Abstract

XML is emerging as a standard format for information interchange and storage of structured information. The wide-spread use of XML has sparked the interest of both the database and information retrieval research communities. XML databases are designed to store and query large volumes of XML data. Structured information retrieval or XML-IR is the application of information retrieval concepts and techniques to search structured data, usually in the form of documents in XML format.

The PF/Tijah XML information retrieval (XML-IR) system combines the expressive power of the XML Query language (XQuery) with techniques for structured information retrieval. PF/Tijah provides an extension, based on the the TIJAH XML-IR research system, to the Pathfinder XML database. Similar to traditional database systems, the PF/Tijah extension is structured along three layers. The conceptual level deals with the user’s search request in the form of NEXI queries. The logical level deals with these queries expressed in the Score Region Algebra (SRA). The physical level provides implementations of the SRA operators on top of the MonetDB open source database kernel.

In this thesis, the physical level implementation of the PF/Tijah XML-IR system is examined. The implementation of optimized IR primitives on top of the MonetDB relational database kernel is demonstrated. The influence of intermediate result size reduction on efficiency and retrieval effectiveness is investigated. Small-scale tests of the individual SRA operators combined with large-scale experiments based on the INEX 2004 and 2005 evaluation initiative methods show that large performance improvements can be achieved with only limited reduction in retrieval effectiveness.
Preface

When I started my graduation project in August of 2005, there was only a vague idea of what I was going to do: to integrate an XML database (Pathfinder) and an XML information retrieval system (TIIAH) into a generally usable product. This is the gist of the description that Djoerd Hiemstra and I composed for my project. A tall order, especially for someone who had never seen the insides (or outsides, for that matter) of either system before. One of these systems was a mature, very complex product with sometimes undocumented inner workings; the other was essentially only a collection of tools created to perform a very specific set of experiments. Fortunately, people with more experience in these matters went ahead and created the foundation: Henning Rode and Jan Flokstra created the low-level index structure (where the XML data is stored) and the user interface necessary to support querying on this data. My job was then to connect the dots: provide implementations of the XML-IR primitives on top of the new index structure. In addition, I provided some documentation on the design, implementation and usage of our new XML-DB/IR system on our wiki. In this manner the PF/Tijah XML-IR system was born.

For a long time, we thought that just describing the low-level implementation of the IR primitives would not result in a sufficiently ‘scientific’ product: I would also have to do some research. We finally settled on research that is closely coupled to the implementation of our XML-IR system: the optimization of query execution. This thesis then describes both aspects: it provides insight in how we achieved a fast XML-IR system by using optimized data structures and algorithms, and it describes how we made our system even faster by scientifically ’cutting corners’: reducing intermediate result sizes. This direction of research was prompted by users and developers of the ‘old’ XML-IR system (TIIAH), who implemented some of the principles of this type of optimization. There was however no research to show the effect on retrieval effectiveness that these optimizations might have. This thesis provides that research.

These activities took quite some time, caused mostly by the amount of reading (code and prose) I had to do to get up to speed on the inner workings of our system. The process was slowed yet further by other, more personal (or actually, professional) concerns: I co-started a small company in the beginning of 2006, with our first large project following shortly after. I was able to juggle these activities for quite some time before I decided to focus on graduating: this would benefit our company as well. This turns out to have been a good idea.

By completing this thesis, I can finally answer all those people who for comfortably more than a year kept asking: ‘When will you be graduating?!!’ I would like to thank all those people, especially my dear wife Annelies, who asked the question more than anyone. I would also like to thank Djoerd, Henning and Jan, for a very pleasant and stimulating cooperation. Just now as I write this, Jan enters, while proudly proclaiming: ‘We can do everything from XQuery now!’ An impressive achievement,
this PF/Tijah of ours; I am thankful to have been able to contribute.

If you would like more information about this thesis and the work I’ve done, you can contact me at roel.van.os@humanitech.nl. For more information on the PF/Tijah system, you can take a look at the ‘old’ wiki[1] and the new documentation website[2].

Note to the reader This thesis has been written for readers with an affinity with computer science: the reader is expected to be familiar with software development in general and XML in particular. Knowledge of (relational) database technology is preferable, but not required: advanced concepts are introduced as necessary. Most if not all information retrieval concepts are fully introduced.

Chapter 1

Introduction

1.1 Background

In this thesis we report on some of the issues that were encountered while designing and implementing the PF/Tijah XML information retrieval system. This product combines into a single search system concepts from XML databases on one hand and structured information retrieval on the other. These two fields are briefly described below.

1.1.1 XML Database Technology and XML Information Retrieval

The ongoing adoption of XML as an interchange and storage format has resulted in the development of XML databases: systems designed to enable fast retrieval and manipulation of large volumes of data in XML format. Queries on this data specify exactly which pieces of data are to be retrieved, and how this data should be presented.

Some XML databases are built on top of more traditional relational databases, which provide a solid foundation of optimized and tested storage structures and algorithms. An example of one of these systems is the XML database developed by the Pathfinder project. The project aims to use the full potential of relational database technology to construct an efficient and scalable XQuery implementation\[7\]. The research team is composed of members from the Technische Universität München, CWI Amsterdam and the University of Twente Database group. The system developed by this project, the Pathfinder XML database (also known as MonetDB/XQuery), implements an XQuery processor on top of MonetDB, an open-source database system\[5\]. Pathfinder is mature enough to be used in application development. Research and development currently focuses on further improving query performance on large documents and supporting updates. The XQuery language used by this system is described more fully in section\[2,4\].

Having efficient XML databases is important, however these systems usually don’t effectively support full-text search: searching for objects in the database that satisfy a query on the textual contents and showing the most relevant results first. These queries are usually expressed as a set of search terms (keywords). This kind of searching is in the domain of information retrieval (IR). The central concept in IR is the ranking of pieces of information (e.g. documents) according to their estimated relevance to a user query. For example, most if not all web search engines present a ranking of web pages
1.1. BACKGROUND

that are relevant to the query: the page that is estimated by the search engine to be most relevant to the query is displayed at the top. **Structured information retrieval**, or in our case more specifically **XML information retrieval** (XML-IR), is a relatively recent development in the field of information retrieval. XML-IR focuses on the retrieval of semi-structured data in the form of XML documents: the structured nature of the documents is also taken into account when performing queries.

One of the research projects active in XML-IR is **CIRQUID**\(^1\), which aims to design and build a database management system that integrates relevance-oriented querying of semi-structured data (IR) with traditional querying of this data\(^1\). The research team is composed of members from the University of Twente Database group and CWI Amsterdam. The group has implemented their concepts and ideas in the **TII\(^1\)** system, using it to participate in a number of information retrieval evaluation initiatives such as TREC and INEX to test their theories. TIIAH is also built upon the MonetDB binary relational database kernel.

