-address horizontal pointers with target Agg_x[d, y[l, *]]
319                                        to new address Agg_x[d, y[l', *]];
320         erase Agg_x[d, y[l, *]];
321       }
322       return true;
```

```
323     }
324
325
326     procedure Void ELIM-PC(Aggt Agg, QNode x, DBNode d, QNode y, Nat l,
327                            IsolCols Isol-PC, IsolFields Isol-F) {
328       if Agg_x[d, y[l, v]] is not dangling then {
329         Agg_y[e] := address field of Agg_x[d, y[l, v]];
330         if Agg_y[e] is upwards isolated up to Agg_x[d, y[l, v]] then {
331           add Agg_y[e] to Isol-F;
332           COLOR-RED(Agg,y,e);
333         }
334       }
335       erase Agg_x[d, y[l, v]];
336
337       if Agg_x[d, y[l, *]] is the only non-red column of Agg_x[d, y] then {
338         add Agg_x[d] to Isol-F;
339         COLOR-RED(Agg,x,d);
340       }
341       else {
342         if Agg_x[d, y[l, *]] is the leftmost non-red column of Agg_x[d, y] then {
343           let Agg_x[d, y[l', *]] be the next non-red column of Agg_x[d, y];
344           for each hor. pointer Agg_x[d, y[l', y']] of Agg_x[d, y[l', *]] do {
345             let Agg_x[d, y_i[k, *]] := target of Agg_x[d, y[l', y_i]];
346             for each k' < k where column Agg_x[d, y_i[k', *]] neither yellow nor red do {
347               add Agg_x[d, y_i[k', *]] to Isol-PC;
348               COLOR-YELLOW(Agg,x,d,y_i,k');
349             }
350           }
351         }
352         else {
353           if  Agg_x[d, y[l, *]] is the rightmost non-red column of Agg_x[d, y] then {
354             let Agg_x[d, y[l_0, *]] be the preceding non-red column of Agg_x[d, y];
355             let l_1, ..., l_k = l denote the indexes of successor columns
356                                 of Agg_x[d, y[l_0, *]] ending at Agg_x[d, y[l, *]];
357             for each 1 ≤ i ≤ k do
358               for each hor. pointer Agg_x[d, y_j[m, y]] with address Agg_x[d, y[l_i, *]] do {
359                 add Agg_x[d, y_j[m, *]] to Isol-PC;
360                 COLOR-YELLOW(Agg,x,d,y_j,m);
361               }
362           }
363         }
364
365         erase all hor. pointers departing from Agg_x[d, y[l, *]];
366
367         if Agg_x[d, y[l, *]] is not target of any hor. pointer then
368           erase Agg_x[d, y[l, *]]
369         else
370           COLOR-RED Agg_x[d, y[l, *]];
371       }
372     }
373
374
375     procedure Bool ELIM-F(Aggt Agg, QNode x, DBNode d,
376                           IsolCols Isol-PC, IsolFields Isol-F) {
377       for each vertical pointer Agg_z[d', x[k, v]] with address field Agg_x[d] do {
378         redefine address of Agg_z[d', x[k, v]] to ⊥;
379         add Agg_z[d', x[k, *]] to Isol-PC;
380         COLOR-YELLOW(Agg,z,d',x,k);
381       }
382
383       for each child y of x in Q do
384         for each column Agg_x[d, y[l, *]] of Agg_x[d, y] do {
385           Agg_y[e] := address field of Agg_x[d, y[l, v]];
386           if Agg_y[e] is upwards isolated up to Agg_x[d, y[l, v]] then {
387             add Agg_y[e] to Isol-F;
388             COLOR-RED(Agg,y,e);
389           }
```

```
390           erase Agg_x[d, y[l, *]];
391         }
392
393       erase Agg_x[d];
394       if Agg_x = empty record then
395         return false
396       else
397         return true;
398     }
```

5.4.3 Optimizations

We will introduce two optimizations for the basic algorithm in order to detect and treat special cases where the performance can be made more efficient. These special cases are given by the number of paths returned by the path selection index.

Order of Query Path Treatment

We can reduce the number of target candidates that are stored in the answer aggregate during computation if we first treat those query paths that have to be compared with a small number of database paths. Therefore we first chose query paths whose index accesses return a small number of database paths.

We need therefore a further index structure, the *recall index* returning the recall $\mathrm{REC}(\pi_Q)$ of the (combined) accesses to the the path selection index triggered by a query path π_Q. The *recall* of an index access is defined as the number of paths returned.

Top-down Treatment

In many cases the recall of the path selection index accesses is too high to perform efficiently. Take as an example a query specifying a set of articles of interest by their content and structure, and additionally restricting the articles to ones containing a figure. Such a query contains a query path ending at a structural leaf with the label "figure" that contains no additional index information. It is likely that the path selection index will return very many database paths to a "figure" node, i.e. the recall of the path selection index access with label "figure" is very high. We again use the recall index here. If the recall value $\mathrm{REC}(\pi_Q)$ for a given query path π_Q is above a given threshold, the algorithm treats the query path in a top-down fashion in a navigational way using the node database.

The algorithm incorporating the two optimizations has the same basic structure as before. In lines 6-15 of *COMP-AGG3* we define two path sets $\Pi_Q^{>t}$ for the query paths with recall higher than threshold t and $\Pi_Q^{\leq t}$ containing query paths with a recall smaller or equal than t. The paths in $\Pi_Q^{\leq t}$ are sorted according to their recall: the paths with lower recall precede ones with higher recall. If no query path has a recall below the threshold, $\Pi_Q^{\leq t}$ contains the path with the lowest recall and $\Pi_Q^{>t}$ all other query paths (lines 12-15). This is necessary, since we need at least one path with bottom-up treatment for filling the aggregate partially before beginning the top-down treatment. The query paths in $\Pi_Q^{\leq t}$ are treated in the main loop (lines 16-25) in the bottom-up fashion as in *COMP-AGG2*, apart from the fact that query paths with lower recall are treated first. For the paths in $\Pi_Q^{>t}$ a call to *INIT-TD-ALIGN* triggers the top-down treatment (lines 26 and 27) with procedure *TD-ALIGN*.

The structure of *TD-ALIGN* and *INIT-TD-ALIGN* is similar to the structure of *SEL-ANCn* and *CREATEn* with the difference that *TD-ALIGN* proceeds in a top-down fashion and that the alignment process is guided by the node database (with calls to the set of children of a database node d) instead of path sets returned

by the path selection index. *INIT-TD-ALIGN* finds the bottom-most slot, that has already been treated as part of another query path (line 403). It then tests for all fields contained in this slot and their descendants whether they can be aligned with the rest of the query path. These alignments tests are performed by the recursive procedure *TD-ALIGN*: This procedure assumes that database node d has already been entered as a field into record Agg_{x_1} and now tries, whether e can be entered as field into Agg_{x_2}, where x_2 is the child of x_1 in the actual path. If there exists no record Agg_{x_2}, it is created (lines 4219 and 420). If the field $Agg_{x_2}[e]$ already exists and is not colored red, it is linked to $Agg_{x_1}[d]$ and the recursive procedure stops (lines 423-429). In the other case it is only linked if $\langle d, e\rangle$ satisfy the index formulae for $\langle x_1, x_2\rangle$ and if one of the recursive calls to *TD-ALIGN* succeeds (lines 431-447). If the edge between x_1 and x_2 is a soft edge, then the recursive calls are in addition made to the children of e (lines 448-451).

The procedure *COMP-AGG3* contains, besides the call to the path selection index in line 18 and 19, a call to the recall index that is implicit in the program code by referring to the function REC (lines 5-11). The procedures *INIT-TD-ALIGN* and *TD-ALIGN* contain a call to the node database in lines 406, 436 and 449. In addition, *TD-ALIGN* contains a call to the alignment index in line 431.

```
1    procedure Aggt COMP-AGG3(Query Q, Database db, int t) {
2      Aggt Agg := ∅;
3      IsolFields Isol-F := ∅;
4
5      let π_Q^0 be the inv. query path with lowest recall;
6      if REC(π_Q^0) < t then {
7        let Π_Q^{>t} be the list of inv. query paths with recall
8                          higher than threshold t;
9        let Π_Q^{≤t} be the list of inv. query paths with recall
10                         lower or equal than threshold t with ascending recall;
11     }
12     else {
13       let Π_Q^{≤t} := {π_Q^0};
14       let Π_Q^{>t} be the set of all query paths excluding π_Q^0;
15     }
16     for all (sorted) inv. query paths π_Q := ⟨x_k, ..., x_0⟩ in Π_Q^{≤t} do {
17       Agg := Agg ∪ {Agg_{x_k}} where Agg_{x_k} is an empty record marked new;
18       Π := list of inv. doc. paths obtained from path selection index
19                                                    for π_Q and db;
20       for each path π = ⟨d_m, d_{m-1}, ..., d_0⟩ of Π do {
21          introduce a field Agg_{x_k}[d_m];
22          SEL-ANC2(Agg, x_k, ⟨d_m, d_{m-1} ..., d_0⟩, Isol-F);
23       }
24       mark all new records as old;
25     }
26     for all inverted query paths π_Q in Π_Q^{>t} do
27       INIT-TD-ALIGN(Agg,π_Q,Isol-F);
28
29     IsolCols Isol-PC := ∅;
30     INTR-HOR-P(Q, Agg, Isol-F, Isol-PC)
31     CLEAN2(Agg, Isol-PC, Isol-F);
32     return Agg;
33   }


401  procedure Void INIT-TD-ALIGN (Aggt Agg, QueryPath π_Q, IsolFields Isol-F) {
402    let π_Q = ⟨x_m, ..., x_0⟩;
403    let k be maximal so that x_k is marked old;
404    for all non-red fields Agg_{x_k}[d] in Agg_{x_k} do {
405      Bool Node-Found := false;
406      for all children e of d do
407        if TD-ALIGN(Agg, ⟨x_k, ..., x_m⟩,d,e,Isol-F) then
408          Node-Found := true;
```

```
409         if not Node-Found then {
410           add Agg_{x_k}[d] to Isol-F;
411           COLOR-RED(Agg,x_k,d);
412         }
413       }
414     }
415
416
417     procedure Bool TD-ALIGN(Aggt Agg, QNodes ⟨x_1,...,x_m⟩,
418                                       DBNode d, DBNode e, IsolFields Isol-F) {
419       if record Agg_{x_2} does not exist then
420         introduce empty record Agg_{x_2};
421
422       Bool Node-Found := false;
423       if field Agg_{x_2}[e] exists {
424         if Agg_{x_2}[e] is not red {
425           add a pointer column Agg_{x_1}[d, x_2[l,*]] to Agg_x[d_j, y];
426           introduce vertical pointer from Agg_{x_1}[d, x_2[l,v]] to Agg_{x_2}[e];
427            Node-Found := true;
428         }
429       }
430       else {
431         if ⟨d,e⟩ satisfy index formulae for ⟨x_1,x_2⟩ then {
432           introduce field Agg_{x_2}[e] with empty pointer array for the children y of x_2;
433           if m = 2 then
434              Node-Found := true;
435           else
436             for all children f of e do
437               if TD-ALIGN(Agg,⟨x_2,...,x_m⟩,e,f,Isol-F)) then
438                  Node-Found := true;
439           if Node-Found then {
440             add a pointer column Agg_{x_1}[d, x_2[l,*]] to Agg_x[d_j, y];
441             introduce vertical pointer from Agg_{x_1}[d, x_2[l,v]] to Agg_{x_2}[e];
442           }
443           else {
444             add Agg_{x_2}[e] to Isol-F;
445             COLOR-RED(Agg,x_2,e);
446           }
447         }
448         if x_2 is a soft child of x_1 then
449           for all children e' of e do
450             if TD-ALIGN(Agg,⟨x_1,...,x_m⟩,d,e',Isol-F) then
451                Node-Found := true;
452       }
453       return Node-Found;
454     }
```

5.5 Correctness

In this section we prove the existence parts of Theorems 4.11 and 4.40. Clearly, simple tree queries are a special case of partially ordered tree queries and it suffices to prove Theorem 4.40. Since the algorithm computes complete answer aggregates as opposed to complete answer formulae we first give an internal characterization of complete answer aggregates. We will show that the algorithm computes an aggregate that satisfies the conditions of this characterization, which proves Theorem 4.40.

Lemma 5.10 *Let Q be a partially ordered tree query and $\mathcal{D}$ an ordered document structure. An aggregate Agg is the complete answer aggregate Agg_Q for Q and $\mathcal{D}$ iff the following conditions are satisfied:*

(AGG1) *Each instantiation ν of Agg is an answer to Q and $\mathcal{D}$ and vice versa,*

(AGG2) *every field/pointer column of Agg belongs to an instantiation of Agg.*

The simple proof is omitted. We only note that (AGG1) exactly corresponds to the "contribution obligation" condition for dependent Q-instantiation formulae (cf. paragraph below Definition 4.31). Hence it remains to prove that the aggregate *Agg* that represents the output of the algorithm satisfies (AGG1) and (AGG2). Let us start with some simple observations.

Lemma 5.11 *Let Agg_1 and Isol-F_1 denote the value of variables Agg and Isol-F in line 29 of COMP-AGG2. Then all upwards isolated fields of Agg_1 are in Isol-F_1.*

Proof. This follows from the fact that whenever we cannot finish the bottom-up alignment of inverted query path and inverted document paths we add the field of the last successful alignment step to *Isol-*F_1 (cf. *SEL-ANC* and *CREATE*). The same is true for the top-down-alignment (cf. *TD-ALIGN*). □

Lemma 5.12 *Let Agg_2, Isol-F_2 and Isol-PC_2 denote the value of variables Agg, Isol-F and Isol-PCafter treating line 30. Then all isolated fields and pointer columns of Agg_2 are in Isol-F_2 and Isol-PC_2 respectively.*

Proof. Clearly Lemma 5.11 implies that upwards isolated fields are in *Isol-*F_2. *INTR-HOR-P* treats each array $Agg_x[d, y]$ of the aggregate and adds downwards isolated fields to *Isol-F*. It also adds each right or left isolated pointer column to *Isol-PC*. □

During phase 2, let us call a field/pointer column *quasi-isolated* if the element becomes isolated when removing red pointer columns.

Lemma 5.13 *At each time of the computation in phase 2, each quasi-isolated field is either on the actual stack Isol-F or it represents the actual argument of the elimination sub-procedure that is executed. Each quasi-isolated pointer column is either on the actual stack Isol-PC, or it represents the actual argument of the elimination sub-procedure that is executed, or it is coloured "red".*

Proof. Follows from Lemma 5.12 by a trivial induction and inspection of *CLEAN*2, noticing that a new quasi-isolated field/pointer column can only be the result of the elimination/red colouring of a field/pointer column that had been quasi-isolated previously. □

Lemma 5.14 *Let Agg_3 denote the output of the algorithm. Then Agg_3 does not have isolated fields or pointer columns.*

Proof. Consider the situation in *CLEAN*2 where both *Isol-F* and *Isol-PC* are empty. In this situation by Lemma 5.13, the only quasi-isolated elements that are left are the red pointer columns. These columns are erased in *CLEAN*2. Clearly the elimination of a red column cannot lead to a new quasi-isolated element. □

Lemma 5.15 *If an aggregate Agg for Q does not have any isolated field/pointer column, then every field/pointer column of Agg belongs to an instantiation of Agg.*

Proof. We proceed by induction on $h_Q(x)$ where x is the root of Q. If $h_Q(x) = 0$, then Agg_x is the only record of *Agg* and the statement is trivial. Assume now that $h_Q(x) > 0$, let $y_1, \ldots, y_h$ denote the children of x in Q. Let Agg_i denote the sub-aggregate with topmost record Agg_{y_i} $(1 \leq i \leq h)$. By induction hypothesis each field/pointer column of Agg_i belongs to an instantiation of Agg_i. Now let $Agg_z[d_0]$ be a field of *Agg*. We distinguish two cases.
In the first case, $Agg_z[d_0]$ is a field of a sub-aggregate Agg_i. We may use the induction hypothesis to obtain an instantiation ν_i of Agg_i such that $Agg_z[d]$ belongs

to ν_i. Let $e_i = \nu(y_i)$. Since $Agg_{y_i}[e_i]$ is not upwards isolated there exists a field $Agg_x[d]$ with a vertical pointer $Agg_x[d, y_i[l_i, v]]$ (for suitable l_i) with address field $Agg_{y_i}[e_i]$. Since $Agg_x[d]$ is not downwards isolated and since no pointer column of $Agg_x[d]$ is left or right isolated it follows easily that we may select for all $1 \leq j \neq i \leq h$ vertical pointers $Agg_x[d, y_j[l_j, v]]$ with address fields $Agg_{y_j}[e_j]$ that obey condition (INST3) of Definition 4.18. By induction hypothesis, each of the fields $Agg_{y_j}[e_j]$ belongs to an instantiation ν_j of Agg_j $(1 \leq j \neq i \leq h)$. Combining the mappings ν_j for $1 \leq j \leq h$ and mapping x to d we obtain an instantiation ν of Agg such that $Agg_z[d_0]$ belongs to ν.
In the second case, where $Agg_z[d_0] = Agg_x[d]$ is in Agg_x we may directly use the fact that $Agg_x[d]$ is not downwards isolated and no pointer column of $Agg_x[d]$ is left or right isolated to conclude with the induction hypothesis that there exists an instantiation ν of Agg such that $Agg_z[d_0]$ belongs to ν.
The proof that each pointer column of Agg belongs to a suitable instantiation of Agg is analogous. □

Summing up, we have seen that the output aggregate of the algorithm satisfies (AGG2) of Lemma 5.10. We now show *completeness* of the algorithm.

Lemma 5.16 *Each answer to Q may be obtained as an instantiation of the output aggregate Agg of the algorithm.*

Proof. Let $\nu : fr(Q) \to D$ be an answer to Q. If $\pi_Q = \langle x_k, \ldots, x_0 \rangle$ is an inverted query path, then let $\pi = \langle d_m, d_{m-1}, \ldots, d_0 \rangle$ denote the unique inverted document path with first (bottom-most) element $\nu(x_k)$. By assumption, π is one of the paths obtained by the index access for π_Q (cf. Remark 5.3). The mapping ν determines a match $\nu_{\pi_Q} : \{x_k, \ldots, x_0\} \to \{d_m, d_{m-1}, \ldots, d_0\}$ that can be used by the algorithm for successful alignment of the two paths. This shows that in the state of line 29 each record Agg_{x_i} will have a field $Agg_{x_i}[\nu(x_i)]$ $(0 \leq i \leq k)$, with vertical pointer $Agg_{x_i}[\nu(x_i), x_{i+1}[l, v]]$ to $Agg_{x_{i+1}}[\nu(x_{i+1})]$ for $i < k$ and suitable l. The combination of the mappings ν_{π_Q} for distinct inverted query paths π_Q, i.e, the mapping ν, satisfies conditions (INST1) and (INST2) of Definition 4.18. Since ν is an answer to Q, and by definition of horizontal pointer addresses, it follows also that ν satisfies condition (INST3) of Definition 4.18. Hence it defines an instantiation of the aggregate Agg_1 that is reached after Phase 1.
Clearly none of the fields/pointer columns that belong to ν are isolated in Agg_1. We now show by induction that none of the fields/pointer columns that belong to ν becomes quasi-isolated when applying *ELIM-PC* and *ELIM-F*. This shows that ν is an instantiation of the aggregate Agg_2 obtained as output of the algorithm and finishes the proof. By Lemma 5.13 it suffices to show that none of the fields/pointer columns that belong to ν are added to *Isol-F* and *Isol-PC* respectively. A simple inspection of the procedures *ELIM-PC* and *ELIM-F* shows that a field/pointer column that belongs to ν can only be added to *Isol-F* and *Isol-PC* during a process where we actually eliminate (or colour red) another field/pointer column belonging to ν. This would mean that the latter field/pointer column had been added to *Isol-F* and *Isol-PC* before, which contradicts the induction hypothesis. □

It remains to prove *soundness* of the algorithm.

Lemma 5.17 *Each instantiation of the output aggregate Agg of the algorithm is an answer to Q.*

Proof. Let ν be an instantiation of Agg. Let y be a child of x in Q, and let $e := \nu(y)$ and $d := \nu(x)$. Remark 5.3 and the tests in *SEL-ANC* ensure that (e, d) satisfies all unary and binary index formulae imposed on (y, x) in Q. Condition (INST3) of Definition 4.18 ensures that ν satisfies all ordering constraints of Q. Hence ν is an answer to Q. □

5.6 Time and Space Complexity

Before we state the main complexity result we start with some general remarks.

Remark 5.18 The pre-order relationship $<_p^D$ on $\mathcal{D}$ can be represented by assigning to every node d a natural number $\mathrm{ord}(d)$ as identifier so that

$$\mathrm{ord}(d_1) < \mathrm{ord}(d_2) \text{ iff } d_1 <_p d_2.$$

We assume that the comparison of two natural numbers is of constant-time complexity.[4] Procedure *INTR-HOR-P* also includes tests $d_1 <_{lr}^D d_2$. For these tests we use a supplementary pointer structure (that can be integrated as part of the node database): each node $d \in D$ has a pointer to the node $e = \mathit{minsucc}(d)$ that represents the first $<_p^D$-successor of d that is larger than d with respect to left-to-right ordering, i.e. that is no descendant of d. In other words, we have $d <_{lr}^D e$ and there is no node $e' <_p^D e$ such that $d <_{lr}^D e'$. With this prerequisite, Lemma 3.4 allows to reduce the left-to-right ordering to the pre-order relationship: in fact the lemma shows that for $d_1, d_2 \in D$ we have $d_1 <_{lr}^D d_2$ iff $\mathit{minsucc}(d_1) \leq_p^D d_2$ and the latter formula can be tested in constant time.

Remark 5.19 Let $E \subseteq D$ and assume that the pre-order relationship $<_p^D$ on E is encoded using a balanced binary tree with height of order $\mathcal{O}(\log(|D|))$ where the elements of E are represented by the leaves. Assume that each node is coloured black, yellow, or red. Then the following operations can be computed in time $\mathcal{O}(\log(|D|))$:

- find the first predecessor (successor) (w.r.t. "$<_p^D$") of a given element that has a given colour,
- check if a given element is the only (right-most, left-most) element of a given colour,
- change the colour of a given element.

The operations can be implemented by adding to each inner node of the binary tree a label "black" ("yellow", or "red") iff it has a successor leaf that has the respective colour.

We may now state the main complexity result.

Theorem 5.20 *The worst-case time-complexity of the algorithm described in Section 5.4.2 is of order $\mathcal{O}(|Q| \cdot |D| \cdot h_{\mathcal{D}} \cdot \log|D|)$. If Q is rigid, or if $\mathcal{D}$ is non-recursive and Q is labeling-complete, then the time-complexity is $\mathcal{O}(|Q| \cdot |D| \cdot \log(|D|))$. In either case the worst-case space complexity is $\mathcal{O}(|Q| \cdot |D| \cdot h_{\mathcal{D}})$.*

Proof. Since the maximal size of an aggregate for a query Q and ordered relational document structure is according to Theorem 4.49 of order $\mathcal{O}(|Q| \cdot |D| \cdot h_{\mathcal{D}})$, the space complexity result holds trivially.
Lemma 5.5 shows that we may ignore the access to the path selection index for obtaining the above bound.
First we consider lines 16-29 of the algorithm.
As we noted in the description of the algorithm the total number of calls to procedure *SEL-ANC* is bounded by $|Q| \cdot |D|$. Each non-trivial test in *SEL-ANC* can

[4] In fact, if we do not impose an upper bound on the natural numbers, the time complexity of the comparison is $\mathcal{O}(\log(n))$. But it is a reasonable and canonical assumption in database theory to impose an upper bound on the size of databases, e.g. that the database should not contain more than 2^{32} nodes. Then we can rely on efficient hardware treatment for the comparison of two 32-bit integers.

be performed in constant time (cf. Remark 5.6), the number of tests in one call to *SEL-ANC*, similarly as the number of possible calls to *CREATE*, is bounded by $h_{\mathcal{D}}$. Let us investigate each of the steps of procedure *CREATE*. Obviously, given $x \in fr(Q)$ it is possible (e.g., by adding appropriate information to the query) to check in constant-time if a record Agg_x exists and to determine whether it is marked as new or old. Using binary search it takes time $\mathcal{O}(log(|D|))$ to check if a field $Agg_x[d_j]$ exists for given x and d_j. The same bound holds for the introduction of new pointer columns and fields where pre-ordering has to be respected. It follows that one call to *CREATE* needs time $\mathcal{O}(log(|D|))$. Hence we obtain a bound $\mathcal{O}(|Q| \cdot |D| \cdot h_{\mathcal{D}} \cdot log(|D|))$ for the first step.
We consider now procedure *INTR-HOR-P*.
Remark 4.48 shows that the total number of horizontal pointers is bounded by $\mathcal{O}(|Q| \cdot |D| \cdot h_{\mathcal{D}})$. Using binary search it takes time $\mathcal{O}(log(|D|))$ to determine the correct address for a given pointer. Since each column is added to the stack of isolated columns at most once we receive the bound $\mathcal{O}(|Q| \cdot |D| \cdot h_{\mathcal{D}} \cdot log(|D|))$ for Step 2. Summing up, this bound is also obtained for Phase 1 in total.
If Q is rigid, or if Q is labeling-complete and $\mathcal{D}$ is non-recursive, then each call to *SEL-ANC* leads to just one test and to at most to one call of *CREATE*. We obtain a bound $\mathcal{O}(|Q| \cdot |D| \cdot log(|D|))$ for lines 16-19. By Remark 4.48, the total number of pointer columns/vertical pointers that are introduced in Phase 1 is bounded by $|Q| \cdot |D|$.
A simple analysis of *Phase 2* shows that for each element (field/pointer/pointer column) there is only a fixed number of operations that is possibly applied (put the element on a stack, colour it, compute/redefine address, erase the element, check if element is only non-red column, or left-most non-red column etc.). This is fairly obvious, we just add some remarks. The horizontal pointers that are inspected in line 344 are only inspected once. In fact these pointers belong to the second non-red column of a given array; after finishing the actual call to *ELIM-PC* this column will be the first column of the array, which shows that the same pointers cannot be inspected a second time in line 344. In lines 346 and 347, note that the computation of each column that has to be added to *Isol-PC* takes time $\mathcal{O}(\log(|D|))$, by Remark 5.19. The same remark shows that each of the operations mentioned above can be applied in time $\mathcal{O}(\log(|D|))$. Since the number of elements is bounded by $\mathcal{O}(|Q| \cdot |D| \cdot h_{\mathcal{D}})$ (general case) and $\mathcal{O}(|Q| \cdot |D|)$ (rigid queries, or labeling-complete queries over non-recursive databases) respectively, the result follows. □

5.7 Implementation and Evaluation Issues

The algorithm described in Section 5.4.2 has been implemented as a prototype in Java. The status of the implementation is described in the first part of this section. The algorithmic part was taken over almost literally, therefore we will only mention some basic facts about the implementation of the data structures. In the second part we will discuss some additional possibilities for optimization, that are partially based on existing work. We will end this section with remarks concerning the evaluation of the algorithm.

Status of the Implementation

The algorithm described in Section 5.4.2 has been fully implemented in Java 2 ([Sun99]) taking 4000 lines of code. Java was the language of choice due to its portability and special concern about XML and structured documents: Java has been developed to perform in a Web environment and incorporates therefore many

packages for treating structured documents, e.g. parsers for XML and HTML documents. A second motivation was the existence of classes implementing data structures that we need to implement aggregates. We used the Java-class `TreeMap` for implementing ordered association lists that are needed for fields in slots or for pointer columns in pointer arrays. `TreeMap` objects organize the elements as a balanced binary tree sorted by their identifiers, and thus guarantee logarithmic access, insertion and deletion. The third motivation was the expressive language provided for implementing graphical user interfaces (GUIs) that will be needed when implementing the graphical retrieval model proposed in Chapter 6.

The points mentioned in this work that haven't been included so far in the implementation are the following: (1) Implementation of the persistent data structures (cf. Section 5.2), (2) implementation of the optimizations (cf. Section 5.4.3), (3) implementation of a GUI, and (4) implementation of the exploration techniques described in Chapter 6. This means that the algorithm implemented so far is only a proof of concept, since it can not yet be connected to a persistent database storing the information about the documents, and the input and output is so far only text-based (what makes the concepts of aggregates in fact unusable).

The reason for not yet connecting the algorithm to a persistent storage is that the logical and physical storage of the data has to be taken care of with great expertise. Decisions have to be made, whether management of the index structures is handled with database management systems, be it relational, object-relational or object-oriented, or with dedicated text indexing systems like Altavista's Search Intranet Developer's Kit (Altavista SDK: [Alt99]). A comprehensive comparison of all design decisions that have to be made at this level, and the following implementation of the index structures on the basis of the chosen persistent storage medium was not possible in the given amount of time for this work.

We shall add that a new DFG project is dedicated to the storage of tree-structured data, mainly structured documents, and will implement and evaluate the concept and computation of complete answer aggregates fully, including persistent index structures and a GUI.

Additional Optimizations

We will now outline two general methods for optimization of the query evaluation process. One is based on the use of a grammar for query expansion and the second gives two possibilities for using filters to reduce the set of paths returned by the path selection index.

Using the Grammar

In many cases a grammar describing the hierarchical structure of the relational document structure is known, for example for all SGML and most XML documents a DTD (document type definition) is provided. If this structural description is known, it can be used in two ways for query optimization: (1) Matching the query with the grammar in order to detect unsatisfiability before beginning the computation of the complete answer aggregate, or (2) preprocessing the query with replacing certain soft edges by rigid edges.

[Kil92] discusses a method how to match the query against the productions in the grammar in order to detect inconsistencies. The basic idea is the following: If, for example, a query node has label A and one of its rigid children has label B, then there must exist a production of the form $A \Rightarrow \alpha B \beta$. This procedure can be extended to soft edges by applying it recursively.

The second idea, that is mentioned in [Kil92], is based on the observation that rigid edges are treated more efficiently than soft edges by his algorithms. This is

also the case for our algorithm (cf. lines 108-119 and 446-449). The first approach to this idea is to replace a soft edge in the query by a rigid edge. Unfortunately, it is only applicable in certain circumstances. Consider a tree query containing an atomic descendant formula $x \lhd^+ y$ and labeling formulae $M_1(x)$ and $M_2(y)$. If the grammar proves M_1 to be nonperiodic and restricts nodes with label M_2 to appear as children of nodes with label M_1 only, we can replace in the query the formula $x \lhd^+ y$ by $x \lhd y$ without changing the complete set of answers. In any other case this simple approach is not applicable.

A second approach concerns the transformation of tree queries that are not data-anchored into data-anchored queries by replacing soft edges by a chain of rigid edges. Unfortunately this is not possible. Let us first consider as an illustration the first query (on the left) and the relational document structure in Example 3.31. This query is not data-anchored. The replacement of the soft edge between p and v by a chain of rigid edges yields a data-anchored query: From an (assumed) grammar we can derive that `var` nodes can only be children of `body` nodes who themselves can only be children of `proc` nodes. But if we replace the soft edge by a chain specifying the `proc-body-var` sequence, we result, e.g., in the query on the right side, that has certainly different answers in the relational document structure from the original query. The reason for this difference is the periodicity of the `proc` label. We will now inspect the general case: Consider a query Q that is not data-anchored. Then there exists a variable x in Q with a periodic label M and an edge $x \lhd^+ y$. A database node d with label M may have descendants with label M in any level of the relational document structure. Therefore we cannot replace the soft edge $x \lhd^+ y$ by a sequence of edges, since one sequence (with fixed length) of rigid query edges cannot match all arbitrary long sequences of database edges.

Path Filtering Based on Labels

Query evaluation can be speeded up, if we can early discard paths that obviously cannot contribute to answers. In [MS99a] and [SM99] new methods for filtering the results of calls to index structures have been proposed for structured document retrieval systems. The basic idea is to check for every occurrence whether its inverted document path contains all labels that are required by the respective query path. This test is performed by adding to each occurrence stored in the path selection index additional information about the labels in its inverted document path as bit strings. These bit strings do not require much space and can be compared efficiently with the labels required by the query path. We show how to apply this generic mechanism to our algorithm: The path selection index returns for every call a set of inverted document paths whose initial node meets the requirements of the index call (lines 18 and 19). The additional label additional information is attached to every path (during database construction) and matched (during query evaluation) against the label requirements imposed by the actual query path. Database paths not meeting these requirements are removed from the path set and not further treated by the algorithm. Since the label information is encoded in bit strings, matching can be executed very efficiently and storage of the additional label information is not very space consuming.

Additionally, [MS99a] and [SM99] provide compression techniques for the added bit strings and means to analyze the added value of integrating information about a given label into the index structure. Labels with not enough added value can be neglected what makes the additional space demand smaller and the comparison more efficient.

Path Filtering Based on Nodes

We will now propose a second modification that allows to early filter out inverted document paths that cannot contribute to answers. In contrast to the last modification, path filtering based on nodes refers to the nodes that have already been entered into the aggregate at a given time.

When treating a new query path π_Q, the bottom-most variable x of π_Q that has occurred already in one of the earlier query paths is computed. Here Agg_x is the bottom-most record with marker "old" in π_Q. (For the first query path treated during query evaluation no such record exists and the optimization cannot be applied in this case.) An inverse document path $\pi_D \in \Pi(\pi_Q)$ can only be entered successfully if it contains a node d such that Agg_x has a field $Agg_x[d]$. Let us call such a path *relevant*. We show how to compute the subset of relevant paths of $\Pi(\pi_Q)$ in linear time.

Let $\langle d_1, \ldots, d_m \rangle$ denote the sequence of nodes of the record Agg_x, in pre-order enumeration. An element $d \in \{d_1, \ldots, d_m\}$ is called *maximal* iff no ancestor of d belongs to $\{d_1, \ldots, d_m\}$. For each node $d_l \in \{d_1, \ldots, d_m\}$, the set of descendants defines an interval of the form $[d_l, d_l^*]$ of the pre-order relation on $\mathcal{D}$. Node d_l^* is the right-most leaf among the descendants of d_l and can be obtained in constant time (cf. Remark 5.18) using the formula $d^* = pred_p(minsucc\ (d))$. (The expression $pred_p(e)$ stands for the predecessor of e with respect to pre-order relation $<_p$.) Two intervals $[d_l, d_l^*]$ and $[d_h, d_h^*]$ are either disjoint or one is contained in the other. The intervals of maximal elements are pairwise disjoint and cover all the intervals of nodes in $\{d_1, \ldots, d_m\}$. The sequence of all maximal elements d_l, together with the right boundaries d_l^* of their intervals, can be computed in time $\mathcal{O}(m)$. In fact, d_1 is always maximal. Once we have found that d_l is maximal, the first element of $d_{l+1}, \ldots, d_m$ that does not belong to $[d_l, d_l^*]$ is the next maximal element of the sequence.

Obviously an inverted document path $\pi_D \in \Pi(\pi_Q)$ with initial (bottom-most) node e is relevant iff e is in the pre-order interval $[d_l, d_l^*]$ for a maximal node d_l of $\{d_1, \ldots, d_m\}$. Recall that the path selection index returns each set $\Pi(\pi_Q)$ ordered according to the pre-order of the initial nodes. Let $\langle e_1, \ldots, e_n \rangle$ denote the sequence of all initial nodes of paths in $\Pi(\pi_Q)$, ordered in this way. Using one simultaneous traversal of (the intervals associated with) the subsequence of maximal nodes in $\langle d_1, \ldots, d_m \rangle$ on the one hand and $\langle e_1, \ldots, e_n \rangle$ on the other hand we may filter out the list of all relevant paths in time $\mathcal{O}(n + m) \leq \mathcal{O}(|D|)$.

Since the number of query paths is bounded by $|Q|$ the total time-complexity of all filtering steps is bounded by $\mathcal{O}(|Q| \cdot |D|)$. This shows that the bound for the worst-case complexity of the algorithm is not affected by this filtering. But we can expect an improvement of the average runtime since irrelevant query paths are detected early, i.e. before it is tried to align them with the query path.

A similar idea simply compares the target candidates in the root record Agg_y (y is the root of the query) with the root nodes of the inverted document paths returned by the path selection index. Inverted document paths $\langle e_k, \ldots, e_0 \rangle$ with a root node e_0 not occurring in Agg_y cannot be successfully aligned with the query path and are not treated further. The comparison between the root nodes and the target candidates in Agg_y can be computed in time $\mathcal{O}(|\mathcal{D}|)$ as before, since the inverted document paths are ordered.

Evaluation Issues

An exhaustive evaluation of the algorithm was neither the goal of our work nor was it possible in the given time for the following reasons: An elaborate evaluation of

the algorithm's efficiency requires very careful handling of the underlying storage of the index structures as discussed in Section 5.7.

Another major problem for the evaluation process is the availability of adequate document collections. In order to exploit the added value of our formalism (as compared to other, simpler structured document retrieval systems), the document collection has to be highly structured on a semantic level beyond layout markup. This is the case for legal documents like law collections, for linguistic data like corpora with parsed text, or for other scientific or technical XML and SGML applications. Appropriate collections exist but are not easy to access since they often contain internal company knowledge, that is not destined to be distributed externally. Other evaluation studies (e.g. [FK99] and [WCB+00]) used artificial documents that were constructed randomly for evaluation purposes only on the basis of statistical assumptions. We do not think that this procedure can reflect actual document collections. But it serves well to measure the basic behaviour of the underlying system under varying parameters.

We are aware of the fact that a new form of answer presentation as presented in this work does not only need exhaustive efficiency measuring but extensive user acceptability studies even more. Only if users accept the notion of complete answer aggregates and can exploit the added value provided, the formalism will prove successful. But user acceptability studies are a time- and money consuming process that certainly are beyond the scope of this work.

Nonetheless, the development of the notion of complete answer aggregates led to the establishment of a DFG project that will be concerned with the use of complete answer aggregates for retrieval of structured documents. One of the project goals is the evaluation of a system on the basis of real-world document collections.

5.8 Related Work

In this section we discuss work related to the algorithm for evaluating queries. We first compare our algorithm to Kilpeläinen's original Tree Matching algorithms and the similar behaviour of tree automata. In the second part we review results of two preceding works concerned with complete answer aggregates, that have better time complexity results for queries without constraints. In the second part we will make some comments on the relation between complete answer aggregates and constraint networks.

5.8.1 Kilpeläinen's Tree Matching Algorithms and Bottom-Up Tree Automata