Both Pathfinder and TIIAH will be described in more detail in the following chapters.

1.1.2 PF/Tijah – Integrating Pathfinder and TIIAH

As stated before, this thesis describes aspects of the PF/Tijah system, which integrates a structured information retrieval system (TIIAH) into an XML database (Pathfinder). One significant disadvantage of the TIIAH XML-IR system that we wanted to solve by this integration was the fact that it had no standardized user interface: there was no way to manipulate the results of information retrieval queries using a well-defined language. This makes developing robust applications very difficult. By integrating TIIAH into Pathfinder, results of information retrieval queries can be be processed by a standardized XML database query language (**XQuery**). This is useful for both experimental purposes and application development. In addition, this creates an information retrieval system that benefits from optimized XML database techniques implemented in Pathfinder, among others: the staircase join algorithm and a fast and standard-compliant document shredder.

The integrated system – named **PF/Tijah** to demonstrate that it is an extension to Pathfinder – exposes IR querying functionality at the conceptual level as a set of XQuery functions that allow a predefined collection to be queried with NEXI queries. The results of these queries are returned as XQuery node sequences, ranked according to their relevance to the query. These sequences can then be processed using all the available XQuery facilities, such as axis steps, loops and functions. At the logical level, PF/Tijah reuses concepts and tools from the CIRQUID project: NEXI queries are translated into logical level algebraic query plans, defined in the Score Region Algebra (SRA)\(^2\). At the physical level, PF/Tijah adds to the existing Pathfinder data structures a full-text index that contains the necessary information to perform IR queries\(^1\). Sections \(2.2.2\) and \(2.2.3\) explain NEXI and SRA in more detail. The physical level index structure and IR primitive implementations are described in chapter \(3\).

Open research and implementation issues at the time of this writing include:

- Addition of documents to existing search collections is supported, however inserting XML fragments at arbitrary points in a collection is not.

\(^1\)The acronym CIRQUID stands for ‘Complex Information Retrieval Queries in a Database’.

\(^2\)Beside being almost a shibboleth, the name TIIAH is also an acronym, the meaning of which is a closely guarded secret.
– Advanced IR search techniques, such as proximity and phrase search and relevance feedback, are not supported. This is partially due to lack of expressiveness in the IR querying language used (NEXI), and also because we simply haven’t gotten around to implementing them in a user-friendly way, yet.

– At this time, IR querying is exposed to the user by custom XQuery functions. A nicer solution would be to use a standardized query language such as XQuery Full-Text (described in section 2.2.1).

Despite these issues, PF/Tijah is being used in favor of TJAH in several experiments, for example INEX 2006 (unpublished at this time).

1.2 Contribution and Goals

The contribution of this thesis is threefold.

Before and during the writing of this thesis, the design and implementation of the PF/Tijah XML-IR system took shape. We reused the conceptual and logical layers of the IR querying system almost without change from TJAH and integrated them with Pathfinder. However, we need to adapt the implementation of the physical level IR primitives to make use of the existing Pathfinder index and the new PF/Tijah IR index. While we were implementing the physical level IR primitives, we had the opportunity to take a good look at their efficiency. Part of this efficiency improvement was gained by using data structures and algorithms that are specially suited for this task. We report on the properties and use of these data structures and algorithms in this thesis.

Besides the use of optimized data structures and algorithms, database systems also attempt to improve efficiency by reducing intermediate result sizes while performing queries. By executing the most selective (and inexpensive) operators first, execution speed can be increased because subsequent operations have less data to process. This is a form of query rewriting. Since in PF/Tijah there is already a form of query rewriting in place at the logical level for this purpose, we will instead focus on steps that can be taken at the the physical level implementation of PF/Tijah’s IR primitives to reduce intermediate result sizes.

The retrieval model (score computation function) that was used most for retrieval experiments by the CIRQUID group is a probabilistic language model. A probability is assigned to each search element according to the chances that it satisfies the query. There is also a class of models that is based on logarithmic likelihood ratios. A feature of the particular likelihood model implemented on PF/Tijah (namely NLLR) is that it assigns a zero score to search elements that contain none of the query words. Since a zero score means that an element is not retrieved, this element can be left out of the result set. Besides this, these models have nice numerical properties: probabilistic models tend to generate very small scores because of successive multiplications of probabilities (which are, by definition, between zero and one). Logarithmic likelihood models on the other hand, do not have this limitation. In this thesis, we examine the application of the NLLR model in SRA and the influence on retrieval effectiveness and efficiency.

To sum up, besides describing the physical level implementation of the IR primitives, we will try to find an answer to the following research questions:
1.2. CONTRIBUTION AND GOALS

1. What steps can be taken at the physical level of an XML-IR system based on SRA to reduce intermediate result sizes during IR query execution?
   
   For these steps we would like to know:
   
   (a) what is the (expected and actual) effect on memory usage and execution speed?
   
   (b) what is the (expected and actual) effect on retrieval effectiveness?

2. What is the effect of introducing retrieval models that are not based on probabilities but on logarithmic likelihoods?
   
   (a) What is the effect on retrieval effectiveness?
   
   (b) Can these models be used to reduce intermediate result sizes?
   
   (c) How does this influence combination and propagation?

In chapter 2, we describe the research area, concepts and systems that support the work reported in this thesis. In chapter 3, we examine the physical level implementation of the Score Region Algebra, including an approach that aims to reduce intermediate result sizes at the physical level.

Any change to the physical level implementation of IR operators might have consequences that fall into two categories. The first category is efficiency, e.g. how fast a query can be executed, while keeping memory usage as small as possible. The second category is retrieval effectiveness (also called precision), in this case defined by the quality of the result rankings.

The aspects of efficiency – in our case we look at execution time and memory usage – can be measured relatively easily and unambiguously. Measuring retrieval effectiveness on the other hand is not so simple. Effectiveness depends very much on what the user thinks of the result ranking, how many relevant and irrelevant elements it contains. For true scientific evaluation, we need a reproducible, automatic means of determining the quality of result rankings so that they can be compared for effectiveness. Information retrieval evaluation initiatives such as TREC and INEX have developed methodologies to evaluate the effectiveness of search systems.

In this thesis, we use the methodology defined by INEX, specifically the one used for the 2004 and 2005 conferences[23, 22]. In chapter 4, we describe the experiments we performed to validate our approach to intermediate result size reduction. We also report the experimental results in this chapter. Finally, in chapter 5, we examine the experimental results to see how the approaches proposed in chapter 3 fulfill the goals set out here.
Chapter 2

XML Retrieval – Concepts, Languages and Systems

In this chapter, an overview is presented of concepts, systems and research efforts from the fields XML-DB and XML-IR.

2.1 XML and XML Databases

The interchange of information has always been an interesting problem in computer science. The challenge lies in creating a format for messages to be interchanged that is both concise and adaptable to the need of the application, without being expensive in terms of processing power and code complexity. Also, a format that can be written and interpreted by humans has a clear advantage for software development. The Extensible Markup Language (XML) was designed to meet these requirements.

XML is a meta-language: a language to express other languages in. The XML standard prescribes a method of encoding structured information in an extensible text-based format. Being text-based, it is both writable and readable by humans, without the use of specialized (binary) editors. Its extensibility comes from the ability to define custom markup languages to fit the application domain. The standard only prescribes a small set of syntactical requirements, but leaves the semantics of the format open to the users, whereas for example HTML prescribes both syntax and semantics. With HTML, which is a language designed for hypertext markup, application is limited to creating web pages. XML was designed to be a universal language to create interchange languages. For example, the XML version of HTML: XHTML.

XML is the successor to the Standard Generalized Markup Language (SGML). This language has been in use for some time, mainly for document markup (e.g. newspapers). XML was designed to be a simpler form of SGML, while at the same time being compatible with SGML.

With the adoption of XML, the need arose to store and query volumes of XML data, in the same manner that relational databases are used. To this end, several XML query languages have been designed. The XML Path Language (XPath) is a language to address parts of an XML document. This language has been designed to be used by e.g. the Extensible Stylesheet Language Transformations
(XSLT)\(^{[8]}\), which allows an XML document to be transformed into another XML document. Both of these languages are W3C recommendations.

The XQuery language\(^{[4]}\), a W3C proposed recommendation, combines the expression of axis steps as specified by the XPath language with a functional, side-effect free language, which allows XML to be retrieved and manipulated. An example XQuery expression displaying these elements might be specified as follows:

```
(: This is a comment :) 

(: Bind a document to a variable :) 
let $doc := fn:doc('http://www.example.com/test.xml')

(: Bind a set of articles inside that document to a variable using an axis step that selects all elements called article :) 
let $articles := $doc//article

(: Iterate over the articles :) 
for $article in $articles 
return <article-info>
  <title>{$article//title}</title>
  <article-id>{$article/@id}</article-id>
</article-info>
```

This query retrieves an XML document from a website and returns every article title and its unique identifier (id).

Several XQuery processors exist at the moment. The Pathfinder research project is working to find an answer to the question: ‘How far can we push relational database technology to construct an efficient and scalable XQuery implementation?’\(^{[7]}\). To this end, an XQuery processor has been implemented on top of the MonetDB database system\(^{[5]}\). The project, named MonetDB/XQuery, is open source, is actively developed and has regular stable releases. Its stable release at the time of this writing supports a large part of the XQuery recommendation, an extension for Burkowski standoff annotations\(^{[1]}\) and facilities for updates\(^{[6]}\).

Galax is an open-source implementation of XQuery, developed specifically to be the reference implementation of this recommendation\(^{[11]}\). Several authors of the XQuery recommendation are on the team behind Galax. It is therefore one of the most complete and standard-compliant XQuery processors available. This engine is mentioned here because it has extensions for full-text search (GalaTex, see section 2.2.1). Galax is designed to be independent of specific database engines or application areas.

There are many more XQuery implementations available; the \[W3X Query web page\(^{[4]}\] lists many systems, both open-source and commercial.

\(^{[1]}\) http://www.w3.org/XML/Query/
2.2 (Structured) Information Retrieval

Wikipedia defines *information retrieval* as follows:

> Information retrieval (IR) is the science of searching for information in documents, searching for documents themselves, searching for metadata which describe documents, or searching within databases, whether relational stand-alone databases or hypertext networked databases such as the Internet or intranets, for text, sound, images or data.

It is an interdisciplinary field, cutting across computer science, library science, psychology, linguistics and statistics. There are many information retrieval systems available, ranging from library retrieval systems to web search engines (e.g. Google). One of the most influential evaluation initiatives where aspects of IR are researched is the Text REtrieval Conference (TREC).

A relatively recent development is the combination of structural retrieval with information or content retrieval. This development is the result of the adoption of first SGML and later XML as an archival format for large volumes of data, e.g. the entire corpus of a newspaper. Once such a corpus has been collected, it must be searched effectively. Both traditional IR and XML-DB fall short of this goal. Traditional IR does not take into account the structured nature of the documents in the collection. In traditional IR systems, the search area and return elements are predefined: only documents as a whole can be searched and returned. On the other hand, XML-DB query languages do not natively have the concept of ranking elements according to their relevance to a set of query terms.

In recent years, this has produced the field of structured information retrieval, also called XML Information Retrieval (XML-IR), since in most cases XML provides the structure. This field combines methods from XML-DB (i.e. structural retrieval) and information retrieval (i.e. content retrieval). The emerging standard language for structured IR systems is arguably *XQuery Full-Text*. This language is described in the next subsection. For research in structured information retrieval, the NITiative for the Evaluation of XML Retrieval (INEX) is one of the evaluation initiatives where ideas and systems for this are evaluated and discussed. A simple XML-IR querying language was designed for this initiative: *NEXI*, which we describe in subsection 2.2.2. Finally, the *Score Region Algebra*, an algebra for expressing structured information retrieval queries at the logical level between the user query and the physical level implementation is examined in detail in subsection 2.2.3. We examine NEXI and SRA in this level of detail since it is the foundation of PF/Tijah and the research done in this thesis.

2.2.1 XQuery Full-Text

Since the XQuery language is becoming the XML querying language of choice, it is the logical starting point for designing and implementing an XML-IR querying language. The XQuery Full-Text proposal, currently being developed by the W3C, is precisely that: an extension of the XQuery language to support full text searching. The XQuery-FT proposal is based, with some modifications, on the TeXQuery language, proposed in [3]. XQuery-FT has a reference implementation on top of the Galax XQuery engine, called GalaTex[10]. Both of these systems are open source.

XQuery-FT consists of a number of extensions to the XQuery language, demonstrated in the example below. This example was taken from [2], but rewritten slightly for readability.

```xml
(: Load a document from an URL and select all books in this document :)
```
In this example, a sequence of book elements is searched for usability. Every book that has a non-zero score is returned, in order of relevance to the term usability. Note that in this example, the order in which the books are processed is dependent on the score value, i.e. scores and book elements are sorted side-by-side.