Kilpeläinen presented in [Kil92] a set of algorithms for the original Tree Matching formalism, each algorithm dedicated to a special Tree Matching problem, e.g. ordered/unordered tree/path inclusion. He showed the unordered tree inclusion problem to be NP-complete. In Chapter 2 we saw that even for the problem classes where the decision problem is not NP-complete there are $\mathcal{O}(n^q)$ answers to a given query and tree database due to the permutational phenomena. We could improve these negative results with minor modifications in the notion of answers (cf. Section 3.4) and with the introduction of complete answer aggregates. Concerning the algorithms we can observe the following: Kilpeläinen's algorithms are very similar to bottom-up tree automata (e.g. [CDG+97, Pre98, GS84]) that visit each node of the input tree (the database) beginning with the leaves. The nodes of the input tree are assigned states and a transition maps the states of the children of a node v together with an alphabet symbol attached to v to a state that is assigned to v. If the tree automaton reaches the root and assigns a state to it that is an accepting state,

the tree is accepted by the automaton. Kilpeläinens algorithms work very similar for each database tree: Starting with the leaves they wander upwards through the database tree, attaching information to the nodes of the tree. In each upwards step the information attached to the children of a node v and the label of v are compared with the requirements in the query, thus generating a piece of information that is attached to v. In fact, the information attached to the database nodes are exactly the nodes of the query tree that are candidates for a possible match at the respective state of query evaluation. If the root could be reached successfully, a matching tree is found.

We can observe an interesting symmetry between Kilpeläinen's algorithms and tree automata on the one hand and our algorithm on the other hand: The former wander through the input tree (database tree) attaching information (states or query nodes, respectively) to its nodes depending on the transition function (query tree), whereas the latter wanders through the query tree, attaching information (database nodes) to its nodes (represented as slots in an aggregate) depending on the relational document structure. If we neglect the fact that our algorithm computes complete answer aggregates and assume that complete answer aggregates are just an intermediary data structure out of which the single answers are produced as the actual query result, it seems hard to say which approach is in fact more efficient. We can expect the data structure of Kilpeläinen's algorithms to be very big especially in the beginning of the computation. But since we are explicitly concerned with computing answer aggregates, we are anyway bound to follow the approach of attaching information to query nodes (slots).

5.8.2 Queries Without Constraints

There are other complexity results for the computation of complete answer aggregates for more special cases. These results do not include the use of constraints and are to be found in [Meu98] and [MS00]. In both articles the pointer columns are organized as arrays instead of balanced trees. Since in the situation of these articles, horizontal pointers are not considered (due to the lack of order constraints), we only have direct access to pointer columns indexed by a target candidate d. If horizontal pointers are involved we also need to find the next non-red pointer column, what can not be performed on arrays as efficiently as on balanced trees. Direct access to given pointer columns organized as arrays is in constant time what helps to improve on efficiency for the case without constraints. [MS00] discusses complete answer aggregates for general graph queries on graph databases. As a special case, it gives a time complexity of $O(q \cdot n \cdot h_{\mathcal{D}})$ for computing a complete answer aggregate for tree queries without constraints. In the case of rigid queries, [Meu98] gives an even better time complexity of $O(q^3 \cdot n)$. This complexity can be improved with a change of data structure and some algorithmic techniques used in this work to $O(q \cdot n)$. The relation between these two time complexity results is the same as stated in Theorem 5.20: When the query contains no soft edges, the factor $h_{\mathcal{D}}$ disappears in the complexity results.

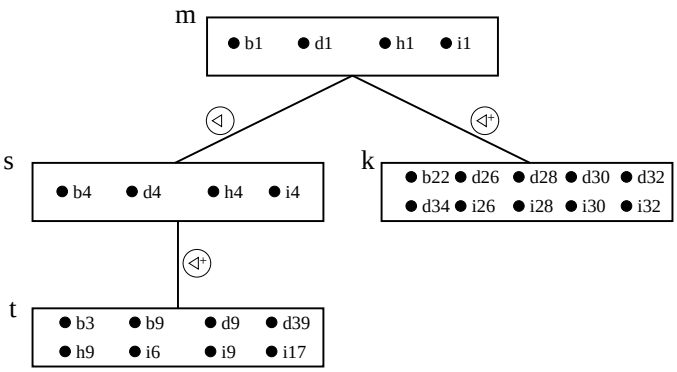
5.8.3 Arc-Consistency and Constraint Networks

In this section we investigate the close relationship between complete answer aggregates and constraint networks as discussed, e.g., in [Mac77, Mon74, MF85, MH86]. First we recall the basic definitions for constraint networks.

Constraint Networks

Definition 5.21 Let $\mathcal{D}$ be a relational structure (in the general sense), i.e. a domain D with a set $C_{\mathcal{D}}$ of binary relations $c_i \subseteq D \times D$. Then a *constraint network* over $\mathcal{D}$ is a tuple (Y, Dom, C) where Y is a finite set of variables, $Dom : Y \to 2^D$ a function assigning every variable a subset of D as domain and C a set of triples $\langle c_{i_j}, x_{k_j}, x_{l_j} \rangle$ stating that variables x_{k_j} and x_{l_j} are connected with a c_{i_j}-constraint ($c_{i_j} \in C_{\mathcal{D}}$, $x_{k_j}, x_{l_j} \in Y$). A *solution* of a constraint network is a variable assignment ν for the variables in Y so that for all triples $\langle c, x, y \rangle \in C$ the relation $c(\nu(x), \nu(y))$ holds in $\mathcal{D}$.

Example 5.22 The following illustration depicts a constraint network (Y, Dom, C) with binary relations $\lhd$ and $\lhd^+$. The domain is the set of nodes in the "Movie Collection" in Appendix B.2.



(Y, Dom, C) contains four variables $m, s, t, k \in Y$. Dom assigns domains to the variables, for example $Dom(m) = \{\texttt{b1}, \texttt{d1}, \texttt{h1}, \texttt{i1}\}$. $C = \{\langle \lhd, m, s \rangle, \langle \lhd^+, s, t \rangle, \langle \lhd^+, m, k \rangle\}$ describes the constraints between the variables. If we assume that the relation $d \lhd e$ ($d \lhd^+ e$, resp.) holds iff $d \rightharpoonup e$ ($d \rightharpoonup^+ e$) in the "Movie Collection", then the following variable assignment is a solution to (Y, Dom, C):

$$\{m \mapsto \texttt{i1}, s \mapsto \texttt{i4}, m \mapsto \texttt{i9}, m \mapsto \texttt{i30}\}$$

One goal in the field of constraint networks is to find "minimal" networks in the sense that only elements that can contribute to solutions are contained in the domains of the variables. In this effort various notions of consistency have been investigated, together with algorithms for computing "minimal" networks. We will describe the notion of arc-consistency, which formalizes a local form of minimality:

Definition 5.23 A constraint network (Y, Dom, C) is *arc-consistent* if for each triple $\langle c, x, y \rangle \in C$ the following conditions hold:

1. for each $d \in Dom(x)$ there exists $e \in Dom(y)$ such that $c(d, e)$ holds in $\mathcal{D}$, and
2. for each $e \in Dom(y)$ there exists $d \in Dom(x)$ such that $c(d, e)$ holds in $\mathcal{D}$.

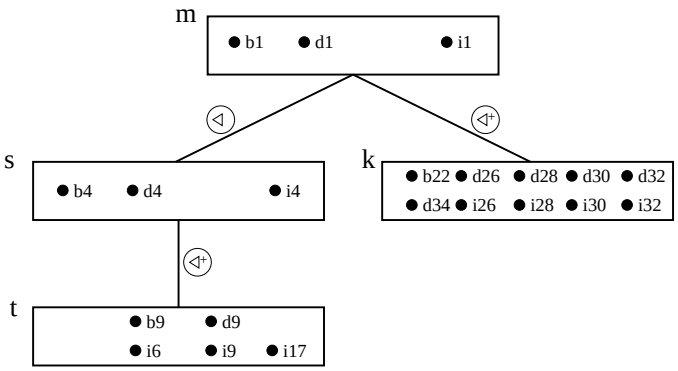
Note that, in general, arc-consistency does not imply global minimality in the sense that every element contributes to a solution. There are other, stronger notions of consistency implying global minimality, but which are not computable efficiently. Nonetheless, for the special case of tree-structured constraint networks arc-consistency does imply global minimality as defined above.

A constraint network (Y, Dom', C) is a *refinement* of (Y, Dom, C) if $Dom'(x) \subseteq Dom(x)$ for all $x \in Y$. For each constraint network (Y, Dom, C) there exists a

unique refinement (Y, Dom', C) that is arc-consistent and maximal in the sense that every other arc-consistent refinement of (Y, Dom, C) is a refinement of (Y, Dom', C). This network will be called the *maximal arc-consistent refinement* of (Y, Dom, C). There exist various "arc-consistency algorithms" that compute the maximal arc-consistent refinement of a given constraint network (Y, Dom, C) ([MH86, MF85]). The algorithm in [MH86] needs time $O(e \cdot n^2)$ where e is the number of constraints in C and n is the maximal cardinality of a domain $Dom(x)$.

Example 5.24 The constraint network in Example 5.22 is not arc-consistent. We can see that, for example, `h1` $\in Dom(m)$ but there is no node e in $Dom(k)$ so that e is a descendant of `h1`. This means that the first condition in Definition 5.23 is violated.

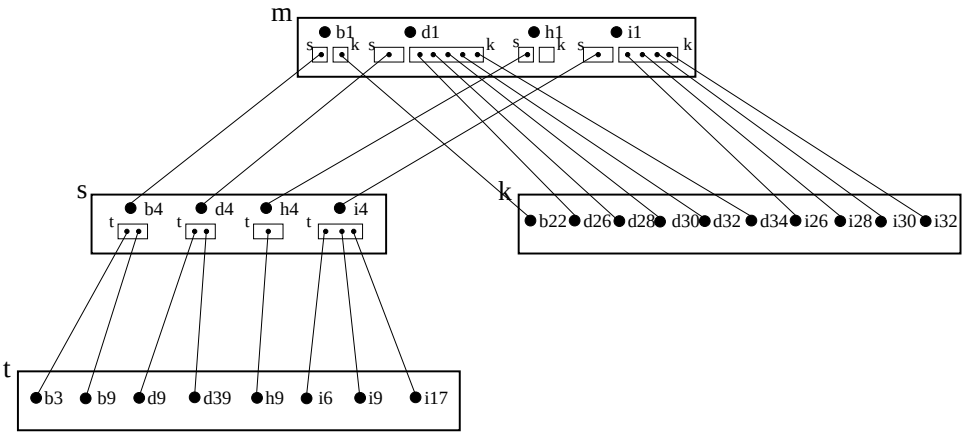
The following picture illustrates the maximal arc-consistent refinement (Y, Dom', C) of the constraint network in Example 5.22. Nodes that violated the conditions in Definition 5.23 have been removed.



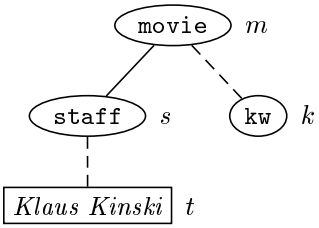
Arc-Consistent Constraint Networks and Complete Answer Aggregates

There exists a simple correspondence between a tree-shaped constraint network (Y, Dom, C) and an aggregate Agg. The variables in Y correspond to query variables, the target candidates in a slot correspond to the elements in the domain of a query variable and the constraints (edge and other binary constraints) in the query correspond to the constraints of the constraint network. Aggregates can be conceived as constraint networks, if we chose to remove the explicit links between target candidates. In the other direction, a constraint network yields an aggregate if we explicitly add the links using the information about the relations extensions.

Example 5.25 The constraint network in Example 5.22 corresponds to an aggregate



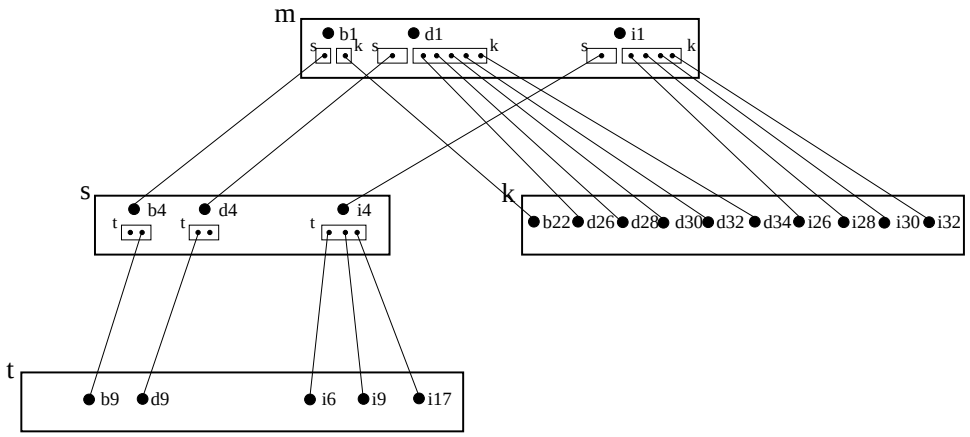
for the following query (from Examples 4.10 and 4.16) retrieving keywords of movies where *Klaus Kinski* was in the staff:



(Note that the above aggregate is not the complete answer aggregate for the query and relational document structure in Appendix B.2.)

If we exclude relations and constraints (in the sense of Definition 3.8), it is also fairly obvious that a complete answer aggregate corresponds to an arc-consistent constraint network. If we define for a given query the variables of the network as the query variables, the constraints as the edge constraints of the query and the domains for all variables in the constraint network as the set of all nodes in the document database having the appropriate label as stated in the query, an algorithm computing the maximal arc-consistent refinement for this network yields the complete answer aggregate, if we add the links between target candidates.

Example 5.26 The complete answer aggregate for the query in Example 5.25 and the relational document structure in Appendix B.2 is



Note the correspondence between the complete answer aggregate and the arc-consistent constraint network in Example 5.24. The only difference is that in the complete answer aggregate the pointers between the target candidates are represented explicitly, whereas in the constraint network this information remains implicit.

From the above discussion we can conclude that the algorithm in Section 5.4 is an arc-consistency algorithm for the special case of tree-structured constraint networks. And in fact, a closer inspection of AC-2 ([MH86]), an algorithm for computing the maximal arc-consistent refinement of a constraint network, reveals similarities on an algorithmic level. This algorithm is based on the notion of "support": For every element in a domain x information is kept, to how many elements in another domain y it stands in a c-relation, if the set C of constraints in the network contains the triple $\langle c, x, y\rangle$. If this value drops to zero the element has to be removed. This includes the decrease of other elements' support values. The algorithm keeps for every element a table, which other elements stand in a c-relation to the given element. A lookup

in this table reveals the elements whose support is decreased if an element has to be removed. The table together with the support values corresponds to the links that are attached to target candidates in aggregates. The situation that an element's support value drops to zero corresponds to the situation that a target candidate becomes isolated.

The fact that the computation of a complete answer aggregate can be performed in time $\mathcal{O}(q \cdot n \cdot h)$ if we have no constraints (see Section 5.8.2), what corresponds roughly to the time complexity $\mathcal{O}(q \cdot n^2)$ of AC-2 on tree-structured constraint networks, emphasizes the correspondence between the two concepts and algorithms.

Nonetheless, we think that even in the presence of general arc-consistency algorithms for constraint networks with similar time complexity, our algorithm for computing the complete answer aggregate exploits the special peculiarities of structured documents better and has therefore advantages over the general arc-consistency algorithms:

- Our algorithm can treat order constraints. Although arc-consistency algorithms for constraint networks can treat all kinds of constraints, including order constraints, the notion of arc-consistency is, due to its limitation to local inconsistencies, not as strong as the notion of complete answer aggregates if order constraints are involved. This is due to the fact that the incorporation of order constraints gives the constraint network an overall structure that is in general not a tree structure. In this case the maximal arc-consistent refinement of a constraint network may contain elements that cannot contribute to a solution due to global inconsistencies.

- In our algorithm the construction of the aggregate and the insertion of target candidates is interleaved with the removal of isolated target candidates, whereas in arc-consistency algorithms the two processes are divided. Interleaved construction and removal has the advantage that many target candidates can be removed early, thus speeding up the computation, and that many nodes are not entered into the aggregate at all since the alignment process stops beforehand.

- Our algorithm is constructed in a way that it can exploit the index structures in many ways. This gives a great advantage against the generic arc-consistency algorithms that do not provide an interface to index structures.

Chapter 6

Using Complete Answer Aggregates

In this chapter we discuss the possible uses of complete answer aggregates with their benefit and added value for the user in the querying process. Since the retrieval of structured documents has a strong similarity to queries in Database Systems (DBS) on the one hand and querying Information Retrieval (IR) systems on the other hand, we will first take a closer look at the retrieval process in these two paradigms.

6.1 The Retrieval Process in DBS and IR

We can now describe the retrieval processes in DBS and IR in more detail. DBS[1] ([Ull89]) store data in the form of relations. Retrieving data in DBS means to locate[2] the data in the relations. A query describes the relational properties the data has with respect to the schema of the data stored. IR systems ([BYRN99b, vR79, Fuh92, SM83, FBY92b]) store information in the form of documents. Retrieving informations means to locate documents containing the information being searched for. A query describes the information need of the user, in most cases in the form of keywords that describe the relevant documents.

Starting point for the retrieval process in both DBS and IR is an information need of the user. Before describing the retrieval process in more detail we will review a classification of information needs by Mizzaro ([Miz98]). This framework will elucidate the peculiarities of the retrieval process in DBS on the one hand and IR on the other hand.

Mizzaro distinguished four manifestations of information needs:

Real information need (RIN) expresses the piece of information that a user needs to solve a "problematic situation".

Perceived information need (PIN) is a cognitive representation of the RIN and can differ from the RIN due to incorrect perception or representation.

Expressed information need (EIN) is the utterance or representation in natural language of the the PIN.

[1] We refer in this chapter to relational DBS that represent the bigger part and the industrial mainstream in DBS.

[2] We only refer to the pure retrieval aspect of DBS, neglecting the construction and manipulation aspects of queries in DBS.

Formalized information need (FIN) is the formalization of the EIN in the form of a query that is a submitted to an IR system.

During the retrieval process the information need undergoes transformation processes from the RIN to finally the FIN. In iterative retrieval models (used in IR) this process is also subject to feedbacks, since a user may change his or her PIN after inspection of a query result.

We can observe that DBS only take the EIN and FIN into account. This is a reasonable assumption since queries to DBS operate on purely syntactic, uninterpreted structures. The retrieval process is linear, i.e. the user formulates a query then inspects the result, a set of tuples, afterwards. The user has to know the structure (schema) of the data and the query syntax in order to be in a position to formulate the query, i.e. to transform his or her EIN to a FIN. This transformation process can be supported, for example in QBE (Query by Example, [Zlo75]) or with DataGuides ([GW97]), a formalism helping the user to formulate queries in Lore ([MAG+97]) a database system for semistructured data.

In IR, the retrieval process is typically an iterative process. The user formulates the query (containing little or no structure at all), inspects the result and then reformulates the query. This reformulation is based on the impact of the result inspection on PIN, EIN, or FIN, i.e. the user may change his or her perception of the information need, the cognitive representation of the information need or the formalization of the information need. A change in one of the three manifestations entails in general a change in the query formulation. The involvement of RIN and PIN in the IR process, as opposed to the DBS retrieval process, can be explained by the different nature of what is searched for: DBS are concerned with data, a syntactical concept on the level of FIN, whereas IR supports the user in searching for information, a concept clearly beyond the levels of expressed or formalized information needs. The result of a query is ordered according to the computed relevance of the single answers with respect to the formulated query. This fact again owes to the very special nature of information, as opposed to data. Information accessed in IR is poorly structured or contains no structure at all. It is accessed in an imprecise way by describing the content of a document, e.g. by stating keywords. It is by no means obvious, neither for man nor for machine, what content a document contains and which documents are relevant with respect to an information need. Since there is no sharp boundary between relevant and irrelevant documents to a query, imprecise measures of relevance proved itself as a useful and natural tool to deal with this problem. The result can then be ranked according to the imprecise relevance values.

Since the retrieval of structured documents incorporates aspects from DBS as well as from IR, we should aim towards a retrieval process that reflects the peculiarities of both the DBS retrieval process and the IR retrieval process, the data-centered view of DBS and the user-centered view of IR. We will elaborate in the sequel how complete answer aggregates support this.

6.2 An Iterative Two-Step Retrieval Process

We will introduce a model of an iterative two-step retrieval process that allows the user to reformulate the query and investigate the query result on a graphical level. We put a focus on the graphical interaction, since we think a visual approach can directly address the perceived information need and thus support the user in his navigation and manipulation in a complex information space as constituted by structured documents. The model developed here is currently in the process of being implemented and should be understood as a requirements analysis rather than a system description. Other research on visual interfaces for DBS and IR systems

(e.g. [GW98, GW97, CCD+99, BYRN99c, Shn98, ABY98, BYNVdlF98, CSC97, CCLB95, GPT93]) confirms this deliberation.

6.2.1 The Sphere of Interest

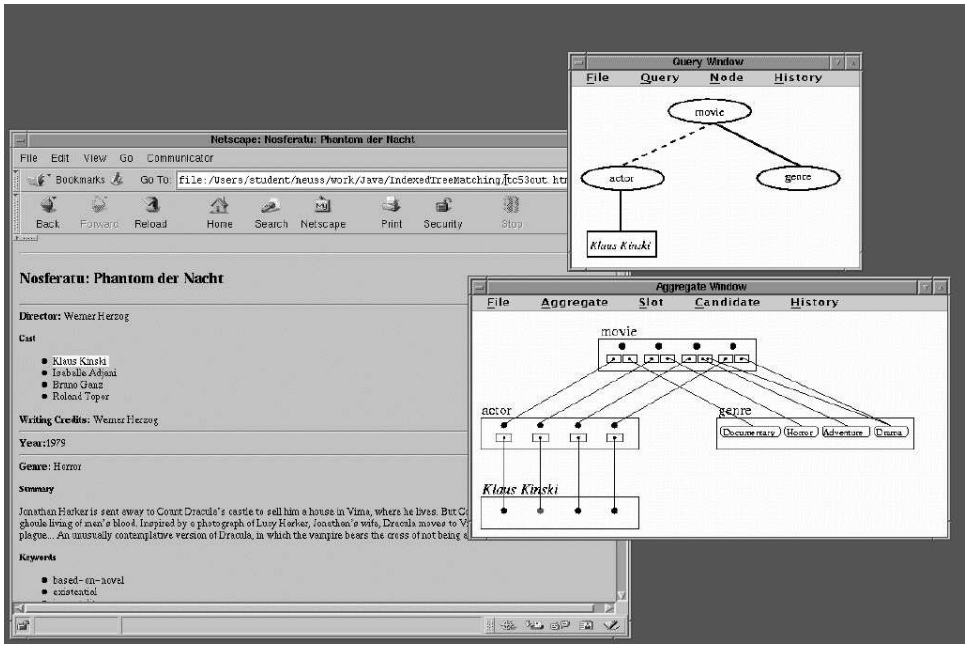
We divide the retrieval process into two steps: In a first step, the user defines with a rich query (i.e. many nodes with few restrictions) a *sphere of interest* represented by the complete answer aggregate as the retrieval result. In a second step, this complete answer aggregate is explored and manipulated in order to explore the topology and information contained in the sphere of interest. Examples for exploration and manipulation techniques are given in this chapter. All of these techniques operate on a graphical level in order to support the user in intuitively gaining more insight into the sphere of interest. With this new insight the user is in a position to reformulate the original query (based on changes in the PIN, EIN, or FIN) and thus iterate the retrieval process until the query result is satisfying.

Goldman and Widom proposed in [GW98] a similar model for an iterative two-step retrieval process for querying semistructured data. We compare it with our model in Section 6.2.5.

With the use of global comparator functions and displaying the node database in a special tree window, the user may leave to a certain extent the local sphere of interest defined by the complete answer aggregate if properties of target candidates have to examined that are not accessible in the complete answer aggregate.

6.2.2 The Graphical User Interface

The graphical interaction with the retrieval environment is a key aspect that supports the user in his or her retrieval task. We will briefly discuss the graphical user interface (GUI). The following picture illustrates a non-functional prototype of the GUI.



The GUI is divided into distinct windows, each of them dedicated to visualize different types of objects:

- The **query window** depicts the user query in a graphical tree representation. It allows interactive query formulation by constructing the query tree in a style like file browsers by adding or removing children to selected nodes. Constraints, labels, variables, and additional information like attributes can be formulated in a hybrid graphical and textual way by selecting nodes and entering the corresponding text.

- The **aggregate window** shows the sphere of interest (the complete answer aggregate) defined by the user query. Target candidates, edges and slots may be selected with the mouse.

- The **text window** shows textual content of selected nodes. This may also involve attribute values or other textual information attached to nodes. The text can be displayed in a formatted way, supposed that stylesheets exist. In the illustration above a Web browser is used as the text window.

In addition to these three central window classes the following three may be employed for more sophisticated tasks. Some of the features of theses two advanced window classes require IR techniques that have not been discussed in this work. We will not go into detail for these techniques, since we would only like to mention that it is possible to mix these advanced techniques with the retrieval model introduced here.

- The **tree window** can depict relational document structures. This can either be the underlying document database or parts of it, chosen for example by selecting a node in the aggregate window.

- The **topology window** can arrange nodes according to a given distance measure on a two-dimensional plane. A common distance measure is the semantic relatedness that can be computed by IR techniques based on keywords. The nodes displayed here may be all nodes from document database, or a subset defined by selecting nodes in one of the other windows.

- The **ranked aggregate window** can display some relevance information about how relevant a target candidate is for the given part of the query. This can visualized by changing the order, size, texture, etc. of the target candidates in the aggregate. There exists no formal account yet on how to integrate the notion of relevance into the logical Tree Matching framework wit complete answer aggregates, see Section 7.2 for a short discussion.

There may be more windows of the same class depicting distinct or the same objects in different views. Interaction between the windows is organized by selection of an object in one window and sending it to another window. This process may create dependencies between windows or objects in distinct windows, that may trigger automatic updates on certain changes. Objects can be displayed in different ways in the windows, reflecting different views upon the object.

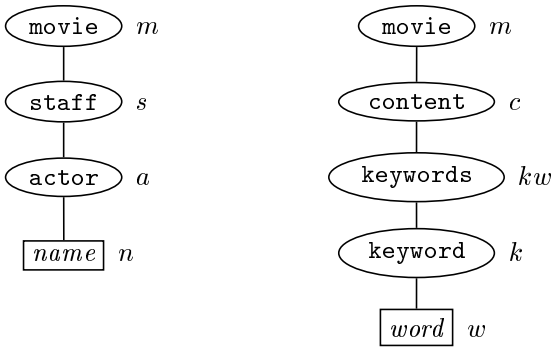
6.2.3 Queries

Queries are entered in a graphical way into the query window. Two distinct queries may be combined by inserting one query as subtree under a selected query node of the other query. We use query nodes without variables to express existentially quantified nodes (see Section 3.3.2).

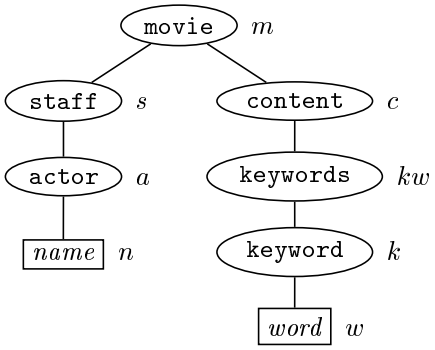
In order to support the user in query formulation two possibilities come to mind: In **grammar-directed** query formulation the user is only allowed to formulate queries that have a solution in a relational document structure described by the

grammar[3] (see also Section 5.7). If the user wants for example to add a child to a node that cannot have children, he is informed by the system that this is not possible. A tree or text window may display the grammar and highlight the position where the conflict occurred. The second possibility, **template-directed** query formulation, uses pre-formulated queries that can be used as templates for further reformulation. Templates are collected in an annotated template library and may be combined.

Example 6.1 The following two query templates describe actors in a movie and keywords of a movie, respectively:



They can be combined to the following query:



6.2.4 Exploration Techniques

This section describes techniques for exploring the sphere of interest, i.e. complete answer aggregates. This involves presenting complete answer aggregates in different views. The original complete answer aggregate defined by the corresponding query is not changed, but only presented in different ways to the user. Nonetheless, we sometimes speak about removing target candidates from an aggregate, although we mean only the removal form the actual view. The view onto a complete answer aggregate can be changed incrementally resulting in a view that is not a complete answer aggregate, not even an aggregate in the strict sense of the definition. But we refrain from reformulating the definitions in order to handle the more general structures treated here, since the similarities to aggregates are close enough to be obvious.

Most of the following techniques have their counterpart in the relational algebra or in SQL-like languages. This relation is discussed in Section 6.2.5. Note that most operations could also be part of the query language, instead of triggering them interactively.

[3] This requires, that a grammar describing the structure of the document database is known.

In some of the techniques introduced in the sequel we make use of an external function f on database nodes or target candidates. This function maps database nodes or target candidates onto a given set in order to make them comparable. With the help of this function we can perform for example sorting or grouping of target candidates based on various criteria, defined by the function f. If the domain of f is the set of database nodes, we say that the corresponding operation is *global* because then the function can access all properties of the nodes stored in the node database, e.g. number and identity of children, attributes, etc. If the domain of f is the set of target candidates, then the corresponding operation is *local* since the value of f can only depend on information accessible in the complete answer aggregate, e.g. number and identity of children of a target candidate (i.e. target candidates pointed at with horizontal pointers). This differentiation is useful since in some cases the user want to refer to the sphere of interest only ("How many movies of interest...?") and sometimes to the document database as a whole ("How many movies in total...?"). For the sake of reference we call these functions (local or global) *comparators*. Comparators can (and in most cases have to) be partial functions. If the comparator value of a node or target candidate is needed but not defined the operation can not be performed.

We use comparators in three operations: (1) For sorting, target candidates are mapped to natural numbers, e.g. the textual content (representing a date) to an integer representation, or mapping a target candidate to the number of children it has (in the aggregate or in the node database). (2) For restriction, target candidates are mapped to Boolean values depending on certain properties of the target candidates, e.g. the truth value of "its contained text represents a date that is newer than '1.1.80"', or of "it has at least three children (in the aggregate or in the node database)". (3) For grouping, target candidates are mapped to an arbitrary domain representing the property on the basis of which target candidates are grouped, e.g. an integer domain where the target candidate is mapped to the number of children it has (in the aggregate or in the node database), or a string domain where the aggregate is mapped to the text it contains.

Note that comparator functions are in general invisible to the user. They are only used as a means to explain the following techniques in a simplified way. The user chooses operations like `sort` or `group` without having to use or even know about the existence of a comparator function. Comparators bear some resemblance to the concept of type coercion, for example used in the Lore system ([MAG+97, AQM+97]), a database system for semistructured data.

The explanation of the exploration techniques will be based on examples in the most cases, in order to point out the advantage of having a visual manipulation language.

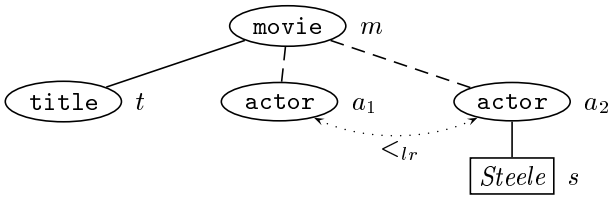
Degrees of Detail

When a complete answer aggregate is presented the user has the choice of how detailed it is presented. Every slot may contain

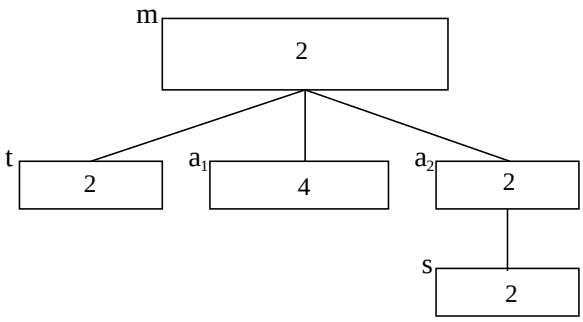
- only the number of target candidates residing in it,
- the target candidates but not the links between them,
- the target candidates with the vertical links, or
- the target candidates with all links (horizontal and vertical).

It is possible to specify for each slot on its own the way it presents itself to the user.

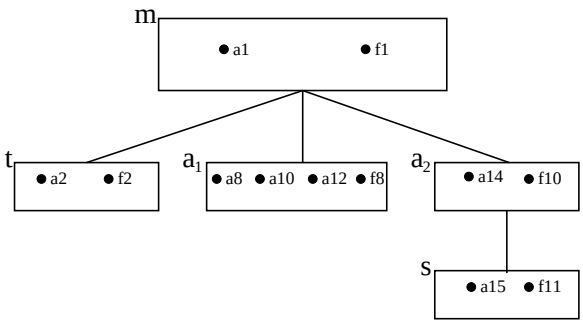
Example 6.2 Consider the query Q retrieving all actors having costarred with *Barbara Steele* in a role more important (reflected by the order in the enumeration of actors) together with the respective movie title:



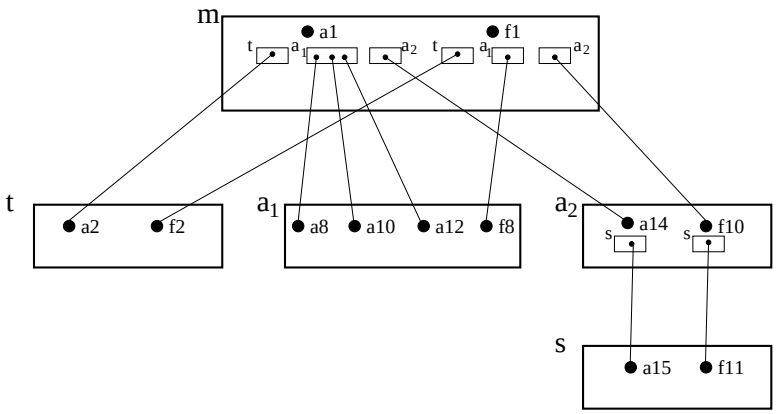
We give as examples the four views upon the complete answer aggregate for Q and the relational document structure in Appendix B.2. We begin with the less detailed degree, showing only the number of target candidates in each slot.



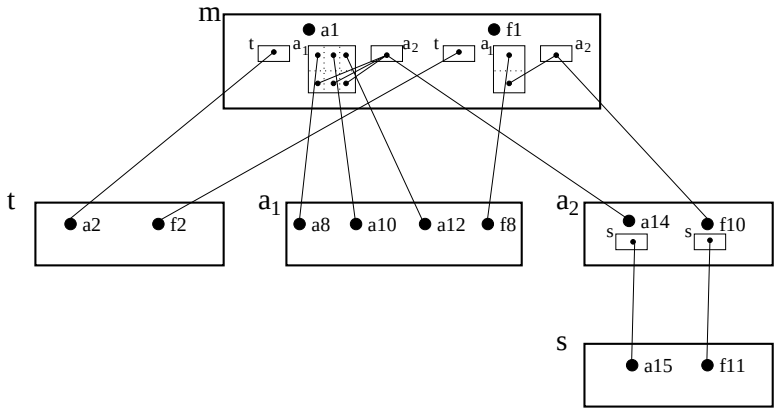
In a more detailed view the user can see the target candidates and has access to them for further exploration and manipulation.



The next aggregate shows the links between the target candidates that represent the local dependencies.



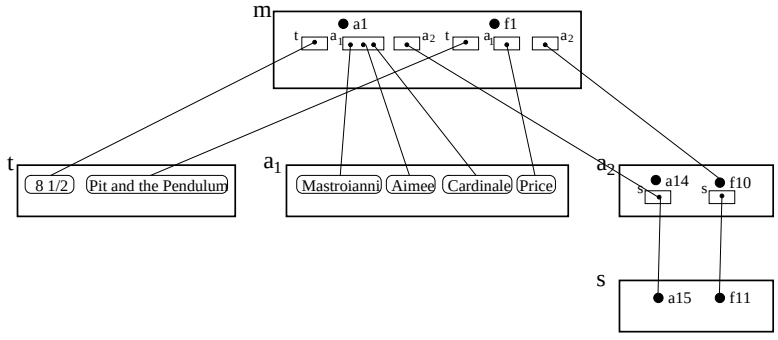
The last aggregate shows all information including the order information maintained by the target candidates:



Textual Content