The XQuery-FT proposal expresses semantics in terms of normal XQuery, so a modification of the XQuery data model is not necessary. This was one of the requirements, since otherwise all existing XQuery functions and expressions would have to be modified. Besides this, the language is also designed in such a way that queries can be checked for syntax and type correctness without actually running the query (static type checking). There are no ‘black box’ query strings.

XQuery-FT serves the same function as SQL does in relational database technology: it is a language between the front-end application and the back-end database (in this case, an XML database). Because of the complexity of XQuery-FT, it is not suitable for research systems, which are mostly constructed by the researchers themselves in limited time. The NEXI language, described in the next subsection, is designed specifically for such situations.

2.2.2 Narrowed Extended XPath I (NEXI)

Narrowed Extended XPath I (NEXI) was developed for the INitiative for the Evaluation of XML Retrieval (INEX). The example queries (topics) that are used to evaluate systems used by the participants are expressed in NEXI, alongside a plain-text description. NEXI is an extended version of a subset of the XPath Query language. NEXI is a subset of XPath because it only supports a small part of the XPath specification. NEXI extends XPath with an about function, that enables the expression of IR queries. The language contains just enough features to be able to express ‘interesting’ IR queries, while being small enough to be implementable by researchers.

Because NEXI has been used primarily for research in the field of XML-IR in general and INEX
in particular, implementations of this language are mostly research systems. This also explains why NEXI is as limited as it is: it does not contain primitives for e.g. loops and element construction, since these are not a focus of research in this field.

The language distinguishes two types of queries: one can express content-only (CO) and content-and-structure (CAS) queries. Content-only queries (CO) are simply query words, possibly prefixed with + and − symbols. These symbols indicate the importance of the word to the query: + indicates that the word is very important, − indicates that documents containing that word are not relevant. This is how queries are defined for most internet search engines. The following example CO query was taken from the INEX topics:

```
Internet web page +prefetching algorithms −CPU −memory −disk
```

The query requests elements describing algorithms for pre-fetching of web pages by the client. Text about CPU, memory, disk is not relevant.

CO queries assume that the system will determine the search and return elements. In contrast, content-and-structure (CAS) queries allow both of these to be specified by the user. CAS queries are composed using the following elements:

- Path specifications using the descendant axis step (//): this is used to select elements that should be searched and elements that should be returned. This feature has been taken from XPath. Whereas XPath supports a wide variety of axis steps (child, parent, preceding-sibling, etc.), NEXI only supports the descendant axis.

- The about function, including `and` and `or` operators: using the about function, a set of elements can be tested on whether they contain a set of query terms (essentially, this set of terms is a CO query).

An example CAS query, combining axis steps and the about function, also taken from the INEX topics:

```
//article[about(.//abs,classification)]//sec[about(.,experiment compare)]
```

The query requests sections about experiment and compare, that are contained by articles that have an abstract that is about classification. The abs and sec elements are search elements, because they are searched by an about expression. The sec elements are also answer elements: since they are the last elements to be specified in the query path, they are returned to the user.

An example of a system using NEXI as a query language is the TIJAH research system. TIJAH is a set of tools that enables a researcher to investigate aspects of the IR search process. It is not a system that is usable for production work, since it does not automate the entire query process. However, TIJAH does embody some interesting concepts, such as the use of a relational database system as an implementation platform and the use of a three-level architecture. These aspects are described more fully in the next section.

The PF/Tijah system adds IR querying possibilities to the Pathfinder XQuery processor by exposing TIJAH functionality through XQuery functions. It therefore inherits characteristics of both Pathfinder and TIJAH. This enables a researcher to run structured IR queries (using TIJAH concepts) and post-process the results of these IR queries using XQuery.
2.2.3 Score Region Algebra

The CIRQUID research group proposed to create an architecture for their XML-IR system that follows the general design of existing relational database systems. The main feature of relational database design is the strong separation between conceptual, logical and physical levels. At the conceptual layer the database deals with the user query. This query is expressed in e.g. SQL in the case of relational databases. In an XML-IR, the query could be expressed in NEXI. The user query is translated to a logical level algebra expression, which can be rewritten for more efficient computation. Finally, the optimized logical level expression is translated to physical level algebra, which takes care of the actual data manipulation and storage management.

A consequence of the three-layered approach is that the physical level is free to implement its own optimized storage structures, as long as the physical operators behave according to the rules specified by the logical operator definitions. For example, at the physical level, a pre-size encoding can be used, while the logical algebra is defined over a pre-post encoding.

The logical level algebra proposed by CIRQUID is the Score Region Algebra[24]. This is an algebra that is similar to relational algebra, however it takes into account the structure of XML documents. Also, operators have been added to compute scores based on the occurrence of query terms in search elements, and operators to pass these scores to retrieval elements.

In SRA, documents are represented as regions containing other regions. In effect, an XML element is a region, which can contain other regions (elements or text nodes). In XPath parlance, containment expresses the descendant and ancestor axes. At the moment, SRA does not support any other axes (such as preceding and following). This is not yet necessary since the conceptual language (NEXI) does not support these, but adding these axes should not be difficult.

Because in the rest of this thesis we closely examine the physical level implementation of the SRA operators in PF/Tijah, the following subsections describe the logical level SRA in more detail. These subsections cite formal definitions and their interpretations from [24].

Data model – Regions and Region Sets

Regions are defined as tuples \((s, e, n, t, p)\) containing five attributes:

- \(s\): the start position of the region;
- \(e\): the end position of the region. It is always greater than or equal to the start position.
- \(n\): the name of the region. For a node, this is the node name; for a term, this is the term itself.
- \(t\): the type of the region. XML elements are of type node, terms (words) are of type term.
- \(p\): the score of the region. This score represents the relevance of this region with respect to the current query.

Region sets are sets of region tuples, defined as elements from the region powerset \(P(R) = \{R' | R' \subseteq R\}\) (all possible subsets of region \(R\)). \(C\) is used to denote the collection region set: this is the region set that contains all regions that a query can access.
In order for an XML fragment to be manipulated by SRA, it must be expressed in terms of regions. Take the following document for example:

```xml
<thesis>
  <title>XML Information Retrieval</title>
  <section>
    <title>XML</title>
  </section>
</thesis>
```

![Figure 2.1: Document position assignments](image)

Every element and term in this document is assigned a starting and ending position (see figure 2.1). The region set $R$ for this document would then consist of the following regions:

$$R = \{(s = 1, e = 12, n = \text{thesis}, t = \text{node}, p = 1) \}
\quad \text{(2.1)}$$

A number of these documents can be combined to form a search collection $C$.

Several operators have been defined over the region tuples. These operators are described in the next subsections.

**Selection**

The selection operator ($\sigma$) is formally defined as follows:

$$\sigma_{n=\text{name}, t=\text{type}}(R) = \{ r | r \in R \land r.n = \text{name} \land r.t = \text{type} \}$$

This operator selects all regions $r$ from region set $R$ that have the indicated name and type. It can be used to filter term or element (node) regions from a region set – this region set can be the universal collection region set $C$, to select all occurrences in the collection.
2.2. (STRUCTURED) INFORMATION RETRIEVAL

Containment Relation Computation

A NEXI query can start with a path to the first search element:

//article//sec[about(.,databases)]

This query specifies that the regions to be searched and returned are sec nodes that are descendants of article nodes. This would be expressed in SRA like this:

\((\sigma_n=\text{sec}, t=\text{node}) \sqsupseteq (\sigma_n=\text{article}, t=\text{node}) \sqsubseteq \sigma_n=\text{databases}, t=\text{node})\)

Containment relations between two region sets are defined by the containing (\(\sqsupseteq\)) and contained-by (\(\sqsubseteq\)) operators as follows:

\[ R_1 \sqsupseteq R_2 = \{ r_1 | r_1 \in R_1 \land \exists r_2 \in R_2 \land r_2 \preceq r_1 \} \]  \hspace{1cm} (2.2)

Every region \(r_1\) from \(R_1\) that contains at least one region \(r_2\) from \(R_2\) is placed in the result set.

\[ R_1 \sqsubseteq R_2 = \{ r_1 | r_1 \in R_1 \land \exists r_2 \in R_2 \land r_1 \preceq r_2 \} \]  \hspace{1cm} (2.3)

Every region \(r_1\) from \(R_1\) that is contained by at least one region \(r_2\) from \(R_2\) is placed in the result set. The \(\preceq\) symbol has been used to express the containment relation between two regions: if \(r_j \preceq r_i\) that means that \(r_i\) is contained by \(r_j\):

\[ r_i \preceq r_j \iff r_j.s \leq r_i.s < r_i.e < r_j.e \]  \hspace{1cm} (2.4)

Note that these containment operators do not change scores that may already have been associated with regions. Probabilistic containment or score computation operators are explained in the next subsection.

Score Computation

At some point in query processing, the relevance of a search element with respect to a query term or set of query terms has to be determined. At the logical level, score computation is performed by the probabilistic containment operator (\(\sqsupseteq_p\)), formally defined as follows:

\[ R_1 \sqsupseteq_p R_2 = \{ r | r_1 \in R_1 \land (r_1.s, r_1.e, r_1.n, r_1.t, f\sqsupseteq_p(r_1, R_2)) \land t_1 = \text{node} \land t_2 = \text{term} \} \]  \hspace{1cm} (2.5)

All regions in \(R_1\) are assigned scores based on their relevance to the terms in \(R_2\). The actual score computation is delegated to an auxiliary function \((f\sqsupseteq)\). In section 3.2.6 two retrieval models (i.e. \(f\sqsupseteq\) implementations) are examined in detail. For the sake of brevity, we use the term ‘computation operator’ in this thesis instead of ‘probabilistic containment operator’. Another term that is used frequently (but not in this thesis) is ‘scoring operator’.

Using the computation operator, when a region set has to be scored for relevance for multiple search terms, relevance has to be determined for each of the terms separately, after which the different relevances have to be combined into a single score for each region. [24] suggests a complex (or ‘coarse’)}
function \( f \)

For the definition of the ‘plain’ union operator is an intersection of regions

and

For the article a single score for each search element (The scores that have been computed for these separate

using the probabilistic

\( \alpha \) selection operator

18

CHAPTER 2. XML RETRIEVAL – CONCEPTS, LANGUAGES AND SYSTEMS

According to the textual explanation in [24], the or operator \((\sqcup_p)\) computes a union of two regions:

\( R_1 \sqcup_p R_2 \)

is an intersection of regions \( R_1 \) and \( R_2 \). The scores in the resulting set are determined by an auxiliary function \((\oplus, \text{ e.g. } r_1.p \cdot r_2.p)\).

For the and operator \((\sqcap_p)\), a region is placed in the result set if it present in both \( R_1 \) and \( R_2 \): \( R_1 \sqcap_p R_2 \)

is an intersection of regions \( R_1 \) and \( R_2 \). The scores in the resulting set are determined by an auxiliary function \((\otimes, \text{ e.g. } r_1.p \cdot r_2.p)\).

The scores that have been computed for these separate about clauses have to be combined to form

a single score for each search element (article). In the logical algebra, score combination is done using the probabilistic and \((\sqcap_p)\) and or \((\sqcup_p)\) operators, respectively:

\[
R_1 \sqcap_p R_2 = \{(r.s, r.e, r.n, r.t, r_1.p \otimes r_2.p) | r \in R_1 \land r_2 \in R_2 \land (r.s, r.e, r.n, r.t) = (r_1.s, r_1.e, r_1.n, r_1.t) \}
\]

(2.7)

\[
R_1 \sqcup_p R_2 = \{(r.s, r.e, r.n, r.t, r_1.p \oplus r_2.p) | r \in R_1 \land r \in R_2 \land ((r.s, r.e, r.n, r.t) = (r_1.s, r_1.e, r_1.n, r_1.t) \lor (r.s, r.e, r.n, r.t) = (r_2.s, r_2.e, r_2.n, r_2.t)) \}
\]

(2.8)

For the and operator \((\sqcap_p)\), a region is placed in the result set if it present in both \( R_1 \) and \( R_2 \): \( R_1 \sqcap_p R_2 \)

is an intersection of regions \( R_1 \) and \( R_2 \). The scores in the resulting set are determined by an auxiliary function \((\otimes, \text{ e.g. } r_1.p \cdot r_2.p)\).