So far the (intermediate) retrieval result presented with complete answer aggregates is on a very abstract level: The user does not know the textual content of the presented target candidates. This situation resembles an answer to a search engine query to the World Wide Web, where the user is presented a list of links as answers. Only by following the links he or she will see the actual content of the documents. We will present a similar mechanism with a variation of two possibilities: The user sends the content (or attributes) to a text window, or displays it in the complete answer aggregate itself. The latter possibility, which may involve technologies like keyword extraction used in IR and Web search engines, or a simple abbreviation of the participating strings, is illustrated in Example 6.3. For the former option the user selects one or more nodes in the complete answer aggregate (or a tree window or topology window, if this is supported) and sends them to a text window. The text window displays the textual content or attributes, if possible in formatted style. As an option the text of the whole document containing the selected target candidate is displayed and the text belonging to the selected target candidate is highlighted.

Example 6.3 The following aggregate shows the textual content of some of the target candidates:



Controlled Enumeration of Answers

In some cases the user may want to have an enumeration of all answers. He or she may restrict the enumeration to contain one ore more selected nodes in the aggregate. The result of the enumeration is displayed in a text window.

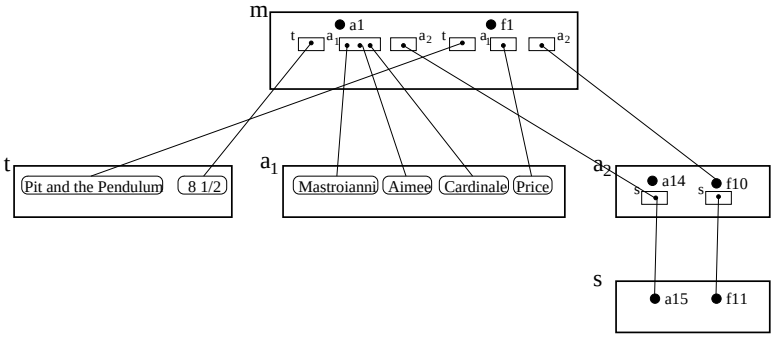
Slot Hiding

In order to gain a better overview slots may be hidden. This corresponds to the existential quantification of the respective variable discussed in Section 3.3.2. If a slot is hidden, it may be necessary to inherit the links of its target candidates. The root slot may not be hidden. In Example 6.3 the user may want to hide slots a_2 and s in order to have a better overview, since he or she already specified in the query that *Barbara Steele* shall be contained in the text nodes appearing slot s and thus loses no information in hiding this part of the aggregate.

Sorting

It is possible to change the order of the target candidates in a slot of a complete answer aggregate. This order can follow semantic values like dates or numbers that are contained in elements or attributes, or it may refer to structural properties, i.e. the number of children/links a target candidate has. This depends on the nature of the actual comparator f that has to map target candidates or database nodes to natural numbers. The target candidates in the slot are sorted according to their comparator values. An example for local ordering based on structure can be found in Example 6.13.

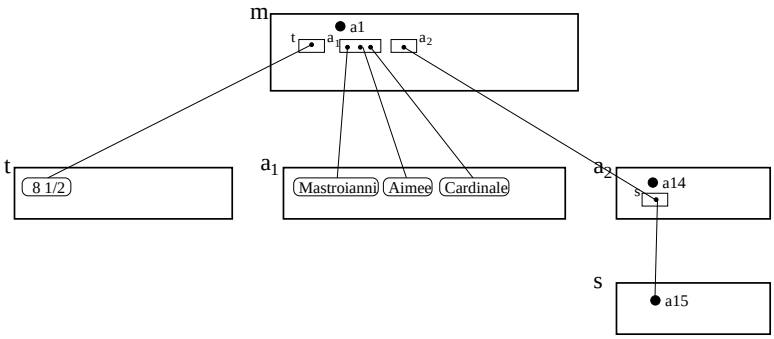
Example 6.4 Consider the complete answer aggregate in Example 6.3. The user may want to sort the movie titles in slot t according to the year of production. This is a global ordering, since the year information is not locally available in the complete answer aggregate: The node database has to be employed. The underlying comparator maps every database node to the integer value of the textual content of a sibling with label year. The resulting view on the complete answer aggregate is the following:



Hiding Target Candidates

In order to improve the overview the user may want to exclude selected target candidates from the aggregate. The aggregate window presents then a "cleaned" view of the aggregate, where all depending target candidates are also hidden. The procedure *CLEAN* can be employed for this task.

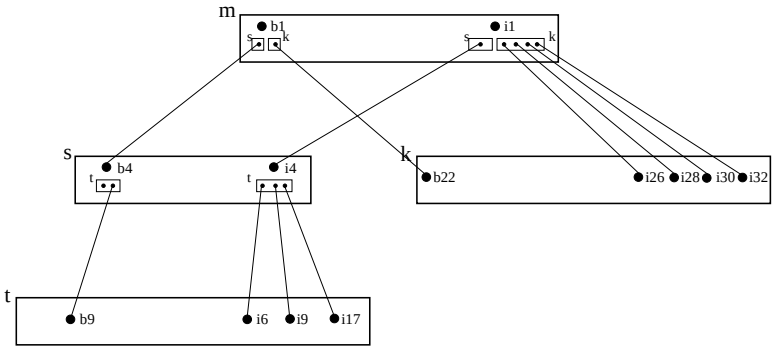
Example 6.5 Consider the complete answer aggregate in Example 6.3. The user selects target candidate *Pit and the Pendulum* and hides it. The resulting aggregate is the following:



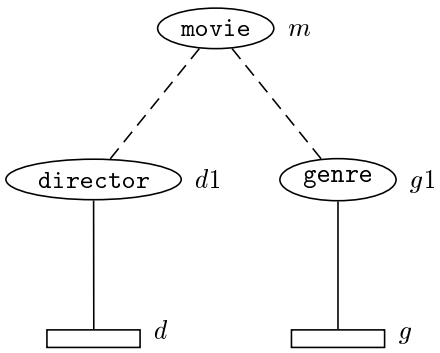
Restriction

We can also hide target candidates on a more abstract level, i.e. hide target target candidates that do or do not meet certain requirements. The requirements are formalized with the help of a comparator mapping target candidates or database nodes to Boolean values. Target candidates that are mapped to false are removed from the actual presentation of aggregate.

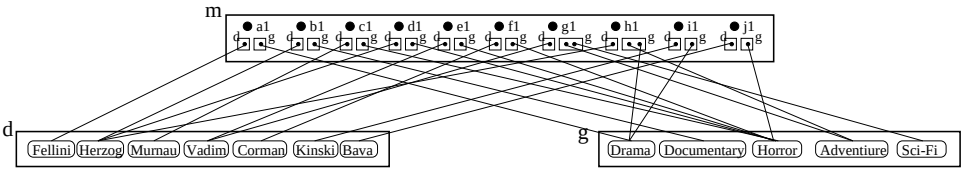
Example 6.6 Consider the complete answer aggregate in Example 5.26. We may restrict the view to movies produced in 1980 or later. This is a global restriction, since the production date is not directly available in the complete answer aggregate. The underlying comparator is almost the same as in Example 6.4 with the addition that the numeric value of the year is compared with 1980 in order to yield a Boolean result.



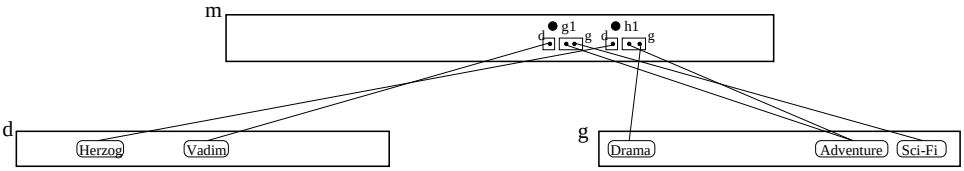
Example 6.7 Consider a query retrieving general information about directors and genres of movies:



The complete answer aggregate, where slots $d1$ and $g1$ are hidden and the genres and directors are grouped according to their textual content (see the next paragraph for a definition of this) is



(We will later see operations how the director-genre relation can be made even more explicit.) We can restrict this aggregate to target candidates in the movie slot with at least two genre-children (in the actual aggregate). The underlying comparator maps every target candidate to the number of its children (in the aggregate) and compares this value with two. This restriction is local since its requirement is restricted to the actual aggregate and results in the aggregate



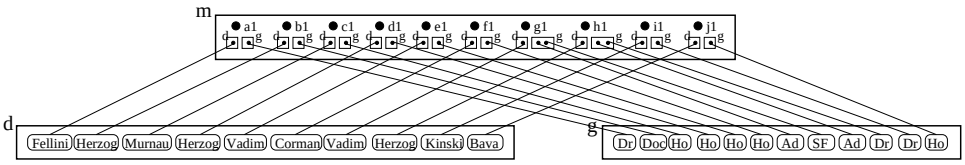
Equivalence Classes (Grouping)

We will now introduce a very powerful operation that allows to group target candidates. Sometimes we want to identify distinct target candidates based on some of their properties (formalized by comparators). Target candidates that are grouped into one class are displayed as one node and inherit all structural relationships from the set of representants they incorporate. This can be done for two purposes.

- To enhance the overview: We might for example collapse different text nodes with the same textual content. This method is elaborated in Example 6.8.
- To make knowledge explicit: The user might for example not care for the details of the single movies, but is interested in a more general classification of the movies by their directors. An example is found in Example 6.9.

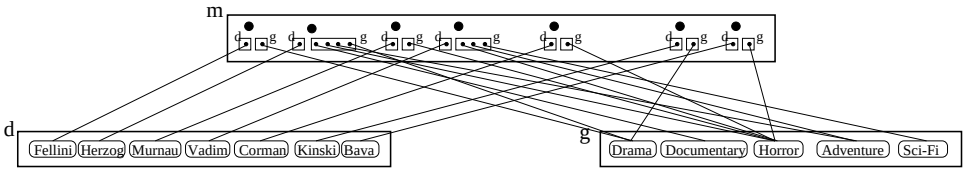
We illustrate the technique with two examples before explaining it more formally.

Example 6.8 In the query in Example 6.7 the user asked for general information about directors and genres of movies. The complete answer aggregate for this query contains target candidates in slots d and g with equal textual content:



We get a better overview if we present all nodes with the same textual content collapsed into one node, as illustrated in Example 6.7.

Example 6.9 Beginning with the aggregate in the Example 6.7, the user might want to group movies according to their directors. The resulting aggregate is depicted in the following figure. One node in the slot m now corresponds to a set of movies all directed by the same director. Note that the target candidates of the slots d and g have already been collapsed. This means that we now have a collapse operation on every slot: Slots d and g are collapsed according to the textual content of the target candidates, whereas target candidates in slot m are grouped according to their (collapsed) children in slot d. The underlying comparator maps target candidates to the textual content of their director child.



The technique of grouping is applicable to many properties of the target candidates including, for example, attributes. Imagine that every movie element in the document collection has a country attribute stating in which country the movie was produced. We can then, inside our defined sphere of interest, investigate features of a countries film production, e.g. genres, actors, keywords, etc., instead of examining features of the single films.

We will now define the grouping (with its special case collapsing) operation more formally:

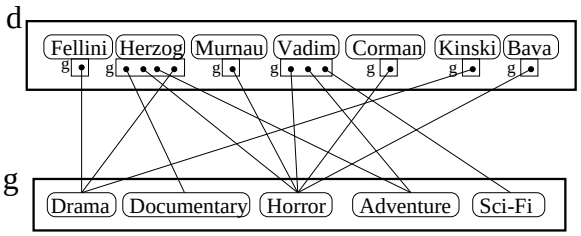
Definition 6.10 Let f be a comparator. We say that two nodes u and v are *f-equivalent* ($u \equiv v$) iff $f(u) = f(v)$. We denote with $[u]_f$ (or $[u]$ if f is fixed) the set of all nodes f-equivalent to u. An aggregate Agg' derives from an aggregate Agg by *grouping on f in slot x* in the following way: The content of all slots apart from slot x remains unchanged. If slot x in Agg contains target candidates $d_1, \ldots, d_n$ it does now contain the set $\{[d_1], \ldots, [d_n]\}$ as target candidates in no special order. These target candidates are linked to target candidates in children slots of x in the following way: $[d_i]$ has a link to target candidate e in a child slot y of x iff exists d_j ($1 \leq j \leq n$) that has a link to e in y.
If the function f maps every node to its textual content (for structural nodes the textual content is inherited from their descendants) we say that Agg' derives from Agg by *collapsing x* iff Agg' derives from Agg by grouping on f in x.

Remark 6.11 One important observation about collapsing target candidates is that we can simulate databases with a DAG structure as for example employed in databases for semistructured data (which can even have a cyclic structure, [ABS99, Suc98, Abi97, Bun97]). Since our model is limited to trees, we have sometimes multiple database objects (nodes) referring to only one object in the real world. With collapsing target candidates this is made invisible to the user as illustrated in Examples 6.8 and 6.7. In addition, we have the advantage that we only have to collapse the target candidates in the sphere of interest, what can be performed efficiently due to the relative small size of a realistic complete answer aggregate compared to the full relational document structure. In comparison, a system based on semistructured data, that uses a collection of structured documents to fill a database, has to make this object identity explicit by collapsing (if the object identity is not made explicit in the collection itself by ID/IDREF pointers). It is not sure whether this operation can be done in acceptable time over all nodes of the document collection since we have to compare pairwisely the textual contents of all nodes in the document collection.

Slot Replacement

Under some circumstances we may even rearrange the outer structure of the aggregate (and hide the root slot, too) in order to make dependencies more explicit. This is done by replacing one slot by another. We illustrate this technique with an example before explaining it in more detail.

Example 6.12 Consider the aggregate from Example 6.7. We would like to present the relation between directors and genres more explicitly. This can be done by replacing the slot m with the slot d, yielding the aggregate

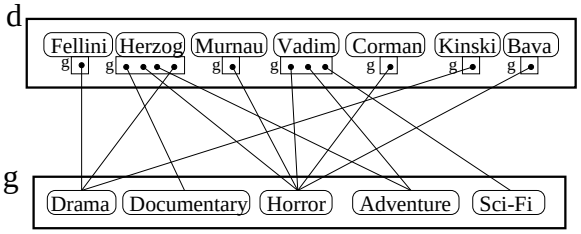


This aggregate obviously does not retain the original hierarchic structure of the query and the target candidates.

The operation behind slot replacement is the following: We can chose to replace a parent slot x with children $y_1, \ldots, y_k$ by one of its child slots y_i with children $z_1, \ldots, z_m$. Slot x is removed and replaced by slot y_i with all target candidates in slot y_k. Every target candidate in slot y_i now has pointers to target candidates in slots $z_1, \ldots, z_m$ and $y_1, \ldots, y_{i-1}, y_{i+1}, \ldots, y_k$. The pointers to slots $z_1, \ldots, z_m$ are the same as in the original aggregate. A target candidate d in slot x_i has a pointer to target candidate e in slot y_j ($j \neq i$) iff there existed a target candidate in the original slot x with a pointer to d and to e.

We now can give a second example for sorting of target candidates:

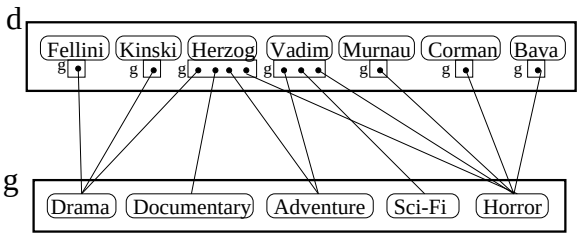
Example 6.13 Imagine that we want the target candidates in slot d of the above aggregate to be ordered according to the number of genres they dominate, i.e. we want order directors according to their variety in genres. This is a local ordering yielding the aggregate



Layout Optimization

If the user does not care for the order of the target candidates inside the slot or of the pointers, he or she can choose a form of presentation that tries to order these elements in a way that makes it easier for a human to understand the relations. The arrangement of target candidates is computed automatically, e.g. by trying to minimize crossing links. Alternatively, the user could arrange the target candidates by hand.

Example 6.14 The aggregate of Example 6.12 in a optimized layout is depicted in the following illustration:



6.2.5 Related Work

This section compares our basic query language augmented with the operations defined in this section with the relational algebra [Ull89] and a new work on querying semistructured data with a similar flavour.

Exploration Techniques and the Relational Algebra

As mentioned in the beginning of the last section, most exploration techniques introduced here on a visual level for complete answer aggregates have a counterpart in the relational algebra or in SQL-like languages. This section will elaborate on this correspondence further, although it is for the most operations obvious.

SQL covers the five basic operations of the relational algebra, projection, Cartesian product, union, difference and selection, and in addition so-called aggregation operators, `avg`, `count`, `sum`, `min`, `max` and `group`, that go beyond the expressivity of the relational algebra. We will first discuss the operations of the relational algebra. Hiding of variables is a standard technique in DBS generally known under the term projection. The technique named here restriction can be found in the relational algebra named selection. Slot replacement is a special case of the join which itself is a special of the Cartesian product (and selection). Another special case of the join occurs in query formulation where we can specify conditions about two or more subtrees of a query node. Our framework is certainly not relationally complete since we do not cover the full Cartesian product nor union or difference. Possibilities for integrating the latter two operations in at least a limited way are discussed in Chapter7.

We can recognize the SQL-aggregation operators `count`, and `group`, respectively, in the less detailed view on aggregates that shows only the number of target candidates inside a slot, and in the equivalence relations that can be defined (Examples 6.8 and 6.9). The other four aggregation operators can be incorporated into our framework in the same style. The SQL function `sort` has its equivalent in our model in the sorting of target candidates in the slots (Examples 6.4 and 6.13),

The above remarks show that the basic techniques underlying the exploration techniques are well-known in the field of DBS. The novelty comes from the way they are visually applied to the new data structure complete answer aggregates.

Goldman's and Widom's Interactive Retrieval Model

In [GW98] Goldman and Widom presented a retrieval model for semistructured data with a strong similarity to the retrieval model proposed in this chapter. Their model proposes an iterative two-step retrieval process as well, with the first step being a simple query to begin a search and the second step being an iterative exploration process into the results returned by the first step. The second step can be performed on a fully visual level using a data structure called DataGuides ([GW97]) that describe the structural regularities of semistructured data (in this case a view onto the complete database defined by the query in the first step and the later exploration steps) as a tree that can be expanded on demand. The model is based on and employs the Lore database system ([MAG+97]), a database system for semistructured data.

Goldman's and Widom's model is simpler than the framework proposed here, since it is directed towards inexperienced, casual Web users trying to find a piece of information in a Web site unknown to them. The initial query contains no structural information but consists of keywords only (though these keywords may match node labels as well). In [GW98] only one keyword is allowed in the initial query though the authors plan to extend the model to at least queries with multiple keywords. The browsing process of step two is more or less one-dimensional (in the

sense described in Section 1.3), since the user always focuses onto one result set. In this way, Goldman's and Widom's model is more a facility in supporting the user in finding a piece of information in an information space defined by a Web site rather than a tool for visualizing and exploring complex information spaces with a strong weight on making dependencies explicit what is the aim of the retrieval model proposed in this chapter.

Chapter 7

Open Questions and Loose Ends

In this chapter we will discuss open problems that are not treated in the rest of this work but can serve as a starting point for future work. We begin with considerations about including disjunctions and negations into the query language, discuss the incorporation of a notion of relevance into our model, and end with some remarks about an algebraic foundation of the visual operations defined in Chapter 6.

7.1 Enriching the Query Language

We began with defining the full first-order logic as a query language for relational document structures. In order to focus on the key aspects and to present an efficient evaluation algorithm, we restricted the query language to tree queries, that contained no logical connectives apart from conjunction and existential quantification. Chapter 6 extended this restricted query language and introduced a family of graphical operations on complete answer aggregates that can be seen as an extension of the query language.[1] We saw that the resulting query formalism provides enough expressivity to formulate powerful and sophisticated queries on relational document structures. Nonetheless, we should think about incorporating more of the original, permissive logical query language in order to improve on expressivity. We will informally discuss two logical connectives and see that they conform with the concept of complete answer aggregates and the basic requirement of efficient evaluation.

7.1.1 Negation

A natural desire in query formulation is the exclusion of certain nodes. This is expressed with the logical negation $\neg$. In Chapter 6 we already showed how to remove target candidates using a comparator function f. We did not specify exactly what properties of a target candidate or database node a comparator may take into account in order to compute the value of a node. This section discusses a possibility for a more formal implementation of negation in our framework. We will restrict the discussion to a simple class of formulae containing negation, where an integration into our framework is obvious.

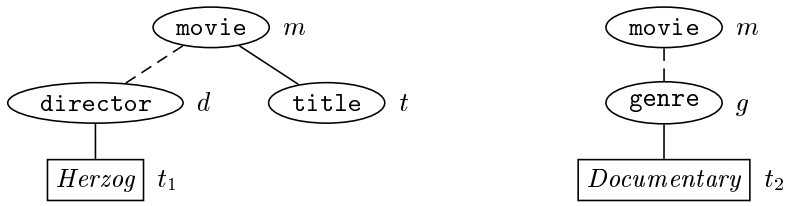
[1] In fact, all operations defined in Chapter 6 can be formalized as part of an extended query language.

Definition 7.1 Let $Q_1 = (\phi_1, \vec{x}_1)$ and $Q_2 = (\phi_2, \vec{x}_2)$ be local tree queries so that $\vec{x}_1$ and $\vec{x}_2$ have exactly one variable y in common. Then the query $Q = (\phi_1 \wedge \neg\phi_2, \vec{x}_1)$ is called an *exclusion tree query*. Q_1 is called the *constituting part* of Q and Q_2 the *excluding part* of Q.

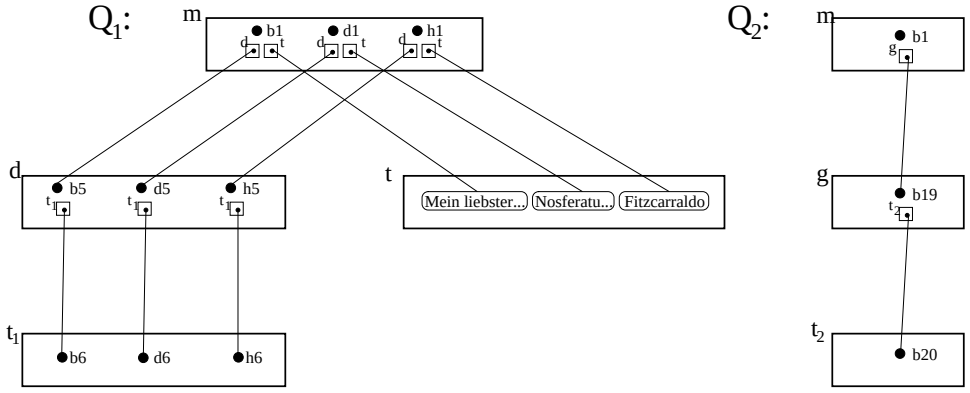
With exclusion tree queries we can exclude those answers from the answer set to Q_1 that map the common variable y to a database node so that there is an answer to Q_2 that maps y to the same database node. The computation of a complete answer aggregate for exclusion tree queries is quite obvious: We first compute answer aggregates Agg_1 and Agg_2 for Q_1 and Q_2 and then remove all target candidates in slot y of Agg_1 that do also occur in slot y of Agg_2. As a conclusion we have to clean (with the procedure `CLEANAGG`) the aggregate to remove newly isolated target candidates yielding finally the complete answer aggregate for the exclusion query.

Instead of making this point more formal we present an example as an illustration:

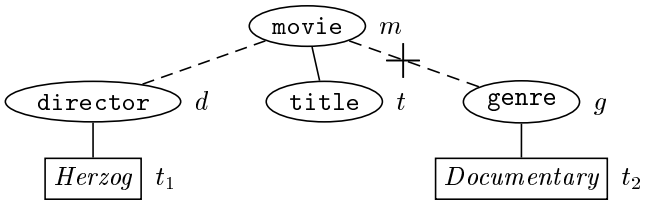
Example 7.2 Imagine that we are interested in titles of movies by *Werner Herzog* that are no documentaries. We can formulate the queries Q_1 and Q_2 as constituting and excluding part:



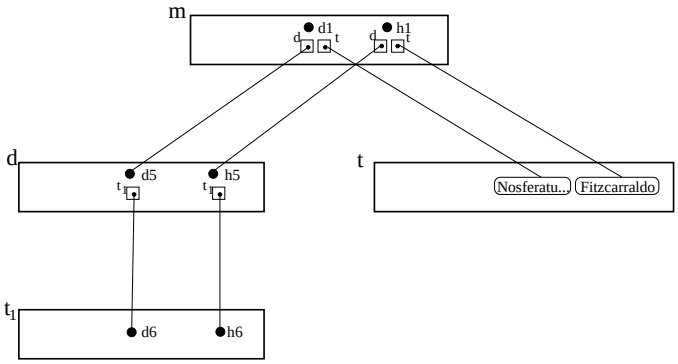
These queries result in the following two answer aggregates:



We can combine the query to the combined query Q (the negation is represented by a crossed edge)



The resulting complete answer aggregate for Q and the relational document structure in Appendix B.2 is



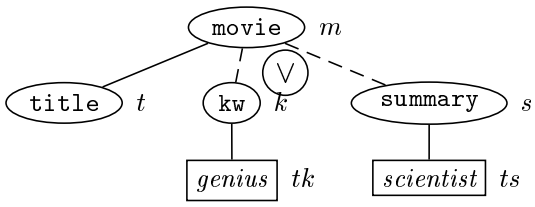
7.1.2 Disjunction

Another point of interest concerns the formulation of disjunctions. In order to implement the concept of disjunctions in our framework we have to extend the notion of aggregates in order to keep pace with the new requirements. We will not delve into details here but will just motivate this with a simple example:

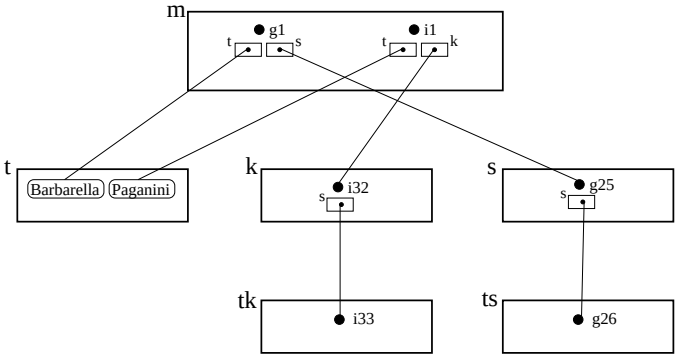
Example 7.3 The user might be interested in titles of movies whose description contains the keyword *genius* or a summary with the term *scientist*. This can be expressed in the following query:

$$\begin{aligned} Q \quad = \quad & (m \lhd t \wedge \texttt{movie}(m) \wedge \texttt{title}(t) \wedge \\ & ((m \lhd^{+} k \wedge \texttt{kw}(k) \wedge k \lhd tk \wedge \mathit{genius} \propto tk) \\ & \vee (m \lhd^{+} s \wedge \texttt{summary}(s) \wedge s \lhd ts \wedge \mathit{scientist} \propto ts)), \\ & \langle m, t, k, tk, s, ts \rangle \end{aligned}$$

This query can be visualized with the following tree, where the disjunction is represented with the symbol $\vee$ between the two respective edges:



The complete answer aggregate for Q and the relational document structure in Appendix B.2 is



We can see that target candidate `g1` only has pointer arrays for slots t and s, whereas target candidate `i1` has pointer arrays for slots t and k. A target candidate may even have pointer arrays for both slot k and slot s.

A simple tree query with disjunction may contain the logical connective $\vee$ at any (syntactical correct) place in the query. For the computation of a complete answer aggregate we split the query into several disjuncts, where each disjunct is a tree query. The disjuncts may contain shared variables. We then compute the complete answer aggregates for the disjuncts. The overall result is a complete answer aggregate that derives from the single aggregates by combination, i.e. combining the content of shared slots by forming a union over the sets of contained target candidates. Pointer arrays are as well combined via union, as are the vertical pointers. The notion of isolated target candidate changes with this new definition.

7.2 A Notion of Relevance

The central notion in Information Retrieval is captured by the notion of relevance. At the same time relevance is the major burden Information Retrieval has to deal with. The discussion on how to define relevance started with the very birth of Information Retrieval as a field in Computer Science and Computational Linguistics and is still lasting. In simple words, relevance describes the degree of how apt a document is in helping the user to satisfy his or her information need. Obviously, there is no sharp boundary between relevant and irrelevant documents. This led to the development that imprecision became one of the central constituents of Information Retrieval. With the help of imprecise relevance values the documents can be presented to the user according to their computed relevance. Only with the use of imprecise relevance values it is possible to retrieve documents in big collections, since normally every search term occurs too often in the collection to enumerate all matching documents. If we have imprecise values we can present the occurrences ordered according to their relevance values and the user is free to examine the first n hits. A field where the number of hits is in most cases too big is for example the World Wide Web. Search engines like Altavista frequently return thousands of hits for a single query.

Since the retrieval of structured documents is similar to Information Retrieval in general we should provide means to include imprecision in our formal model. A first decision has to be made in the choice of the model for imprecision: IR provides a lot of methods to treat imprecision formally, e.g. the Vectorspace model ([SM83]), probabilistic models ([vR79, Fuh92]), extended Boolean models ([SFW83, Rou90, FBY92a, Lee94]), etc. In a very simplified view the models have the following common basic features: A term in a document has a weight with respect to this document, describing the measure of how good the term describes the content of the document. A query contains a set of (weighted) search terms. For every document and every search term the weight of the term is computed, e.g. by accessing an index file. The models mainly differ in the question of how to combine the term weights to a measure of how good a query describes a document (i.e. how good the document is as an answer to the query).

Applying one of these models for imprecision suggests to store for every occurrence its weight inside the smallest surrounding element. We can then compute a ranked aggregate that stores the respective weights for the target candidates in the leaves. These weights have to be combined and traded upwards through the aggregate. If a weight is too small we can neglect it, since it will appear in the result list at positions not relevant. This procedure yields as a result a ranked aggregate where every target candidate d in a slot x owns a weight describing its qualification to be an image of x under an answer mapping.

Although the above strategy may seem obvious, it needs thorough investigation in detail, since the goal is to have a model that maps the user's concept of relevance onto structured documents and complete answer aggregates. First steps towards

the marriage of structured documents (not treating complete answer aggregates) and relevance ranking have been made in [FGR98], [BYNVdlF98] and [SJJ98].

7.3 Miscallenous

Regular Path Expressions

Most query languages for XML and semistructured data allow for regular expressions over paths (cf. Section 8). The user can specify in the query that a node is reachable, say, via an arbitrary long sequence of section or subsection edges using the path expression `(section | subsection)*` as a label for the corresponding query edge. The differentiation between soft and rigid edges in the logical Tree Matching framework is a first step in this direction. An integration of full path expressions will extend the expressive power of logical Tree Matching. An advantage is that the representation techniques with complete answer aggregates are fully compatible with path expressions on edges. The difficult point will be an efficient implementation using appropriate index structures for path expression as presented e.g. in [MS99b].

A Visual Algebra

In Chapter 6 we described the exploration techniques available for complete answer aggregates on a rather informal level. Future work should aim at developing an aggregate algebra that describes the exploration techniques on an algebraic level. With the help of such an algebra more comprehensive propositions about the expressiveness of logical Tree Matching can be made.

Query Reformulation and Aggregate Manipulation

The retrieval model proposed in Chapter 6 is not fully iterative in the following sense: If we have changed a view upon an aggregate and then reformulate a query, a new complete answer aggregate is computed and presented. The original changes in the view upon the aggregate are lost. A fully iterative two-step retrieval process should try to propagate the view onto an aggregate upon a newly computed aggregate as far as possible, if requested by the user. This goes hand in hand with an optimization that does not compute a new complete answer aggregate from scratch if a query is reformulated but tries to use as much as possible of the already computed aggregate. The fact that the algorithm decomposes the query into paths and treats the query pathwise suggests that this optimization can be implemented without major changes to the overall structure.

Construction and Manipulation

In Chapter 8 we will see that many XML query languages also have a construction part that specifies how new XML documents can be generated out of the query result (or as the query result). The rich structure of complete answer aggregates can be exploited in the same way. A first step in this direction is the notion of aggregate structures as defined in Section 4.6. But the user needs more control over the target candidates out of which new XML documents are constructed and on the way how this happens. This can be done with a construction language that works on aggregates (or views upon aggregates) and builds new XML documents according to the specifications formulated with the construction language. Possibly this construction language can be designed as a visual language as well.

A point closely related to this, but involving far more work, is the manipulation of the relational document structure, the document database. This can also be done on the basis of aggregates or views upon aggregates. An appropriate manipulation language allows to formulate how the information in the complete answer aggregate shall be used to replace parts of the relational documents structure or to add new parts. But this effort has consequences far beyond the question of manipulation language design. All problems that have been treated in the field of Database Systems, like updates of index structures, recovery, transaction management, etc., have to be taken into account.

Chapter 8

Related Work

Due to the position of structured document retrieval in the overlap between Database System (DBS) and Information Retrieval (IR) there are numerous models with a similar functionality. In addition there are lots of dedicated formalisms for the retrieval of structured documents, a development that was considerably intensified by the overwhelming success of XML as a data representation and exchange language. In the first two sections we will discuss the relation between logical Tree Matching as introduced in this work and DBS and IR, respectively. We will then review some systems and formalisms for the retrieval of structured documents and compare them with respect to expressivity and efficiency with logical Tree Matching. For DBS and the structured document retrieval formalisms being based on tuples we will in addition discuss possibilities and benefits of employing the concept of complete answer aggregates.

8.1 Database Systems

Ullman gives the following functional definition of a DBS ([Ull89]): (1) A DBS manages persistent data and (2) provides efficient access to large amounts of data. In structured document retrieval we are more concerned with information (semantic entities) than with data (syntactic entities). But the relation is close enough to discuss similarities and differences. In the sequel we will discuss the most prominent three classes of database systems with respect to logical Tree Matching, distinguished by the logical model underlying the data: Relational, object-oriented and graph database systems, where the latter includes database systems for semistructured data.