According to the textual explanation in [24], the or operator \((\sqcup_p)\) computes a union of two regions:

\( R_1 \sqcup_p R_2 \)

contains regions that are present in either \( R_1 \) or \( R_2 \), in addition to regions that are present in both \( R_1 \) and \( R_2 \). For regions that are present in both \( R_1 \) and \( R_2 \), the score is determined by an auxiliary function \((\oplus, \text{ e.g. } r_1.p + r_2.p)\). Scores of regions that are only present in one of the sets are left unchanged. The formal definition given in [24] and cited above does not seem to express this. We assert therefore that the or operator \((\sqcup_p)\) should be defined as follows:

\[
R_1 \sqcup_p R_2 = \{(r.s, r.e, r.n, r.t, r_1.p \oplus r_2.p) | r \in R_1 \land r \in R_2 \land (r.s, r.e, r.n, r.t) = (r_1.s, r_1.e, r_1.n, r_1.t) \} \cup \{(r_1.r \in R_1 \land \exists r_2 \in R_2 \bullet (r.s, r.e, r.n, r.t) = (r_1.s, r_1.e, r_1.n, r_1.t) \} \cup \{(r_1.r \in R_2 \land \exists r_2 \in R_1 \bullet (r.s, r.e, r.n, r.t) = (r_2.s, r_2.e, r_2.n, r_2.t) \}
\]

(2.9)

For the definition of the ‘plain’ union operator \(\sqcup\), see [24], page 73.
Score Propagation

There are two kinds of score propagation operators: upward and downward propagation. Upward propagation (\(\rightarrow\)) is used to propagate scores from search elements inside about clauses to their context elements. For example:

\[
\text{propagation} \quad \text{score computation} \\
\text{//article[about(.//sec, information retrieval)]}
\]

In this example, scores are computed for the sec elements. These scores are propagated upward to the article elements. It might be that multiple sec elements share a common ancestor article element. In this case, scores from these sec elements must be combined into a single score for their common ancestor.

Downward propagation (\(\leftarrow\)) is used in the following example:

\[
\text{score computation} \quad \text{propagation} \\
\text{//article[about(.,information retrieval)]//sec[about(.,progressive indexing)]}
\]

Scores are computed for article elements. Those scores must be taken into account when computing scores for the sec elements: a sec element inside a relevant article must receive a higher score than a sec element inside an irrelevant article. Therefore the scores from the article elements are propagated to the sec elements. To generalize: sec elements that have more relevant ancestors should be considered more relevant than sec elements that have fewer relevant ancestors.

The two propagation operators are formally defined as follows. The upward propagation operator (\(\rightarrow\)) defines the propagation of scores to containing elements (from descendants to ancestors):

\[
R_1 \rightarrow R_2 = \{(r_1.s, r_1.e, r_1.n, r_1.t, f_\rightarrow(r_1, R_2)) | r_1 \in R_1 \land r_1.t = \text{node}\} \tag{2.10}
\]

The downward propagation operator (\(\leftarrow\)) defines the propagation of scores to contained elements (from ancestors to descendants):

\[
R_1 \leftarrow R_2 = \{(r_1.s, r_1.e, r_1.n, r_1.t, f_\leftarrow(r_1, R_2)) | r_1 \in R_1 \land r_1.t = \text{node}\} \tag{2.11}
\]

These definitions are again dependant on auxiliary functions for score computation (\(f_\rightarrow\) and \(f_\leftarrow\)).

Translating NEXI Queries to SRA

The function of the operators described in the previous subsections is illustrated using the following example:

\[
\text{//article[about(.,//abs, classification)]//sec[about(., experiment compare)]}
\]

This query expresses the following information need:
Find all sections about experiment and compare that are contained in articles that contain an abstract about classification.

Conceptually, this query should be interpreted as follows:

- Starting at the collection root, select all article elements. Then, select all abs elements that are contained in an article element. Rank these abs elements according to the occurrence of the query term classification. This ranking should affect the ranking of the article elements.

- Next, select all sec elements that are contained in an article element. These sec elements should be ranked according to the current ranking of the article elements. Finally, rank these sec elements according to the occurrence of the query terms experiment and compare, keeping in mind the already existing ranking of sec elements.

Using this interpretation, the NEXI query can be translated into an SRA tree. This translation maps containment steps on containment operators (⊆ and ⊇), about expressions on the probabilistic containment operator (⊑p or ⊑, see below), and and or combinations of about expressions are mapped on the combination operators (∩p and ⊔p, respectively). Note that this query does not contain explicit and or combinations, however when an about expression contains multiple query terms, scores for these different query terms must be combined into one score for each region. We call this an implicit combination. In the following example and in the rest of this thesis this is done using an and combination.

The expression in an SRA tree of the example above is given in figure 2.2. The SRA expression can also be written in a ‘procedural’ fashion:

\[
\begin{align*}
\text{article} & := \sigma_{n=\text{article},t=\text{node}}(C) \\
\text{abs} & := \sigma_{n=\text{abs},t=\text{node}}(C) \\
\text{sec} & := \sigma_{n=\text{sec},t=\text{node}}(C)
\end{align*}
\]
2.2. (STRUCTURED) INFORMATION RETRIEVAL

Figure 2.3: Score Region Algebra tree, using the complex selection and computation operator ($\alpha$)

Steps 2.12 through 2.14 select the required node regions from the collection; steps 2.15 through 2.17 select the required term regions. Step 2.18 ranks abs elements according to their relevance to the term classification. Step 2.19 propagates scores from the abs elements to the article elements that contain them.

Steps 2.20 through 2.22 rank sec elements according to their relevance to the terms experiment and compare. Finally, in step 2.23 the scores from the article elements are propagated to the sec elements. In this propagation, scores already present on the sec elements are of course taken into account. $R_6$ is the resulting ranking for this query.

Using the complex selection and scoring operator ($\alpha$), the same query can be expressed in a tree as show in figure 2.3 and procedurally as follows:

classification : = $\sigma_{n=\text{classification},t=\text{term}}(C)$ (2.15)
experiment : = $\sigma_{n=\text{experiment},t=\text{term}}(C)$ (2.16)
compare : = $\sigma_{n=\text{compare},t=\text{term}}(C)$ (2.17)

$R_1 := \text{abs } \sqsupset_p \text{classification}$ (2.18)
$R_2 := \text{article } \blacktriangledown R_1$ (2.19)

$R_3 := \text{sec } \sqsupset_p \text{experiment}$ (2.20)
$R_4 := \text{sec } \sqsupset_p \text{compare}$ (2.21)
$R_5 := R_3 \sqcap_p R_4$ (2.22)

$R_6 := R_5 \blacktriangledown R_2$ (2.23)

article : = $\sigma_{n=\text{article},t=\text{node}}(C)$ (2.24)

$R_1 := \alpha_{n=\text{classification}}(C)$ (2.25)
$R_2 := \text{article } \blacktriangledown R_1$ (2.26)

$R_3 := \alpha_{n=\text{experiment.compare}}(C)$ (2.27)
\[ R_4 := R_3 \upharpoonright R_2 \] (2.28)

Notice that \( \alpha \) operator actually performs the work of at least three simpler SRA operators: it has to perform element and term selection (\( \sigma \)), score computation (\( \sqsupseteq_p \)) and score combination (\( \sqcap_p \) or \( \sqcup_p \)) to combine scores from different terms. In section 3.2.6 we elaborate on the advantages and disadvantages of using the complex computation operator (\( \alpha \)).

### 2.2.4 Other Approaches to Structured Information Retrieval

The approaches outlined in the previous sections – XQuery Full-Text, NEXI and SRA – are of course only a fraction of the offering of structured IR systems and concepts.

A widely used method of implementing structured document retrieval is fielded retrieval. This method is based on the grouping documents into collections. The user or administrator specifies which parts – fields – of these documents can be searched. The system then builds indexes for these fields to enable fast searching. At querying time, the user can specifies which fields are to be searched to determine the relevance of a document to a query. The system can then return the field in question or the entire document that contains it. Through the use of the specialized indexes, field-based retrieval systems generally achieve good performance, at the cost of some flexibility: the user cannot search arbitrary parts of the document, only the ones indexed by the system.

A very mature fielded IR framework is the Lemur system[27, 28]. It is designed to facilitate building search systems, both for research and production use. Lemur provides several retrieval models, such as Language Models and Okapi. It does have a predefined ‘document’ concept, but supports structured information retrieval through field/passage retrieval.

Indri (part of the Lemur project) focuses on providing a search engine using Lemur principles and technology. Because Lemur and Indri are targetted at implementing search systems, they have a well-documented API. Lemur and Indri do not have a general-purpose post-processing facility (like XQuery in PF/Tijah) to provide results in a certain format, however the system can be easily extended using the API in C++, Java and PHP.

### 2.3 Conclusion

This chapter has presented a high-level overview of existing XML-IR query languages, concepts and systems. The Score Region Algebra has been introduced. The next chapter takes a look at SRA in more detail, especially at the decisions that can be made in the physical level implementation.
Chapter 3

SRA at the Physical Level

In the previous chapter, we examined the Score Region Algebra in detail. So far, SRA has been implemented in two research systems: TIJAH and its successor PF/Tijah. For more information on TIJAH, refer to [26, 24]. [18] summarizes the design of PF/Tijah; this design is also described in more detail below, starting with its foundation: the Pathfinder XML database and the MonetDB database kernel.

Pathfinder and MonetDB The Pathfinder XQuery processor, described in the previous chapter as an example of an XML database (section 2.1), is built as a front-end to the MonetDB relational database kernel. MonetDB was designed to serve as a highly optimized relational back-end for query-intensive applications. To create for example a relational database management system one can use MonetDB as the physical level back-end and add a front-end that translates SQL queries to MonetDB relational algebra. At the time of this writing, besides the Pathfinder XQuery front-end there is an SQL front-end available.

MonetDB uses a data model based on full vertical fragmentation into binary association tables (BATs). For example a table seen from SQL with four columns is actually stored as four separate binary relations, one BAT for each column. Each binary table stores the row identifier (or object identifier, oid) in the head column and the column value in the tail column. This fragmentation makes processing of queries that access only some of the columns – a pattern frequently seen in query-intensive application – much easier to optimize, since only the needed columns have to be loaded from disk. In addition, the head column storing the object identifier (oid) can be made ‘virtual’ (void) since it is identical in each of the vertical fragments. A void column is a dense ascending column, starting at a certain offset. Because it is dense (i.e. there are no gaps) only the offset needs to be stored, resulting in significant reduction of memory and disk space usage. A table with a void head column is in effect a single array of values, supporting constant-time positional lookups.

The Monet Interpreter Language (MIL) is the language that interfaces the MonetDB back-end with front-ends. Relational primitives that MonetDB implements are exposed as MIL functions. Front-ends such as SQL and XQuery processors consume SQL and XQuery expressions and produce MIL code for MonetDB to execute. MonetDB can also be extended with modules programmed in C to provide data structures and algorithms optimized for specific application areas (such as XML processing). These structures and algorithms can then be used from MIL code. For more information on the design
and implementation of MonetDB, the reader is referred to [5].

**PF/Tijah** The PF/Tijah structured information retrieval system is an extension of the Pathfinder XML database. At Pathfinder’s conceptual level, we have added a number of XQuery functions that allow execution of NEXI queries on a predefined document collection. At Pathfinder’s physical level we have added a NEXI-to-MIL compiler and an index structure (a set of tables) that supplements the Pathfinder data structures. The SRA operators are implemented on top of this combined index. A lot of the ideas and code have been reused from the TIIAH system.

![PF/Tijah Architecture Diagram](image)

Figure 3.1: PF/Tijah architecture

Figure 3.1 shows the architecture of PF/Tijah. Besides the additional XQuery functions at the physical level (shown here by ‘XQuery + NEXI query’) we have implemented PF/Tijah as a self-contained module that extends Pathfinder at the physical level. When an XQuery + NEXI query is passed to Pathfinder, this query is first converted to XQuery Core. From that representation, a syntax tree is
3.1. THE PF/TIJAH INDEX STRUCTURE

built, from which relational algebra (MIL code) is produced directly. A logical algebra for Pathfinder is under development at the time of this writing; PF/Tijah does not support this at the moment.

When the relational algebra produced by the Pathfinder compiler is executed, the NEXI queries embedded in the XQuery query are forwarded to the PF/Tijah module. These NEXI queries pass through the three layers of the PF/Tijah module. The conceptual level parses and rewrites the queries and performs stemming and stop word removal. The queries are then translated into a logical level SRA expression. This expression is rewritten for faster execution and converted to a query plan in physical level relational algebra (again, MIL code). The results of the execution of this query plan are passed back to Pathfinder, which continues processing the XQuery part of the query.

At the physical level, Pathfinder and PF/Tijah each use their own data structures (tables) to enable fast computation their respective primitives. For example, Pathfinder has a table that stores the size of each element, to be used by the staircase join algorithm[14] to compute the results of axis steps. PF/Tijah adds its own tables to the information already present in the Pathfinder tables.

In the following section we describe the PF/Tijah index structure. In the subsequent section we discuss the implementation of the SRA primitives on top of this index. In addition, we analyze the implementation to see how the size of intermediate results can be reduced.

3.1 The PF/Tijah Index Structure

XML data that PF/Tijah can query is placed into collections: one or more XML documents that are treated as a single XML instance by the SRA operators. SRA computation is always confined to a single collection, because otherwise concepts such as background statistics (how many times a word occurs in the collection) are meaningless. It is of course possible to create multiple collections, which can then be accessed by separate NEXI queries.