Over the last twenty years we could observe in the field of DBS a tendency of moving away from general data models being universally applicable in any domain to highly specialized data models dedicated to only one domain. The motivation for this development is the need for specialized algorithms exploiting the peculiarities of data in a given domain, the need of providing a query and manipulation language that is tailored for the specific domain, and the failure of the main model in DBS, the relational model, to model special domains appropriately. As a result, the world of DBS contains now dedicated models for geographic data, for biological data, for multimedia data, for textual data, etc. The advantages of using specialized data models are: (1) specialized index structures, (2) application-specific functionality built into the database system instead of outside of it, thus gaining efficiency and standardization, and (3) an adequate user-interface tailored for the respective domain. If we look for example at geographic databases we can observe these advantages on (1) special index structures for spatial objects, e.g. R-Trees,

(2) a geo-relational algebra defining domain-specific operations like the spatial join, and (3) a spatial database system with an appropriate graphical user interface. Following this development we may take the perspective of logical Tree Matching being a specialized data and query evaluation model for the domain of structured documents. Nonetheless, there is the major difference that DBS allow for the manipulation of data, whereas this is not treated in logical Tree Matching at all.

Relational DBS

Relational DBS are now the most widespread class of DBS. Their success is due to their efficiency and robustness, which is based on the underlying relational algebra. Relational DBS model data on the base of relations and tuples. This allows for efficient index structures supporting evaluation and manipulation, and it is a general model that can describe many different types of data. Nonetheless, as pointed out before, the relational model is not always appropriate.

The query languages for relational DBS have their foundation in the relational algebra, extended by aggregation constructs like `count` or `group`. We already discussed the expressivity of logical Tree Matching compared to the relational algebra in Section 6.2.5.

One could think about storing structured documents in relational DBS[1]. We encounter two problems with this approach: The inability of relational DBS to deal with text in an appropriate way and the excessive use of joins in query evaluation on tree-structured data. The former point is a well-known deficiency of relational DBS. Textual data differs from relational data in its varying length, in its very own syntactical structure and in the fact that text bears a semantical content that requires special techniques to be excavated. The latter point refers to the fact that trees have to be modeled in the relational model as sets of nodes where each node points to its children and parent (e.g. [WCB+00]). Due to the fact that relational DBS employ tuples, which have a fixed length, we have many occurrences of the same node in a tree table, each occurrence pointing to one of its children. [STZ+99] analyzed the storage of XML documents in relational databases and came to the conclusion that a query specifying one or more path patterns has to be translated into a relational query with many natural joins. It is well-known that joins are the operations that perform least efficient in relational DBS. Of course, the evaluation in logical Tree Matching requires operations similar to the join, too. But here we take use of the index structures that are tailored to the application domain of structured documents and make query evaluation efficient.

Due to answers based on tuples, the problem of the combinatorial explosion occurs naturally for relational DBS as well, whether we model structured documents or other types of data. Naturally a similar concept to complete answer aggregates could be useful. For the domain of graph databases this is discussed in more detail.

Object-Oriented DBS

Object-oriented DBS use the object paradigm to store data. This paradigm includes, among other concepts, data abstraction, object identity, and inheritance. The object model provides more flexibility and expressiveness than the relational model to the detriment of efficiency.

A suggesting idea is to use the hierarchic class system modeling the inheritance in object-oriented DBS to model the tree structure of structured documents ([AQM+97]). But apart from the fact that here the details of modeling may cause

[1] In fact, our implementation uses an index structure that is planned to be implemented on top of a relational DBS. What we mean here is a direct encoding of relational document structures into relations.

troubles, the major disadvantage of this approach is the lack of efficiency. We hand over the control to the object-oriented DBS getting ourselves rid of the possibility to evaluate queries in an optimized way tailored to the application domain of structured documents. And we can not expect the object-oriented DBS to provide full functionality for accessing textual data, since it is still a formalism universally applicable.

Since answers are based on tuples, we naturally encounter in object-oriented DBS the combinatorial explosion in the number of answers, what could make the use of complete answer aggregates attractive in this field as well.

Graph DBS and Semistructured Data

Graph DBS are dedicated to store graph structured data, that occur frequently in many application domains as hypertext data, geographic data, multimedia data, etc. The formalisms for graph DBS are numerous, e.g. [GPdBG94, CM90, AS92, Fra96], most of which implement the notion of answers as a mapping from a pattern graph to a target graph. Recently the research on graph DBS has gained particular interest with the development of the semistructured data model ([Suc98, Abi97, Bun97]) and its close relation to XML data ([ABS99]). We will focus the discussion on semistructured data, but most points are also applicable to other graph DBS based on answer mappings.

The model of semistructured data was given birth due to the observation that traditional DBS can not treat data with irregular, partially unknown or changing structure. This observation can be made frequently when exchanging data between heterogenous databases. The problems arising from differing or unknown schemata are legion. The basic idea of semistructured data was to use a graph model to describe the data, thus allowing for arbitrary irregularities and having data instances describing their own schema by means of the graph structure with labels. With these capabilities the semistructured data model can be used as a model for data interchange between databases as well as a data model for databases on its own, e.g. the Lore database system ([MAG$^+$97]). In addition, the semistructured data model revealed itself as an abstract description of XML data (when neglecting the order among elements), thus attracting major interest in the database and World Wide Web communities.

Query languages for semistructured data ([ABS99, AQM$^+$97, BDHS96]) are based upon answer mappings. They are more expressive than logical Tree Matching, including for example regular expressions for paths or edges labeled with formulae. And, of course, the underlying graph model is more expressive than the tree model employed in logical Tree Matching. Not included in the semistructured data model is a notion of ordering between the elements nor a dedicated treatment of textual data with its peculiarities. The absence of order and special treatment of text can make the use of the semistructured data model problematic for structural documents. For query evaluation, [MS00] showed the decision problem whether there exists a solution for a query describing a DAG (directed acyclic graph) to be NP-complete if the database has at least a tree structure. This means that the advantage in expressivity has to be paid with efficiency.

We can expect the effects of the combinatorial explosion in the number of answers to be even worse if we move from tree models to graph models. Therefore graph databases seem to be an interesting field for the application of techniques based on complete answer aggregates. [MS00] showed how complete answer aggregates can be used for graph DBS and elaborate on the benefits of having a visual interface for the examination and reformulation of complete answer aggregates as query results. In the more general graph model we loose the locality property of answer sets (as formulated in Lemma 3.27) resulting in the fact that we can no longer generate all

instantiations out of a complete answer aggregate without searching: The reason is that we can not find an order for the query nodes so that the instantiation of a node depends only on values of preceding nodes. For tree queries we exploited the fact that such an order exists. A deeper investigation on this subject has been made in the field of constraint networks in [Fre82].

8.2 Information Retrieval Systems

Baeza-Yates and Ribeiro-Neto define Information Retrieval as the discipline dealing "with the representation, storage, organization of, and access to information items" ([BYRN99b]). Traditionally, the information is assumed to be contained in a collection of text documents that are accessed by the user with a query in the form of a set keywords. We will refer to this traditional concept of Information retrieval in this section and will not touch more modern aspects like image retrieval, multimedia information retrieval, etc. Due to the fact that "information" is a notion with a less precise semantics than "data" (describing an uninterpreted syntactical object), the whole field of Information Retrieval is concerned with the concepts of imprecision and relevance of a document with respect to a given query. Information Retrieval is a field that is on the technical level very close to Computer Science in general and DBS in particular, on another level it is located in Computational Linguistics due to the fact that information in text documents is represented in natural language.

In Sections 6.1 and 7.2 we describe the retrieval process in IR and the central role of the concepts "relevance" and "imprecision" in more detail. We will now compare the logical Tree Matching formalism with the general model of Information Retrieval. The main difference is that logical Tree Matching has no notion of relevance as present in IR. Answers in logical Tree Matching all have the same relevance with respect to the query, namely "full" relevance. In this aspect logical Tree Matching follows a DBS-oriented approach. So far there are only few formalisms modeling the retrieval of structured documents with taking into account relevancy. Examples for formalisms like this can be found in [FGR98, BYNVdlF98, SJJ98]. The reflection of accumulated occurrences of keywords is not fully satisfying in these formalisms. One would expect that a keyword occurring in a paragraph more than once describes the paragraph better than a keyword occurring only once. On the other hand, consider a keyword occurring in a paragraph with weight w. If the paragraph was the only content of a surrounding section, we can reasonably say that the keyword occurs with weight w in the surrounding section. In the other cases, the weight of the keyword in the section should depend on the other paragraphs in this section, too. The mentioned formalisms do not model these intuitive principles.

In IR no structure or only a very simple structure is assumed for the text. This makes it impossible to implement the retrieval of structured documents with the standard IR tools. Related to this is the principle of always returning a set of full documents as result to a query. Newer developments ([SAB93]) allow to discard this monolithic principle and retrieve passages of documents. As can be inferred from the discussion so far, IR systems are one-dimensional formalisms (in the sense discussed in Section 1.3) since they return unstructured sets of answers as opposed to tuples. This makes it impossible to apply complete answer aggregate techniques to standard IR models. But as discussed in Section 7.2 the incorporation of relevance seems to be a very promising concept for logical Tree Matching and complete answer aggregates.

8.3 Structured Document Retrieval Systems

The retrieval of structured documents gained its first publicity with an article by Gonnet and Tompa on their p-strings model for the retrieval of structured documents ([GT87]). Since then there was a moderately growing interest on various models for the retrieval of structured documents until XML entered the stage and attracted enormous attention among researchers towards structured documents and their retrieval. The first workshop on query languages for XML ([W3C98b]) organized by the W3C (World Wide Web Consortium) presented more than 60 position papers from academic and industrial researchers. The situation in the moment is that there is a Babylonian variety of query languages for XML. We will pick out some of them that are well-known in the community and discuss them together with the other retrieval formalisms developed for structured documents in general. Surveys on research on structured document retrieval can be found in [BC00], [DFF+99], [FSW99], [FLM98], [BYN96] and [Loe94], surveys covering software products implementing query capabilities for XML and SGML in [Bou99] and [KN98a], and a work on algebraic foundation of one-dimensional formalisms in [CM98].

We distinguish between one-dimensional formalisms and multi-dimensional formalisms (as described in Section 1.3). One-dimensional formalisms return unstructured sets of document nodes as a result of a query, whereas multi-dimensional formalisms return tuple-like structures. One-dimensional do not have to struggle with the combinatorial explosion in the number of answers and have a strong advantage in efficiency of query evaluation, whereas they are less expressive. Consens and Milo show in [CM98] that formalisms based on the region algebra, what is the case for most one-dimensional formalisms, can not express the child relation and the both-included relation (e.g. "*Give me all chapters that include both a title reading 'Title' and a section containing the keyword 'section'*") in periodic document structures. In addition, one-dimensional formalisms can not express the full join (since they do not involve the notion of tuples) and can not return structured answers, for example pairs of title-author nodes.

Most of the XML query languages have a selection and a construction part. The former corresponds to the concept of queries employed in this work whereas the latter provides a mechanism for document manipulation and restructuring. In the comparison we will neglect the construction part since there is no equivalent in logical Tree Matching. The formalisms also differ in the point whether they present a query language only or provide a model for database system involving questions of persistent storage and efficient index structures with a corresponding query language.

8.3.1 One-dimensional Formalisms

We will only superficially discuss the one-dimensional formalisms, since they are too different from logical Tree Matching. Their limited notion of answers as sets of nodes makes them less expressive but in general evaluable very efficiently. Due to the simple answer structure they are not subject to the combinatorial explosion in the number of answers and thus not amenable to the aggregate techniques developed in this work.

Overlapped Lists

This formalism consists in fact of a whole family of formalisms: "PAT expressions" ([ST94]), its successor "Overlapped Lists" ([CCB95]) and the latest successor "Nested Text Region Algebra" ([JK99]) with its implementation "sgrep" ([JK96]).

Am expressiveness study on the underlying text region algebra can be found in [CM98].

The formalism is based upon regions describing the structure of the text. These text regions correspond to nodes in the logical Tree Matching formalism, apart from the fact that they may overlap and that the childhood relation is not stored explicitly but can be inferred from the offset points that associate each node with a text region. The algebra is simple and effective, if joins or multi-dimensional answers are not needed, and can be evaluated efficiently in linear time in the size of the database. The evaluation is supported by efficient index structures for text regions.

Proximal Nodes

The basic model of Proximal Nodes is similar to Overlapped Lists, apart from the fact that the structure imposed on the text is a strict hierarchy, i.e. overlaps are not allowed, and that the tree structure of the documents is explicitly stored. This makes direct-childhood queries even in periodic document structures possible. The evaluation, using dedicated index structures, is based on the concept of comparing sets of nodes by traversing them in parallel. Since the query language only allows for operations that can be evaluated on "nearby" nodes (what is no strong restriction) this model evaluates queries in almost linear time in the size of the database ($\mathcal{O}(n \cdot \log(n)$, where n is the number of nodes in the database).

XSL and XQL

XQL ([RLS98, ABS99]) is an extension to a full query language of the W3C working draft for XSL ([W3C00a]), a stylesheet language for XML documents. XSL and XQL are functional languages that make it easy to specify recursive traversals through the structure of XML documents. The syntax of XQL is oriented on the XML syntax and XQL allows to construct new XML documents. XQL does neither provide an evaluation model nor addresses the question of efficient index structures, but has already been implemented in numerous industrial products including Software AG's database system Tamino and Microsoft's Internet Explorer 5.

Query Automata

Query automata ([NS00]) provide a rich query language based on a fragment of monadic second order predicate logic (MSO) that is efficiently evaluable on tree structures. Queries may include regular expressions over formulae that are evaluated on sequences of edges or nodes and may pertain to the subtree or context of a node. Queries can be evaluated in time linear in the size of the database and exponential in the size of the query. Aspects concerning persistent storage are not covered.

Dolores

Dolores ([FGR98]) is an IR system that can handle arbitrary document structures and multimedia documents. It is based on probabilistic logic and thus incorporates the notion of relevance ranking. Queries and documents are formulated and evaluated with an object-oriented and probabilistic extension of first-order predicate logic. Although queries incorporate the use of variables, Dolores is a one-dimensional formalism, since answers are sets of document nodes. The outstanding strength of Dolores is its capability to express imprecise knowledge. But, as mentioned in Section 8.2, the propagation of the relevance values to upper levels of the

documents does not depend on the context, what does not always meet the intuition a user has about "distributed" relevance in documents. A major drawback is its architecture that translates the complete structure of queries and documents to probabilistic Datalog and thus fails to exploit the special features of tree-structured entities in query evaluation. Dolores provides a layer-model for query evaluation, where the lowest level are efficient index structures based on an basic IR engine.

8.3.2 Multi-Dimensional Formalisms

This section compares some of the multi-dimensional formalisms with logical Tree Matching. A comprehensive comparison of logical Tree Matching with Kilpeläinen's Tree Matching formalism can be found in Section 3.4. In general all multi-dimensional formalisms suffer from the combinatoric explosion in the number of answers.

XML-QL

XML-QL ([DFF+99, DFF+98, ABS99]) is a relationally complete XML query language that allows to formulate queries in an XML style, i.e. queries involve the use of element tags, attributes etc. With XML-QL the user may also construct new XML documents and thus manipulate a document collection. Queries consist of a selection, a filter and a construction part and allow for path expressions, specification of order, grouping, node and tag variables. The selection part selects node tuples from the document collection, that are filtered according to the filter part. The construction part constructs new XML documents from the set of tuples. The topic of persistent storage and efficient index structure is not addressed.

If we restrict XML-QL to the selection and filter part, we can see that we can use complete answer aggregates in order to represent the result of a query. Possibly there exist mechanisms similar to the construction part of XML-QL that can transform complete answer aggregates into new XML documents. A first step in this direction are the aggregate structures defined in Section 4.6.

XML-GL

XML-GL ([CCD+99]) is a graphical query language to XML documents. Its expressive power is comparable to XML-QL, but XML-GL provides a fully graphical query and manipulation language. Queries are divided into a selection and a construction part, both specified in a graphical way, and may involve specifications of paths, joins, and grouping. The visual approach makes the query and construction language very appealing and intuitive. Persistent storage and efficient index structures are not covered by XML-GL. Since information is (internally) transported from the selection to the construction part by the means of tuples, XML-GL is possibly in the same way amenable to aggregate techniques as is XML-QL.

Chapter 9

Conclusion

This work presented a logical reformulation of the Tree Matching formalism that is used for the retrieval of structured documents. The new formulation, "logical Tree Matching", added a great deal of flexibility to the original formalism in allowing for hybrid queries that break the strict division of the distinct problem classes in the original Tree Matching formalism. In addition we introduced a generic constraint mechanism into the query formalism that can be adapted to the peculiarities of a document collection in a special domain, and used the left-to-right order as a typical example for constraints in the rest of this work.

A major problem of the original Tree Matching formalism, as well as of all other query formalisms providing complex answers, is the combinatorial explosion in the number of answers that is caused by permutational phenomena. This work introduced the data structure "complete answer aggregate" as an alternative to confronting the user with the list of all answers. Complete answer aggregates avoid the combinatorial explosion in the number of answers by using a shared representation. They provide an intuitive overview over the set of all answers in a graphical way so that their presentation to the user constitutes an added value compared to a simple list of all answers.

We have shown that complete answer aggregates can be computed efficiently in time of order $\mathcal{O}(q \cdot h \cdot n \cdot \log(n))$. For realistic classes of queries and document collections the algorithm runs even in time $\mathcal{O}(q \cdot h \cdot n)$. We elaborated on how the algorithm is supported by dedicated index structures that exploit the peculiarities of structured documents.

We also showed how the sophisticated iterative retrieval process modeled in Information Retrieval can be applied to structured document retrieval, relying heavily on complete answer aggregates and their manipulation. The graphic nature of complete answer aggregates makes them amenable to a family of visual exploration and manipulation techniques that have been presented and with the help of which the user can explore a complex information space and make hidden dependencies explicit.

Appendix A

Example Relations

Since the user can only query those relations on the document nodes that are defined in the relational document structure, it is important to fix the set of relations for an application. We will outline, in addition to the order relation defined in Section 3.2.2, some common relations that are typically for structured documents.

Attributes

Attributes are a central feature of structured document standards like SGML or XML as well as of semistructured data. There are three possibilities for modeling of attributes in our setting: (1) via the generic relation mechanism, (2) via integration of attributes in the definition of the notion of document structures, and (3) as children. We will outline how attributes can be modeled with relations. This helps us to keep the basic model simple. A consequence of this decision is that modeling of attributes seems a bit cumbersome. But this formal deficiency will not be visible for the user of a system based on logical Tree Matching if an appropriate user interface is used.

Attributes in SGML, XML and structured documents in general are attributes on nodes, whereas attributes in semistructured data ([Abi97, Bun97, Suc98]) are modeled as attributes on edges and nodes.[1] We will cover both approaches.

Let $\mathcal{A}$ be the a finite set of attribute names and $\mathcal{V}$ a set of attribute values. We introduce (possibly infinite many) unary and binary relation symbols $r^N_{\{a_1=v_1,\ldots,a_k=v_k\}}$ and $r^E_{\{a_1=v_1,\ldots,a_k=v_k\}}$ for every $k \leq |\mathcal{A}|$, all subsets $\{a_1,\ldots,a_k\}$ of $\mathcal{A}$ with $a_i \neq a_j$ for $i \neq j$ and all subsets $\{v_1,\ldots,v_k\}$ of $\mathcal{V}$. A node d has the values $v_1,\ldots,v_k$ for attributes $a_1,\ldots,a_k$ iff $d \in I(r^N_{\{a_1=v_1,\ldots,a_k=v_k\}})$. An edge from d to e has the values $v_1,\ldots,v_k$ for attributes $a_1,\ldots,a_k$ iff $(d,e) \in I(r^E_{\{a_1=v_1,\ldots,a_k=v_k\}})$.
With this method we can even model multiple labels on nodes and edges.

Typing Information

SGML and XML have no support for basic data types as for example dates, amounts of money, integer numbers, etc. Recent work in progress, like the XML Schema Working Group ([W3C99a, W3C99b]), tries to incorporate data types in the XML framework. Relational document structures provide a very simple model for basic data types: We shall introduce relation symbols r_T for every type T. $d \in I(r_T)$ iff d is of type T.

[1] Since we are restricted to tree-structured documents both approaches are equivalent.

Proximity

Many formalisms for structured documents (e.g. [NBY95, CCB95, ST94]) provide the concept of proximity based on a measure of distance of words in their query languages. A distance measure on word level is difficult to implement directly since the finest level of granularity in our model is the level of text leaves, but not words. Nonetheless we will outline how to implement a distance measure on node level.

A leaf node $d \in V$ in an ordered relational document structure is called the *left-to-right successor* of a leaf node $e \in V$ iff $d <^V_{lr} e$ and there exists no leaf node $e' \neq e$ with $d <^V_{lr} < e' <^V_{lr} e$. The notion *left-to-right predecessor* is defined symmetrically. The *leftmost (rightmost) reflexive descendant* of a node d is a leaf e so that e has no left-to-right predecessor (successor) d' that is a reflexive descendant of d. From this definition follows that the leftmost (rightmost) reflexive descendant of a leaf d is d itself.

We assume that there exist symmetric distance measures δ_l on neighbouring leaves and show how to lift them to distance measures δ on all nodes. Let the distance between a leaf node d and its left-to-right successor be $\delta_l(d, e) = \delta_l(e, d)$. In some cases it is useful to model $\delta_l(d, e) = \delta_l(e, d) = 1$ for all neighbouring leaves d and e. In other cases the distance $\delta_l(d, e)$ can be based on the number of words in d or e. Now let d and e be arbitrary nodes in a relational document structure. If d is a reflexive descendant of e or vice versa, then $\delta(d, e) = 0$. Else $\delta(d, e) = \delta_l(d', e')$ if $d <^V_{lr} e$ ($e <^V_{lr} d$, resp.), d' is the rightmost (leftmost) reflexive descendant of d and e' is the leftmost (rightmost) reflexive descendant of e.

In order to model this distance measure we introduce binary relation symbols δ_m for all distances m two arbitrary document nodes may have. Now we define $(d, e) \in I(\delta_m)$ iff $\delta(d, e) = m$.

Of course, the distance measure $\delta(d, e) = m$ can be lifted to comparisons $\delta(d, e) < m$, $\delta(d, e) \geq m$, etc.

Distance inside of given text nodes only, similar to distance in the Proximal Nodes formalism ([NBY95]) can be simulated with relations in a similar way as attributes on nodes. But it is doubtable that queries operating on this simulation can perform efficiently.

Inequality

As elaborated in Section 3.4 our formalism differs from Kilpeläinen's Tree Matching in the treatment of distinct query nodes (answers do not have to be injective in our formalism). It may be desirable for the user to express the constraint that two distinct query nodes are mapped to two distinct document nodes. We model this with the inequality relation $\neq$ on document nodes: $(d, e) \in I(\neq)$ for all distinct nodes $d, e \in V$.

Semantic Comparisons

It may be useful to model semantic comparisons on certain data types like integers, dates, amounts of money, etc. This is strongly domain specific and not elaborated further here.

Similarity

A central notion in he field of Information Retrieval is the notion of similarity. Various measures for similarity of documents have been proposed: The Vectorspace model ([SM83]), the probabilistic model ([vR79]), etc. We can adopt these models and define a finer notion of similarity on the level nodes instead of documents. Let $\sigma(d, e)$ be the similarity of two documents according to a given similarity model,

e.g. the vectorspace model. It is easy to lift this monolithic notion of similarity to arbitrary document nodes d, e by computing the similarity $\sigma(d', e')$ of the two document fragments represented by d' and e'. We then introduce binary relation symbols σ_s for all values s the similarity of two arbitrary nodes may be. Then $(d, e) \in I(\sigma_s)$ iff $\sigma(d, e) = s$.

Appendix B

"Movie Collection": An Example Database

This appendix contains the description of an example database that is used for most examples in the text. The idea is to have an example database similar to the Internet Movie Database (IMDB: [IMD00]). Our example database "Movie Collection" consists of a collection of XML documents following the DTD defined in the first section. In the second section we give a set of 10 example documents that are contained in "Movie Collection" represented as trees in a graphical form. In the third section, the same set of documents is presented as XML source code.

"Movie Collection" consists of a set of documents storing information about films. Naturally, we can only use a limited set of documents with only a small amount of data for each movie. Most data was taken from the Internet Movie Database[1]. The information for every movie includes title (in original language), year of production, director, actors and writing credits and some information about the content, like genre, summary, keywords and additional comments. The keywords and the actors are ordered according to their importance, i.e. the more significant a keyword or actor is for a movie, the earlier it appears.[2] The comments may contain opinions of other people about the movie and are, like most of the other data, taken from the Internet Movie Database.

Typical for structured documents (and semistructured data) is the fact that the collection has a somewhat irregular structure: Some documents have a summary element, whereas others do not.

B.1 A DTD for "Movie Collection"

```
<!ELEMENT MOVIE      (TITLE, YEAR?, STAFF, CONTENTS)         >
<!ELEMENT TITLE      (#PCDATA)                               >
<!ELEMENT YEAR       (#PCDATA)                               >
<!ELEMENT STAFF      (DIRECTOR, CAST, WRITER*)               >
<!ELEMENT CONTENTS   (GENRE+, SUMMARY?, KEYWORDS?, COMMENT?) >
<!ELEMENT DIRECTOR   (#PCDATA)                               >
<!ELEMENT CAST       (ACTOR*)                                >
<!ELEMENT WRITER     (#PCDATA)                               >
<!ELEMENT ACTOR      (#PCDATA)                               >
```

[1] Note, that in contrast to the Internet Movie Database, that has a full graph structure, our "Movie Collection" is only a set of trees

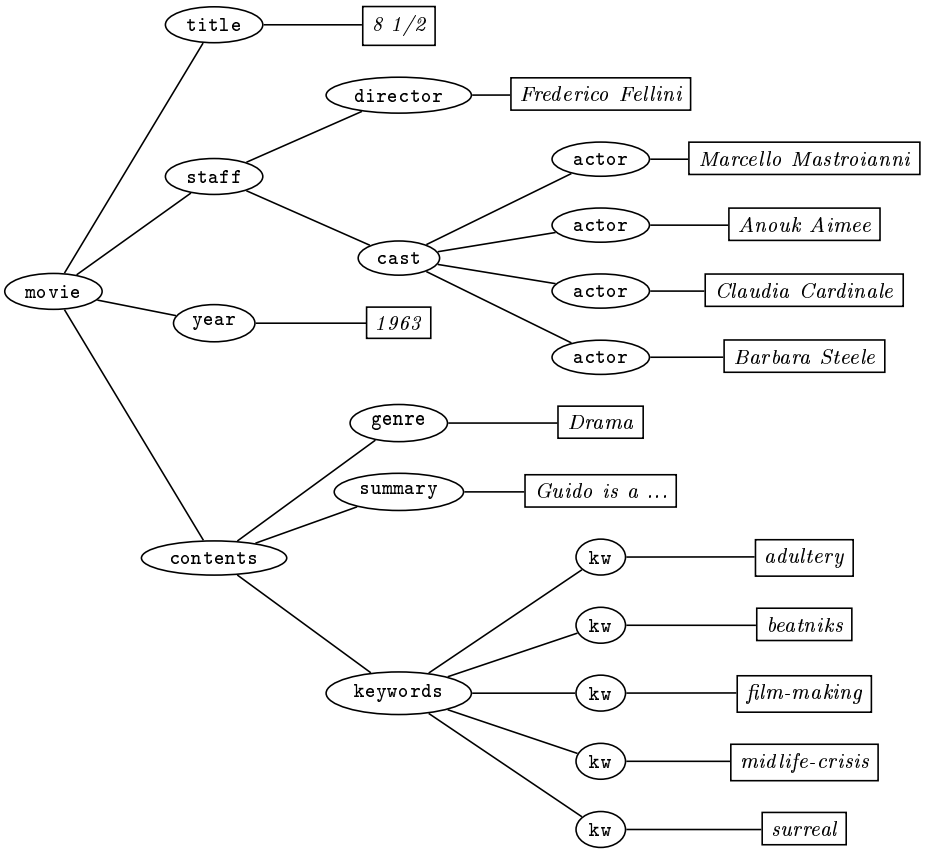
[2] This order is of course subjective, based on personal biases and some pieces of information from the Internet Movie Database.

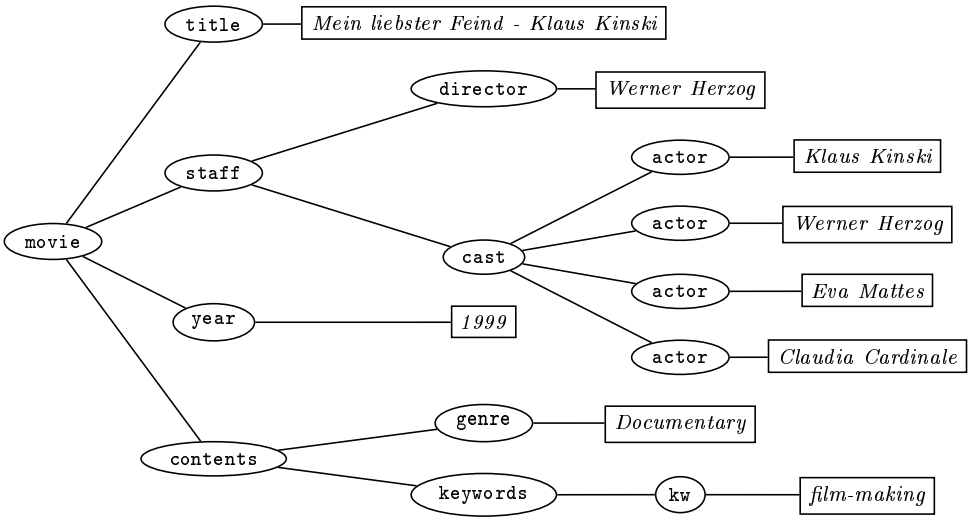
```
<!ELEMENT GENRE      (#PCDATA)                    >
<!ELEMENT SUMMARY    (#PCDATA)                    >
<!ELEMENT KEYWORDS   (KW+)                        >
<!ELEMENT COMMENT    (#PCDATA)                    >
<!ELEMENT KW         (#PCDATA)                    >
```

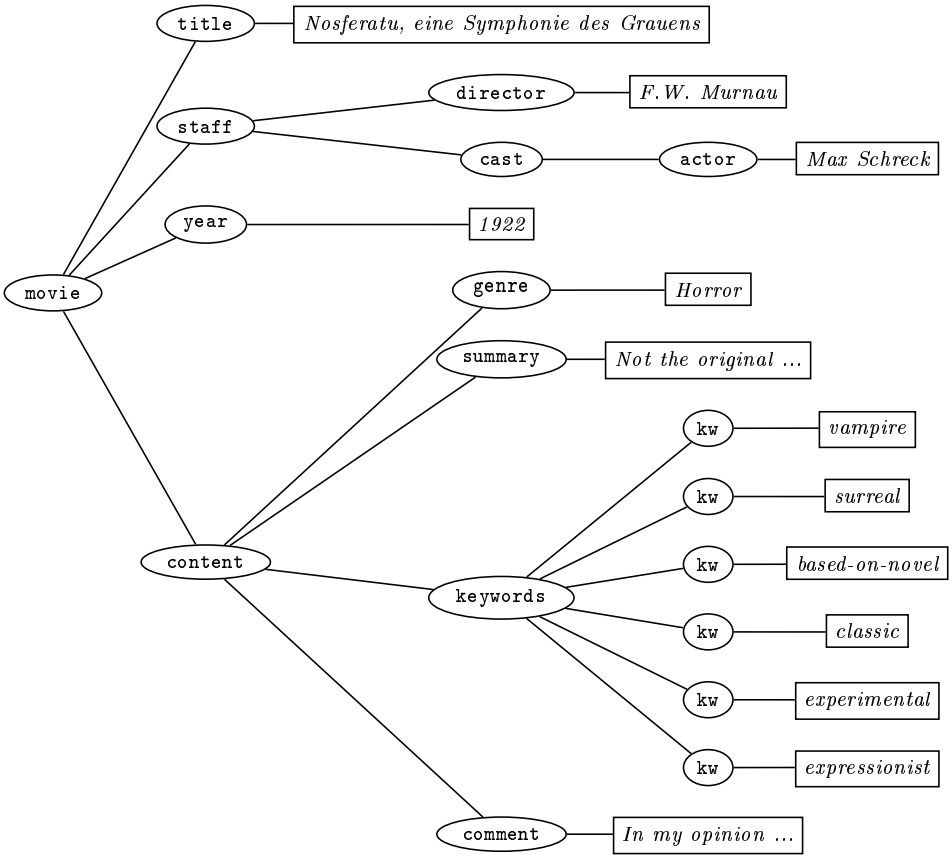
B.2 Tree View upon "Movie Collection"

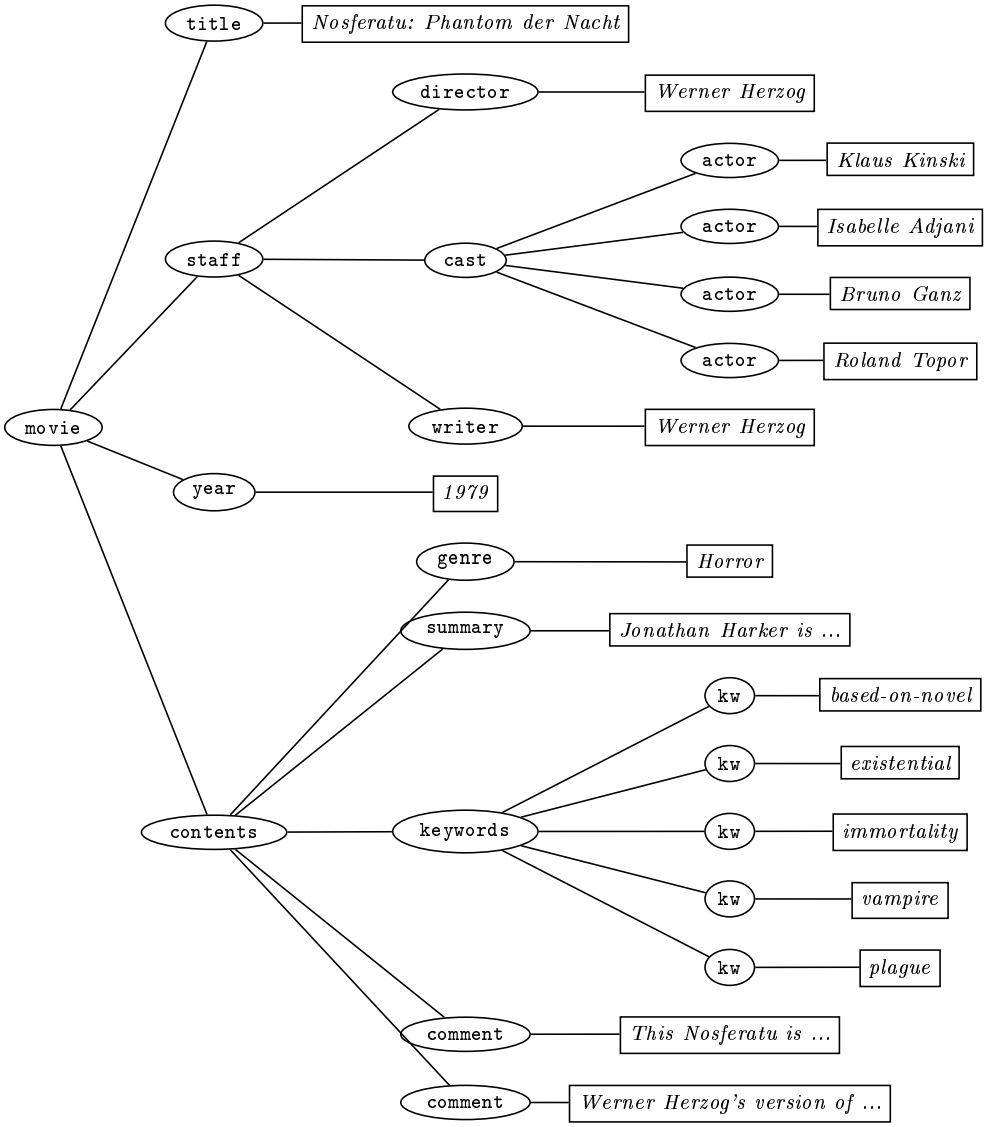
The trees depicted in the following describe a relational document structure. For the graphical presentation we had to abbreviate some texts. This indicated by "...". The full text can be found in Appendix B.3. Due to layout reasons the trees are arranged from left to right, i.e. the root of a tree is on the left side, and the leaves are on the right sides. The left-to-right-order between the nodes is given by the order in the depicted trees, i.e. if a node d appears above a node e so that e is no descendant of d, then $d <_{lr} e$ holds. Note that the relational document structure comprises the forest consisting of all trees in the following.

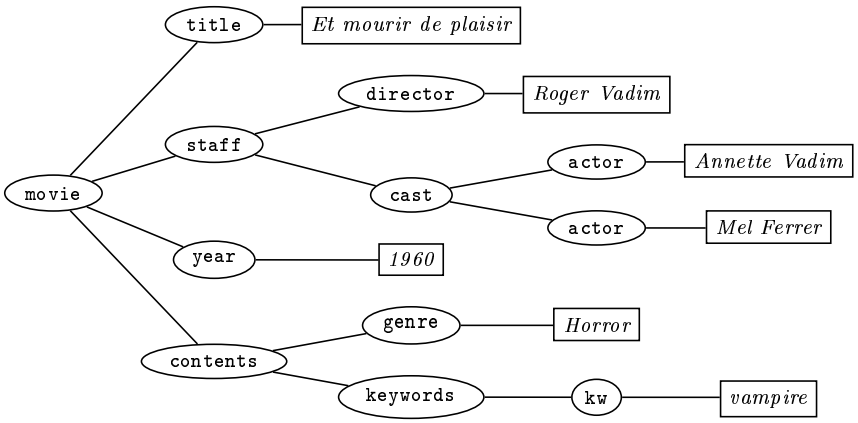
Identifiers Every node in the forest has an identifier with which we refer in the text of this work to the node. These identifiers are composed of an lowercase letter identifying each tree (`a` for the first document, `b` for the second, etc.) followed by a number pointing to the node. The numbers result from visiting the nodes in a pre-order traversal. For example, node `a1` is the node labeled movie in the first document, and node `e6` is the text node containing the text *Roger Vadim* in the fifth document. Thus, the identifiers reflect the preorder relation between the nodes.

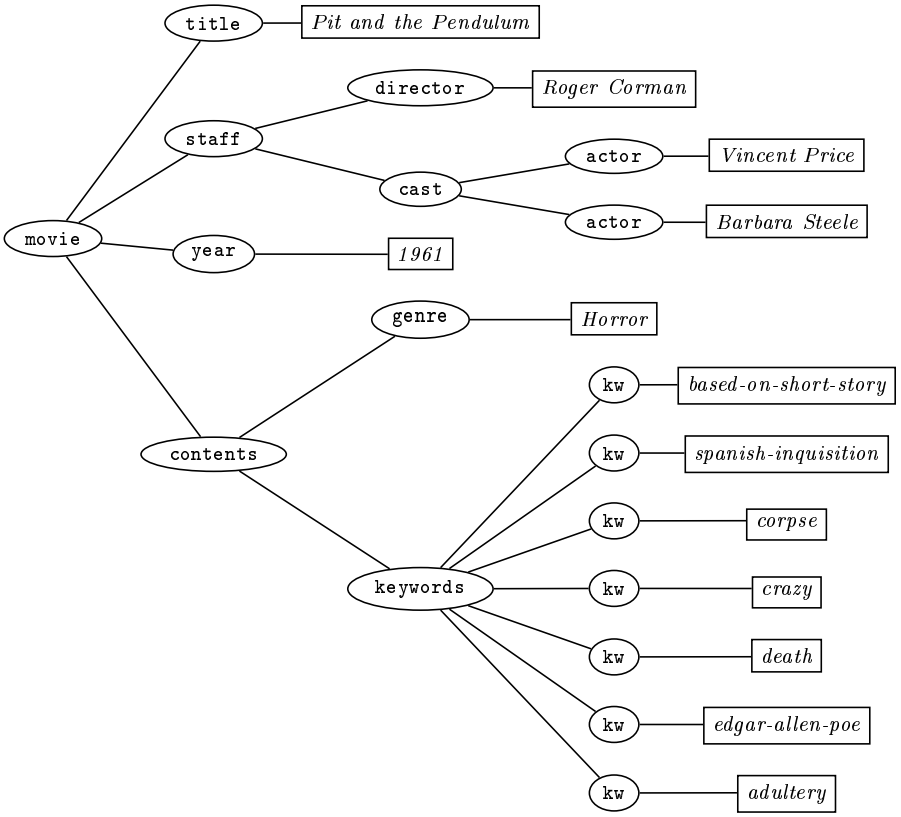


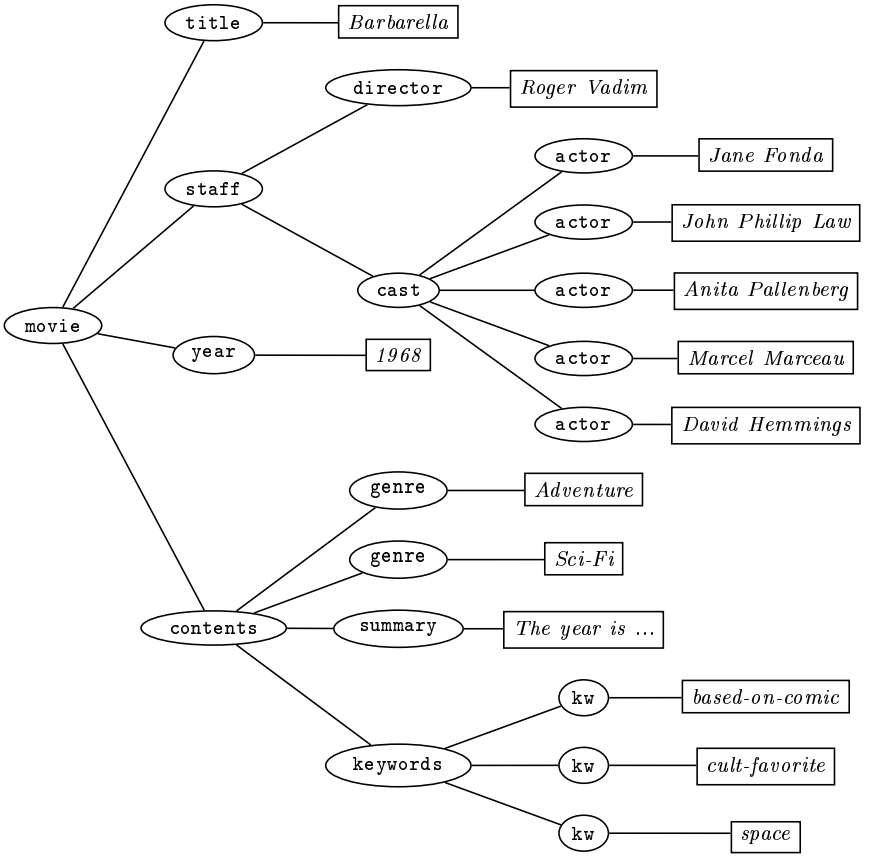

movie
title
Mein liebster Feind - Klaus Kinski
staff
director
Werner Herzog
cast
actor
Klaus Kinski
actor
Werner Herzog
actor
Eva Mattes
actor
Claudia Cardinale
year
1999
contents
genre
Documentary
keywords
kw
film-making

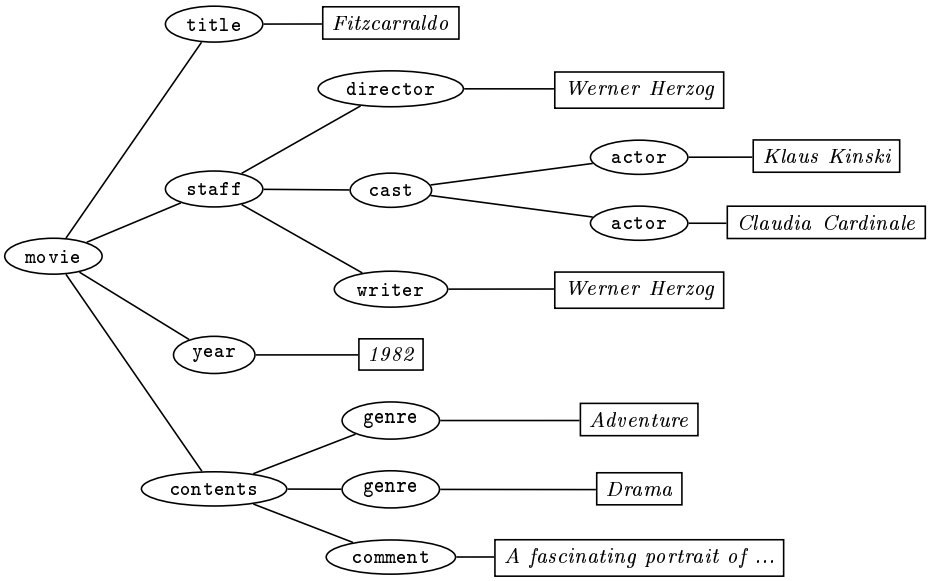

movie
title
Nosferatu, eine Symphonie des Grauens
staff
director
F.W. Murnau
cast
actor
Max Schreck
year
1922
genre
Horror
summary
Not the original ...
content
keywords
kw
vampire
kw
surreal
kw
based-on-novel
kw
classic
kw
experimental
kw
expressionist
comment
In my opinion ...

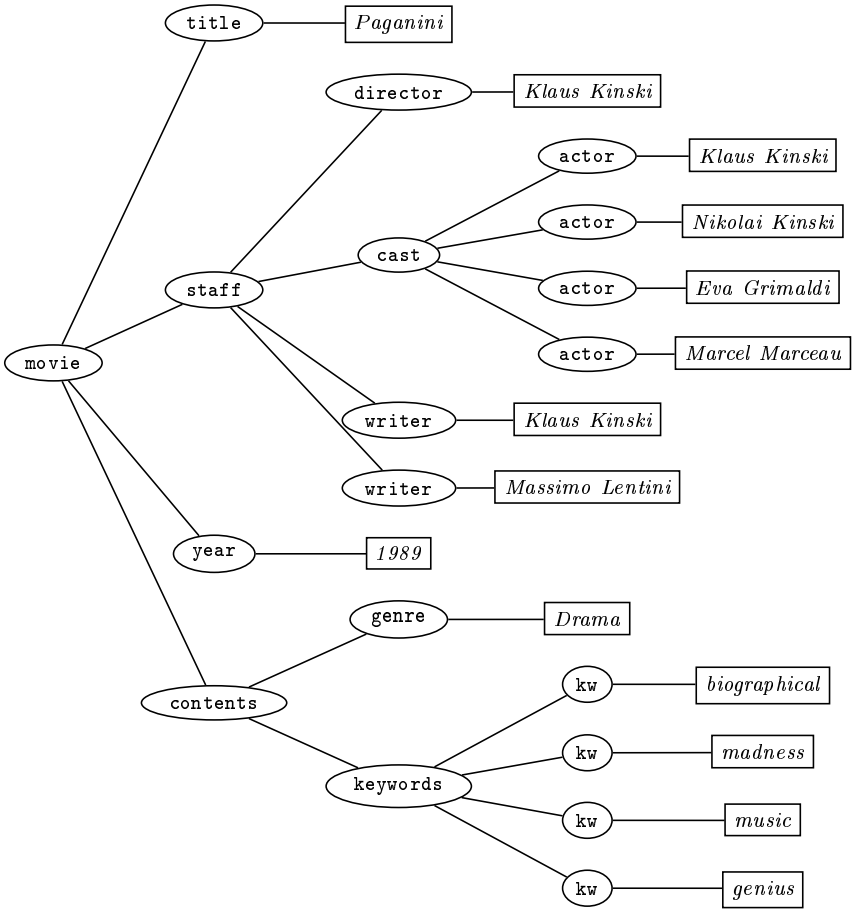

title
Nosferatu: Phantom der Nacht
director
Werner Herzog
actor
Klaus Kinski
actor
Isabelle Adjani
staff
cast
actor
Bruno Ganz
actor
Roland Topor
movie
writer
Werner Herzog
year
1979
genre
Horror
summary
Jonathan Harker is ...
kw
based-on-novel
kw
existential
contents
keywords
kw
immortality
kw
vampire
kw
plague
comment
This Nosferatu is ...
comment
Werner Herzog's version of ...

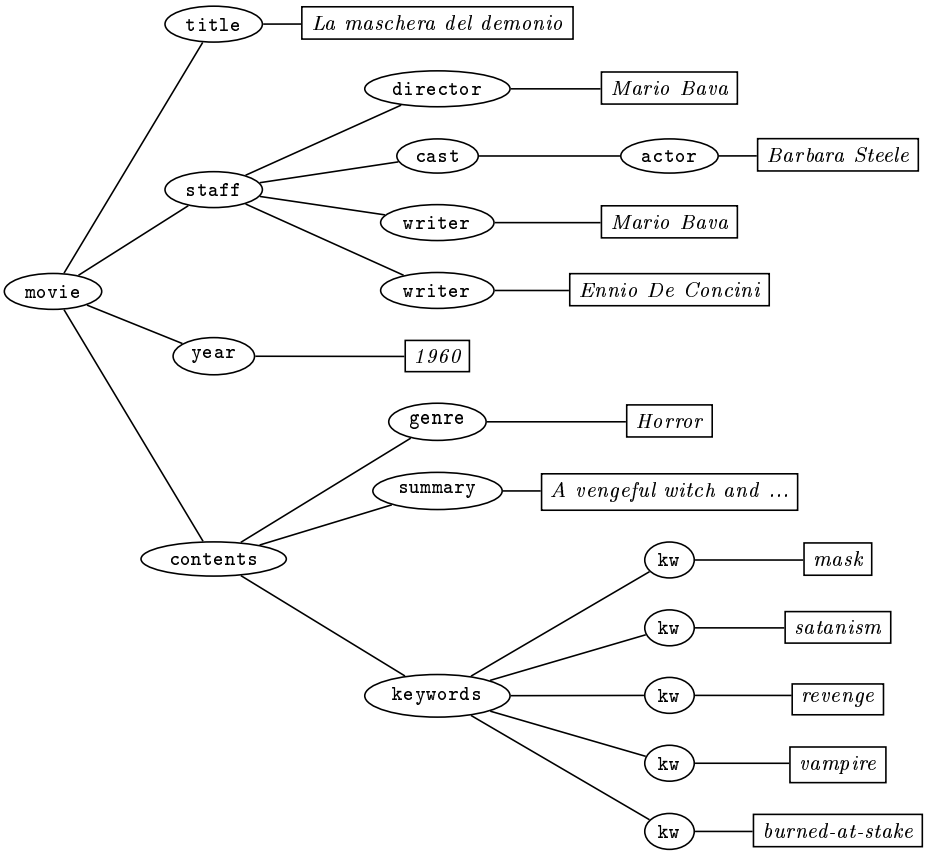

title
Et mourir de plaisir
director
Roger Vadim
staff
actor
Annette Vadim
movie
cast
actor
Mel Ferrer
year
1960
genre
Horror
contents
keywords
kw
vampire


movie
title
Pit and the Pendulum
staff
director
Roger Corman
cast
actor
Vincent Price
actor
Barbara Steele
year
1961
contents
genre
Horror
keywords
kw
based-on-short-story
kw
spanish-inquisition
kw
corpse
kw
crazy
kw
death
kw
edgar-allen-poe
kw
adultery


movie
title
Barbarella
staff
director
Roger Vadim
cast
actor
Jane Fonda
actor
John Phillip Law
actor
Anita Pallenberg
actor
Marcel Marceau
actor
David Hemmings
year
1968
contents
genre
Adventure
genre
Sci-Fi
summary
The year is ...
keywords
kw
based-on-comic
kw
cult-favorite
kw
space


title
Fitzcarraldo
director
Werner Herzog
actor
Klaus Kinski
staff
cast
actor
Claudia Cardinale
movie
writer
Werner Herzog
year
1982
genre
Adventure
contents
genre
Drama
comment
A fascinating portrait of ...


title
Paganini
director
Klaus Kinski
actor
Klaus Kinski
actor
Nikolai Kinski
cast
staff
actor
Eva Grimaldi
movie
actor
Marcel Marceau
writer
Klaus Kinski
writer
Massimo Lentini
year
1989
genre
Drama
contents
kw
biographical
kw
madness
keywords
kw
music
kw
genius



B.3 XML View upon Movie Collection

```
<?xml version='1.0' encoding='ISO-8859-1' ?>
<!DOCTYPE MOVIE SYSTEM 'movie.dtd'>
<MOVIE>
  <TITLE>
    8 1/2
  </TITLE>

  <STAFF>
    <DIRECTOR> Federico Fellini </DIRECTOR>
    <CAST>
      <ACTOR> Marcello Mastroianni </ACTOR>
      <ACTOR> Anouk Aimee </ACTOR>
      <ACTOR> Claudia Cardinale </ACTOR>
      <ACTOR> Barbara Steele </ACTOR>
    </CAST>
  </STAFF>

  <YEAR> 1963 </YEAR>

  <CONTENTS>
    <GENRE> Drama </GENRE>
    <SUMMARY>
      Guido is a film director, trying to relax after his last big
```

```
      hit.  He can't get a moments peace, however, with the people who
      have worked with him in the past constantly looking for more
      work. He wrestles with his conscience, but is unable to come up
      with a new idea. While thinking, he starts to recall major
      happenings in his life, and all the women he has loved and left.
      An autobiographical film of Fellini, about the trials and
      tribulations of film making.
    </SUMMARY>
    <KEYWORDS>
     <KW> adultery </KW>
     <KW> beatniks </KW>
     <KW> film-making </KW>
     <KW> midlife-crisis </KW>
     <KW> surreal </KW>
    </KEYWORDS>
  </CONTENTS>
</MOVIE>


<?xml version='1.0' encoding='ISO-8859-1' ?>
<!DOCTYPE MOVIE SYSTEM 'movie.dtd'>
<MOVIE>
  <TITLE> Mein liebster Feind - Klaus Kinski </TITLE>

  <STAFF>
    <DIRECTOR> Werner Herzog </DIRECTOR>
    <CAST>
      <ACTOR> Klaus Kinski </ACTOR>
      <ACTOR> Werner Herzog </ACTOR>
      <ACTOR> Eva Mattes </ACTOR>
      <ACTOR> Claudia Cardinale </ACTOR>
    </CAST>
  </STAFF>

  <YEAR> 1999 </YEAR>

  <CONTENTS>
    <GENRE> Documentary </GENRE>
    <KEYWORDS>
     <KW> film-making </KW>
    </KEYWORDS>
  </CONTENTS>
</MOVIE>


<?xml version='1.0' encoding='ISO-8859-1' ?>
<!DOCTYPE MOVIE SYSTEM 'movie.dtd'>
<MOVIE>
  <TITLE> Nosferatu, eine Symphonie des Grauens </TITLE>

  <STAFF>
    <DIRECTOR> F.W. Murnau </DIRECTOR>
    <CAST>
      <ACTOR> Max Schreck </ACTOR>
    </CAST>
```

```
  </STAFF>

  <YEAR> 1922 </YEAR>

  <CONTENTS>
    <GENRE> Horror </GENRE>
    <SUMMARY>
      Not the original Dracula movie (but close enough for most
      people), this follows the familiar story of Count Orloc moving
      from his ruined castle to the city of Wisborg, after the visit
      of one Jonathan Harker.  Once there he becomes involved with
      Jonathan's fiancee Nina, who alone holds the power to destroy
      him.
    </SUMMARY>
    <KEYWORDS>
     <KW> vampire </KW>
     <KW> surreal </KW>
     <KW> based-on-novel </KW>
     <KW> classic </KW>
     <KW> experimental </KW>
     <KW> expressionist </KW>
    </KEYWORDS>
    <COMMENT>
      In my opinion, this is the best vampire movie made, ever. But
      beyond that, it is an incredibly powerful visual film as well.
      Max Shreck looks simply fantastic in the makeup that SHOULD have
      defined the role that would be inherited by Bela Lugosi. The
      only vampire movie that stands up to this one is "Vampyr".
    </COMMENT>
  </CONTENTS>
</MOVIE>


<?xml version='1.0' encoding='ISO-8859-1' ?>
<!DOCTYPE MOVIE SYSTEM 'movie.dtd'>
<MOVIE>
  <TITLE> Nosferatu: Phantom der Nacht </TITLE>

  <STAFF>
    <DIRECTOR> Werner Herzog </DIRECTOR>
    <CAST>
      <ACTOR> Klaus Kinski </ACTOR>
      <ACTOR> Isabelle Adjani </ACTOR>
      <ACTOR> Bruno Ganz </ACTOR>
      <ACTOR> Roland Topor </ACTOR>
    </CAST>
    <WRITER> Werner Herzog </WRITER>
  </STAFF>

  <YEAR> 1979 </YEAR>

  <CONTENTS>
    <GENRE> Horror </GENRE>
    <SUMMARY>
```

```
      Jonathan Harker is sent away to Count Dracula's castle to sell
      him a house in Virna, where he lives.  But Count Dracula is a
      vampire, an undead ghoule living of men's blood. Inspired by a
      photograph of Lucy Harker, Jonathan's wife, Dracula moves to
      Virna, bringing with him death and plague... An unusually
      contemplative version of Dracula, in which the vampire bears the
      cross of not being able to get old and die.
    </SUMMARY>
    <KEYWORDS>
      <KW> based-on-novel </KW>
      <KW> existential </KW>
      <KW> immortality </KW>
      <KW> vampire </KW>
      <KW> plague </KW>
    </KEYWORDS>
    <COMMENT>
      This Nosferatu is caught in insanity, because he cannot enjoy
      his immortality, and wants to infect them with his lunacy. Maybe
      ridiculous for some audience, but this is the most horrifying
      vampire movie for a man with heart of vampire. Fantastic beauty
      of Isabelle Adjani also accents the nightmare.
    </COMMENT>
    <COMMENT>
      Werner Herzog's version of Murnau's classic NOSFERATU is a
      captivating experience. Klaus Kinski is perfect as Count
      Dracula. He brilliantly conveys the loneliness and sadness of a
      creature who longs to be human. Count Dracula is the victim in
      this film, he does not enjoy his immortality and wants only to
      live, love and die like a human. Isabelle Adjani's ethereal
      beauty punctuates her ghostlike performance as Lucy, and Bruno
      Ganz turns in another solid performance as Jonathan.
    </COMMENT>
  </CONTENTS>
</MOVIE>


<?xml version='1.0' encoding='ISO-8859-1' ?>
<!DOCTYPE MOVIE SYSTEM 'movie.dtd'>
<MOVIE>
  <TITLE> Et mourir de plaisir </TITLE>

  <STAFF>
    <DIRECTOR> Roger Vadim </DIRECTOR>
    <CAST>
      <ACTOR> Annette Vadim </ACTOR>
      <ACTOR> Mel Ferrer </ACTOR>
    </CAST>
  </STAFF>

  <YEAR> 1960 </YEAR>

  <CONTENTS>
    <GENRE> Horror </GENRE>
    <KEYWORDS>
```

```
     <KW> vampire </KW>
    </KEYWORDS>
  </CONTENTS>
</MOVIE>


<?xml version='1.0' encoding='ISO-8859-1' ?>
<!DOCTYPE MOVIE SYSTEM 'movie.dtd'>
<MOVIE>
  <TITLE> Pit and the Pendulum </TITLE>

  <STAFF>
    <DIRECTOR> Roger Corman </DIRECTOR>
    <CAST>
      <ACTOR> Vincent Price </ACTOR>
      <ACTOR> Barbara Steele </ACTOR>
    </CAST>
  </STAFF>

  <YEAR> 1961 </YEAR>

  <CONTENTS>
    <GENRE> Horror </GENRE>
    <KEYWORDS>
     <KW> based-on-short-story </KW>
     <KW> spanish-inquisition </KW>
     <KW> corpse </KW>
     <KW> crazy </KW>
     <KW> death </KW>
     <KW> edgar-allen-poe </KW>
     <KW> adultery </KW>
    </KEYWORDS>
  </CONTENTS>
</MOVIE>


<?xml version='1.0' encoding='ISO-8859-1' ?>
<!DOCTYPE MOVIE SYSTEM 'movie.dtd'>
<MOVIE>
  <TITLE> Barbarella </TITLE>

  <STAFF>
    <DIRECTOR> Roger Vadim </DIRECTOR>
    <CAST>
      <ACTOR> Jane Fonda </ACTOR>
      <ACTOR> John Phillip Law </ACTOR>
      <ACTOR> Anita Pallenberg </ACTOR>
      <ACTOR> Marcel Marceau </ACTOR>
      <ACTOR> David Hemmings </ACTOR>
    </CAST>
  </STAFF>

  <YEAR> 1968 </YEAR>

  <CONTENTS>
```

```
    <GENRE> Adventure </GENRE>
    <GENRE> Sci-Fi </GENRE>
    <SUMMARY>
      The year is 40.000. Peacefully floating around in zero-gravity
      Barbarella (Jane Fonda) is suddenly interrupted by a call from
      the President of Earth. A young scientist, Durand-Durand, is
      threatening the ancient universal peace and Barbarella is the
      chosen one to find him and save the world. During her mission,
      Barbarella never finds herself in a situation where it isn't
      possible to lose at least part of her already minimal dressing.
    </SUMMARY>
    <KEYWORDS>
     <KW> based-on-comic </KW>
     <KW> cult-favorite </KW>
     <KW> space </KW>
    </KEYWORDS>
  </CONTENTS>
</MOVIE>


<?xml version='1.0' encoding='ISO-8859-1' ?>
<!DOCTYPE MOVIE SYSTEM 'movie.dtd'>
<MOVIE>
  <TITLE> Fitzcarraldo </TITLE>

  <STAFF>
    <DIRECTOR> Werner Herzog </DIRECTOR>
    <CAST>
      <ACTOR> Klaus Kinski </ACTOR>
      <ACTOR> Claudia Cardinale </ACTOR>
    </CAST>
    <WRITER> Werner Herzog </WRITER>
  </STAFF>

  <YEAR> 1982 </YEAR>

  <CONTENTS>
    <GENRE> Adventure </GENRE>
    <GENRE> Drama </GENRE>
    <COMMENT>
      A fascinating portrait of a man whose love for the opera drives
      him on a peculiar mission to build an opera house in the South
      American rain forest. The characters are sharply developed
      without the use of unnecessary "reflective" dialogue. Herzog
      instead chooses to allow his actors to show us the characters
      almost wholly through expression and action.

      There is some beautiful camera work, and the script is always
      fascinating.
    </COMMENT>
  </CONTENTS>
</MOVIE>


<?xml version='1.0' encoding='ISO-8859-1' ?>
```

```
<!DOCTYPE MOVIE SYSTEM 'movie.dtd'>
<MOVIE>
  <TITLE> Paganini </TITLE>

  <STAFF>
    <DIRECTOR> Klaus Kinski </DIRECTOR>
    <CAST>
      <ACTOR> Klaus Kinski </ACTOR>
      <ACTOR> Nikolai Kinski </ACTOR>
      <ACTOR> Eva Grimaldi </ACTOR>
      <ACTOR> Marcel Marceau </ACTOR>
    </CAST>
    <WRITER> Klaus Kinski </WRITER>
    <WRITER> Massimo Lentini </WRITER>
  </STAFF>

  <YEAR> 1989 </YEAR>

  <CONTENTS>
    <GENRE> Drama </GENRE>
    <KEYWORDS>
     <KW> biographical </KW>
     <KW> madness </KW>
     <KW> music </KW>
     <KW> genius </KW>
    </KEYWORDS>
  </CONTENTS>
</MOVIE>


<?xml version='1.0' encoding='ISO-8859-1' ?>
<!DOCTYPE MOVIE SYSTEM 'movie.dtd'>
<MOVIE>
  <TITLE> La maschera del demonio </TITLE>

  <STAFF>
    <DIRECTOR> Mario Bava </DIRECTOR>
    <CAST>
      <ACTOR> Barbara Steele </ACTOR>
    </CAST>
    <WRITER> Mario Bava </WRITER>
    <WRITER> Ennio De Concini </WRITER>
  </STAFF>

  <YEAR> 1960 </YEAR>

  <CONTENTS>
    <GENRE> Horror </GENRE>
    <SUMMARY>
      A vengeful witch and her fiendish servant return from the grave
      and begin a bloody campaign to possess the body of the witch's
      beautiful look-alike descendant. Only the girl's brother and a
      handsome doctor stand in her way.
    </SUMMARY>
```

```
    <KEYWORDS>
     <KW> mask </KW>
     <KW> satanism </KW>
     <KW> revenge </KW>
     <KW> vampire </KW>
     <KW> burned-at-the-stake </KW>
    </KEYWORDS>
  </CONTENTS>
</MOVIE>
```

Bibliography

[Abi97] Serge Abiteboul. Querying semi-structured data. In *ICDT'97, Proc. 6th Int. Conf. on DB Theory*, 1997.

[ABS99] Serge Abiteboul, Peter Buneman, and Dan Suciu. *Data on the Web: From Relations to Semistructured Data and XML.* Morgan Kaufmann Publishers, 1999.

[ABY98] Omar Alonso and Ricardo Baeza-Yates. A model for visualizing large answers in WWW. In *18th Int. Conf. of the Chilean CS Society*, 1998.

[Alt99] AltaVista. *AltaVista Search Intranet Developer's Kit, Version 2.6*, 1999.

[AQM+97] Serge Abiteboul, Dallan Quass, Jason McHugh, Jennifer Widom, and Janet L. Wiener. The Lorel query language for semistructured data. *International Journal on Digital Libraries*, 1(1):68–88, 1997.

[AS92] Bernd Amann and Michel Scholl. Gram: A graph data model and query language. In *Proc. 4th ACM Conf. on Hypertext*, Theoretical Foundations, 1992.

[BC00] Angela Bonifati and Stefano Ceri. A comparative analysis of five XML query languages. *SIGMOD Record*, March 2000.

[BDHS96] Peter Buneman, Susan Davidson, Gerd Hillebrand, and Dan Suciu. A query language and optimization techniques for unstructured data. In *SIGMOD'96*, pages 505–516, 1996.

[Bou99] Ronald Bourret. XML and databases, December 1999. `www.informatik.tu-darmstadt.de/DVS1/staff/bourret/bourret.htm`.

[Bun97] Peter Buneman. Semistructured data. In *Proc. ACM PODS'97*, 1997.

[BYN96] R. Baeza-Yates and G. Navarro. Integrating contents and structure in text retrieval. *SIGMOD Record*, 25(1):67–79, 1996.

[BYNVdlF98] Ricardo Baeza-Yates, Gonzalo Navarro, Jess Vegas, and Pablo de la Fuente. A model and a visual query language for structured text. In *5th South American Symposium on String Processing and Information Retrieval (SPIRE'98)*, 1998.

[BYRN99a] Ricardo Baeza-Yates and Berthier Ribeiro-Neto. *Modern Information Retrieval*, chapter Multimedia IR: Models and Languages, pages 325–343. Addison-Wesley, 1999.

[BYRN99b] Ricardo Baeza-Yates and Berthier Ribeiro-Neto. *Modern Information Retrieval.* Addison-Wesley, 1999.

[BYRN99c] Ricardo Baeza-Yates and Berthier Ribeiro-Neto. *Modern Information Retrieval*, chapter User Interfaces and Visualization, pages 257–323. Addison-Wesley, 1999.

[CCB95] Charles L.A. Clarke, Gordon V. Cormack, and Forbes J. Burkowski. An algebra for structured text search and a framework for its implementation. *The Computer Journal*, 38(1):43–56, 1995.

[CCD+99] Stefano Ceri, Sara Comai, Ernesto Damiani, Piero Fraternali, Stefano Paraboschi, and Letizia Tanca. XML-GL: a graphical language for querying and restructuring XML documents. *Computer Networks*, 31(11–16):1171–1187, May 1999.

[CCLB95] Tiziana Catarci, Maria F. Costabile, Stefano Levialdi, and Carlo Batini. Visual query systems for databases: A survey. Technical Report SI/RR-95/17, Dipartimento di Scienze dell'Informazione, Universita' di Roma La Sapienza, 1995.

[CDG+97] Hubert Comon, Max Dauchet, Remi Gilleron, Florent Jacquemard, Denis Lugiez, Sophie Tison, and Marc Tommasi. Tree automata techniques and applications. Available online: `http://www.grappa.univ-lille3.fr/tata`, 1997.

[CM90] M. Consens and A. Mendelzon. Graphlog: a visual formalism of real life recursion. In *Proc. ACM PODS'90*, 1990.

[CM98] Mariano P. Consens and Tova Milo. Algebras for querying text regions: Expressive power and optimization. *Journal of Computer and System Sciences*, 57(3):272–288, December 1998.

[CSC97] Tiziana Catarci, Giuseppe Santucci, and John Cardiff. Graphical interaction with heterogeneous databases. *The VLDB Journal*, 6(2):97–120, May 1997.

[DFF+98] Alin Deutsch, Mary Fernandez, Daniela Florescu, Alon Levy, and Dan Suciu. XML-QL: A query language for XML. Submission to the WWW Consortium: `http://www.w3.org/TR/NOTE-xml-ql/`, August 1998.

[DFF+99] Alin Deutsch, Mary Fernandez, Daniela Florescu, Alon Levy, David Maier, and Dan Suciu. Querying XML data. *IEEE Data Bulletin*, 22(3):10–18, 1999.

[FBY92a] William B. Frakes and Ricardo Baeza-Yates, editors. *Information Retrieval. Data Structures & Algorithms*, chapter Extended Boolean Models, pages 393–418. Prentice Hall, 1992.

[FBY92b] William B. Frakes and Ricardo Baeza-Yates, editors. *Information Retrieval: Data Structures and Algorithms*. Prentice-Hall, Englewood Cliffs, NJ 07632, USA, 1992.

[FGR98] Norbert Fuhr, Norbert Gövert, and Thomas Röllecke. DOLORES: A system for logic-based retrieval of multimedia objects. In *Proc. ACM SIGIR '98*, 1998.

[FK99] Daniela Florescu and Donald Kossmann. A performance evaluation of alternative mapping schemes for storing XML data in a relational database. Technical report, Inria, Institut National de Recherche en Informatique et en Automatique, 1999.

[FLM98] D. Florescu, A. Levy, and A. Mendelzon. Database techniques for the world-wide-web: A survey. *SIGMOD Record*, 27(3), 1998.

[Fra96] Angelika Franzke. Querying graph structures with G^2QL. Fachberichte Informatik 10-96, Universität Koblenz-Landau, 1996.

[Fre82] Eugene C. Freuder. A sufficient condition for backtrack-free search. *Journal of the ACM*, 29(1):24–32, January 1982.

[FSW99] Mary Fernandez, Jérôme Siméon, and Philip Wadler. XML query languages: Experiences and exemplars. Draft, `http://www-db.research.bell-labs.com/user/simeon/xquery.ps`, 1999.

[Fuh92] Norbert Fuhr. Probabilistic models in information retrieval. *The Computer Journal*, 35(3):243–255, 1992.

[Fuh96] N. Fuhr. Models for integrated Information Retrieval and database systems. *IEEE Data Engineering Bulletin*, 19(1):3–13, June 1996.

[Gol90] C. F. Goldfarb. *The SGML Handbook*. Oxford University Press, 1990.

[GP98] Charles F. Goldfarb and Paul Prescod. *The XML Handbook*. Prentice-Hall, 1998.

[GPdBG94] Marc Gyssens, Jan Paredaens, Jan Van den Bussche, and Dirk Van Gucht. A graph-oriented object database model. *IEEE Transactions on Knowledge and Data Engineering*, 6(4):572–586, August 1994.

[GPT93] Marc Gemis, Jan Paredaens, and Inge Thyssens. A visual database management interface based on GOOD. In *Proc. Int. Workshop on Interfaces to Database Systems*, 1993.

[GS84] Ferenc Gécseg and Magnus Steinby. *Tree Automata*. Akadémiai Kiadó, Budapest, 1984.

[GT87] Gaston Gonnet and Frank W. Tompa. Mind your grammar: a new approach to modelling text. In *Proc. VLDB'87*, 1987.

[GVU99] 10th WWW User Survey. Graphic, Visualization, & Usability Center's (GVU), Georgia Institute of Technology, 1999. `http://www.gvu.gatech.edu/user_surveys/survey-1998-10/`.

[GW97] Roy Goldman and Jennifer Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In *Proc. VLDB'97*, 1997.

[GW98] R. Goldman and J. Widom. Interactive query and search in semistructured databases. In *WebDB'98, Proc. Int. Workshop on the Web and Databases*, 1998.

[HPS98] Eva Hajicova, Jarmila Panevova, and Petr Sgall. Language resources need annotations to make them really reusable: The prague dependency treebank. In *First International Conference on Language Resources and Evaluation, Granada, Spain*, pages 705 – 711, 1998.

[IMD00] The Internet Movie Database. `http://www.imdb.org`, May 2000.

[ISO86] ISO (International Organization for Standardization). *Information Processing - Text and Office Systems - Standard General MarkUp Language (SGML).* ISO8879, 1986.

[ISO89] ISO (International Organization for Standardization). *Information Processing - Text and Office Systems - Office Document Architecture (ODA) and interchange format.* ISO8613, 1989.

[JK96] Jani Jaakkola and Pekka Kilpeläinen. Using sgrep for querying structured text files. In *Proc. of SGML Finland '96*, 1996.

[JK99] Jani Jaakkola and Pekka Kilpeläinen. Nested text-region algebra. Technical Report C-1999-2, Department of Computer Science, University of Helsinki, 1999.

[KB96] Setrag Khoshafian and A. Brad Baker. *MultiMedia and Imaging Databases.* Morgan Kaufmann, 1996.

[Kil92] Pekka Kilpeläinen. *Tree Matching Problems with Applications to Structured Text Databases.* PhD thesis, Dept. of Computer Science, University of Helsinki, 1992.

[KN98a] Eila Kuikka and Erja Nikunen. Survey of software for structured text. Technical Report A/1998/1, University of Helsinki, Department of Computer Science and Applied Mathematics, 1998. `http://www.cs.uku.fi/~kuikka/systems.html`.

[KN98b] Sadao Kurohashi and Makoto Nagao. Buidling a japanese parsed corpus while improving the parsing system. In *First International Conference on Language Resources and Evaluation, Granada, Spain*, 1998.

[Lee94] Joon Ho Lee. Properties of extended boolean models in information retrieval. In *Proc. ACM SIGIR'94*, 1994.

[Loe94] Arjan Loeffen. Text databases: A survey of text models and systems. *SIGMOD Record*, 23(1):97–106, March 1994.

[Mac77] Alan K. Mackworth. Consistency in networks of constraints. *Artificial Intelligence*, 8, 1977.

[MAG+97] Jason McHugh, Serge Abiteboul, Roy Goldman, Dallan Quass, and Jennifer Widom. Lore: A database management system for semistructured data. *SIGMOD Record*, 26(3), 1997.

[Meu98] Holger Meuss. Indexed tree matching with complete answer representations. In *PODDP'98, Proc. 4th Int. Workshop on Principles of Digital Document Processing*, LNCS 1481. Springer, 1998.

[MF85] Alan K. Mackworth and Eugene C. Freuder. The complexity of some polynomial network consistency algorithms for constraint satisfaction problems. *Artificial Intelligence*, 25(1):65–74, 1985.

[MH86] Roger Mohr and Thomas C. Henderson. Arc and path consistency revisited. *Artificial Intelligence*, 28:225–233, 1986.

[Miz98] Stefano Mizzaro. How many relevances in information retrieval? *Interacting with Computers*, 10(3):303–320, 1998.

[Mon74] Ugo Montanari. Networks of constraints: Fundamental properties and applications to picture processing. *Information Sciences*, 7:95–132, 1974.

[MS99a] Holger Meuss and Christian Strohmaier. Improving index structures for structured document retrieval. In *IRSG'99, 21st Annual Colloquium on IR Research*, 1999.

[MS99b] Tova Milo and Dan Suciu. Index structures for path expressions. In *ICDT'99, Proc. 6th Int. Conf. on DB Theory*, 1999.

[MS00] Holger Meuss and Klaus U. Schulz. Complete answer aggregates for answer mappings to sequence, tree and graph databases. Technical Report 00-125, CIS, University of Munich, 2000.

[MSM93] Mitchell P. Marcus, Beatrice Santorini, and Mary Ann Marcinkiewicz. Building a large annotated corpus of english: the Penn Treebank. *Computational Linguistics*, 1993.

[NBY95] Gonzalo Navarro and Ricardo Baeza-Yates. A language for queries on structure and contents of textual databases. In *Proc. ACM SIGIR'95*, pages 93–101, 1995.

[NBY97] Gonzalo Navarro and Ricardo Baeza-Yates. Proximal Nodes: A model to query document databases by contents and structure. *ACM Transactions on Information Systems*, 15(4):400–435, 1997.

[NS00] Frank Neven and Thomas Schwentick. Expressive and efficient pattern languages for tree-structured data. In *Proc. ACM PODS'00*, 2000.

[Oas] The SGML/XML Web Page. `http://www.oasis-open.org/cover/`.

[OMM98] Jürgen Oesterle and Petra Maier-Meyer. The gnop (german noun phrase) treebank. In *First International Conference on Language Resources and Evaluation*, pages 699 – 703, 1998.

[OP98] Iadh Ounis and Marius Paşca. Modeling, indexing and retrieving images using conceptual graphs. In *DEXA '98*, 1998.

[PGMW95] Yannis Papakonstantinou, Hector Garcia-Molina, and Jennifer Widom. Object exchange across heterogeneous information sources. In *ICDT'95, Proc. of the 11th Int. Conf. on Data Engineering*, 1995.

[Pre98] Paul Prescod. Formalizing SGML and XML instances and schemata with forest automata theory. Draft, `http://www.prescod.net/forest/shorttut/`, May 1998.

[Reu99] Out of the Abyss: Surviving the Information Age. Reuters Limited, London, UK, 1999. `http://www.risk.reuters.com/rbb/research/newresframe.htm`.

[RLS98] Jonathan Robie, Joe Lapp, and David Schach. XML query language (XQL), 1998. `http://www.w3.org/TandS/QL/QL98/pp/xql.html`.

[Rou90] Ronald Rousseau. Extended boolean retrieval : A heuristic approach? In *Proc. ACM SIGIR'90*, 1990.

[SAB93] Gerard Salton, James Allan, and Chris Buckley. Approaches to passage retrieval in full text information systems. In *Proc. ACM SIGIR'93*, pages 49–58, 1993.

[SBKU98] Wojciech Skut, Thorsten Brants, Brigitte Krenn, and Hans Uszkoreit. A linguistically interpreted corpus of german newspaper text. In *First International Conference on Language Resources and Evaluation, Granada, Spain*, 1998.

[SFW83] Gerard Salton, Edward A. Fox, and Harry Wu. Extended Boolean information retrieval. *Communications of the ACM*, 26(11):1022–1036, November 1983.

[Shn98] Ben Shneiderman. *Designing the User Interface: Strategies for Effective Human-Computer Interaction*. Addison-Wesley, third edition, 1998.

[SJJ98] Dongwook Shin, Hyuncheol Jang, and Honglan Jin. BUS: An effective indexing and retrieval scheme in structured documents. In *DL'98: Proc. 3rd Int. ACM Conf. on Digital Libraries*, 1998.

[SLG95] Lena Santamarta, Nikolaj Lindberg, and Björn Gambäck. Towards building a swedish treebank. In *Proceedings of the 10th Nordic Conference of Computational Linguistics*, volume Short Papers, pages 37–40, Helsinki, Finland, 1995.

[SM83] Gerard Salton and Michael J. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, Tokio, 1983.

[SM99] Christian Strohmaier and Holger Meuss. A filter for structured document retrieval. Technical Report 99-123, CIS, University of Munich, 1999.

[ST94] A. Salminen and F. W. Tompa. PAT expressions: an algebra for text search. *Acta Linguistica Hungarica*, 41(1-4):277–306, 1994.

[STZ+99] Jayavel Shanmugasundaram, Kristin Tufte, Chun Zhang, Gang He, David J. DeWitt, and Jeffrey F. Naughton. Relational databases for querying XML documents: Limitations and opportunities. In *VLDB'99*, 1999.

[Suc98] Dan Suciu. An overview of semistructured data. *SIGACT News*, 29(4), 1998.

[Sun99] Java 2 platform, 1999. available online at `http://java.sun.com/products/jdk/1.2/`.

[Ull89] Jeffrey D. Ullman. *Database and Knowledge-Base Systems, Volumes I and II*. Computer Science Press, 1989.

[vH94] Eric van Herwijnen. *Practical SGML 2nd edn.* Kluwer, 1994.

[vR79] Cornelis J. van Rijsbergen. *Information Retrieval*. Butterworths, 1979. available online at `http://www.dcs.glasgow.ac.uk/Keith/Preface.html`.

[W3C98a] World Wide Web Consortium: Extensible Markup Language (XML) 1.0. W3C Recommendation, February 1998. `http://http://www.w3.org/TR/REC-xml`.

[W3C98b] World Wide Web Consortium. *QL'98 - The Query Languages Workshop*, December 1998. `http://www.w3.org/TandS/QL/QL98`.

[W3C99a] XML Schema Part 1: Structures. W3C Working Draft, `http://www.w3.org/TR/1999/WD-xmlschema-1-19991105`, November 1999. Eds. Henry S. Thompson and David Beech and Murray Maloney Noah Mendelsohn.

[W3C99b] XML Schema Part 2: Datatypes. W3C Working Draft, `http://www.w3.org/TR/1999/WD-xmlschema-2-19991105/`, November 1999. Eds. Paul V. Biron and Ashok Malhotra.

[W3C00a] Extensible stylesheet language (XSL). W3C Working Draft, `http://www.w3.org/TR/xsl/`, January 2000. Eds. Sharon Adler and Anders Berglund and Jeff Caruso and Stephen Deach and Alex Milowski and Scott Parnell and Jeremy Richman and Steve Zilles.

[W3C00b] World Wide Web Consortium: XML Query Requirements. W3C Working Draft, `http://www.w3.org/TR/2000/WD-xmlquery-req-20000131`, January 2000. Eds. Peter Frankhauser and Massimo Marchiori and Jonathan Robie.

[WCB+00] Fanny Wattez, Sophie Cluet, Veronique Benzaken, Guy Ferran, and Christian Fiegel. Benchmarking queries over trees: Learning the hard truth the hard way. In *Proc. ACM SIGMOD'00*, 2000.

[Wei99] Stuart Weibel. The state of the dublin core metadata initiative april 1999. *D-Lib Magazine*, 5(4), April 1999.

[Zlo75] Moshé Zloof. Query by example. In *Proc. National Computer Conference*, volume 44. AFIPS, 1975.

Index