As stated before in subsection 2.2.3 SRA considers XML data to be structured into regions. Regions have a type attribute. At the time of this writing, only regions of type node and term are stored in the PF/Tijah index. In the following discussion, the words node, (XML) element and tag are considered to be equivalent. Similarly, word and term are used interchangeably.

To support fast and efficient execution of SRA query plans on the MonetDB relational back-end, we have created an index structure to supplement the Pathfinder index[1]. Pathfinder already creates an index for every document that it processes. This index however does not contain enough information to be able to efficiently execute SRA queries. The main issue is that Pathfinder stores pieces of text (character data) in XML documents as it parses them without splitting them up into individual words, while an efficient implementation of SRA requires that the occurrences of individual words can be easily and quickly retrieved.

To make SRA execution straightforward and efficient to implement, we have identified the following operations that the index should support:

- **Lookup of element and term regions by their name**: this is needed for an efficient selection operator (σ) implementation.

---

1The word index can mean many things to many people. In this context (IR) it is a set of data structures that stores the information that is to be queried, in this case a set of XML documents.
To support this we have implemented table structures and an algorithm that are optimized for lookup of the occurrences of sets of element and term identifiers. We use the collection position to uniquely identify element and term regions (occurrences), so a lookup of element names and terms results in a list of collection positions.

- **Computation of containment relations between regions:** this is necessary for both the containment operators ( \( \sqsubseteq \) and \( \sqsupseteq \)) and the retrieval model implementations (i.e. any operator that uses the containment function (\( \prec \))).

We use the staircase join algorithm already present in Pathfinder to efficiently determine containment relations. The staircase join requires *document order* and *size* information for each region (terms and elements). Document order is determined by the collection positions, which are found at selection time using the selection operator. The size information we have stored in a table that stores the sizes of both element and term regions.

- **Lookup of element region sizes:** retrieval model implementations use size information to compute score values for element regions.

Since the staircase join already needs a table that stores size information for all regions, element region sizes can simply be found in this table.

The tables and algorithms are described more fully in the following two subsections.

### 3.1.1 Tables in the PF/Tijah Index

Table 3.1 lists the tables that comprise the physical level information store (index) of PF/Tijah. Tables are sorted by the column given in bold in the type column. The PF/Tijah index is divided into a collection-independent part and a collection-specific part. The collection-independent (global) tables \( t_j \).globalTags and \( t_j \).globalTerms store every unique element (tag) name or word in every collection. These tables only store the *names*; the positions are stored elsewhere (see below). The head values of these table are used as element or term identifiers (tids): these tables are used to look up the element identifier for a given element name or word.

The collection-specific tables store element and term positions for each collection separately. The \( t_j \).PFX.Tags and \( t_j \).PFX.Terms tables store the collection positions (i.e. starting positions, see subsection 2.2.3) of all element and word regions, respectively. The string PFX in these names is the name of the collection that is being queried. There is no direct association between element and term identifiers on the one hand and their collection positions on the other: the element and term tables have to be accessed through their corresponding Index tables. The \( t_j \).PFX.TermIndex table stores for each term identifier the *offset* into the \( t_j \).PFX.Terms table where the collection positions for that term can be found. The exact algorithm for these lookups is described in the next subsection. Finally, the \( t_j \).PFXDEFINED table is used to relate PF/Tijah element regions to Pathfinder elements and vice versa. A complete description of how this is done is outside the scope of this thesis.

Notice that there are significant differences between the logical level definition of the SRA data model (i.e. region attributes) presented in subsection 2.2.3 and the information actually stored in the index:

- Starting positions (s) are present as collection positions.
3.1. THE PF/TIJAH INDEX STRUCTURE

<table>
<thead>
<tr>
<th>Table name</th>
<th>Monet BAT type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tj.globalTags</td>
<td>oid → str</td>
<td>All tag (element) names in all collections</td>
</tr>
<tr>
<td>tj.globalTerms</td>
<td>oid → str</td>
<td>All words in all collections</td>
</tr>
<tr>
<td>tj.PFX_TagIndex</td>
<td>void → oid</td>
<td>Maps element identifiers to offsets into tj.PFX.Tags.</td>
</tr>
<tr>
<td>tj.PFX_Tags</td>
<td>void → oid</td>
<td>Element (tag) positions. Sorted by element id (not stored), then by collection position.</td>
</tr>
<tr>
<td>tj.PFX_TermlIndex</td>
<td>void → oid</td>
<td>Maps term identifiers to offsets into tj.PFX_Terms.</td>
</tr>
<tr>
<td>tj.PFX_Terms</td>
<td>void → oid</td>
<td>Term positions. Sorted by term id (not stored), then by collection position.</td>
</tr>
<tr>
<td>tj.PFX_size</td>
<td>void → int</td>
<td>Size of each node. Term nodes are always zero-sized.</td>
</tr>
</tbody>
</table>

**Mapping to and from Pathfinder nodes:**

<table>
<thead>
<tr>
<th>Table name</th>
<th>Monet BAT type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tj.PFX_pfpre</td>
<td>oid → oid</td>
<td>Mapping of PF/Tijah element positions to Pathfinder preorder positions. This table only stores element nodes, since words are not stored separately in Pathfinder.</td>
</tr>
</tbody>
</table>

**Auxiliary table:**

<table>
<thead>
<tr>
<th>Table name</th>
<th>Monet BAT type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tj.PFX_tid</td>
<td>void → oid</td>
<td>Mapping of all term or element positions to term identifiers. Only used when the element and term index tables have to be rebuilt.</td>
</tr>
</tbody>
</table>

Table 3.1: Tables in the PF/Tijah index. In names of collection-specific tables, the string PFX is replaced by the collection name.

- End positions ($e$) are not stored, because these are not necessary during computation. The only place where this information is used at the logical level is the the containment relation ($≺$) between two regions, specified in terms of region start and end positions, however the physical implementation of this relation (the staircase join) needs the starting position and the size.

- The name attribute ($n$) is stored in the collection-independent tables.

- The type attribute ($t$) is present implicitly by storing term and element positions in two separate tables. In practice, at the physical level it is never necessary to determine the type of a given region, this is always known from context.

- The score attribute ($p$) is not a persistent property of regions: it does not need to be stored in a table. The scores are stored in the intermediate result sets: see subsection 3.2.1.

This difference in representation – made possible by the separation of the logical and physical layers – enables the physical level to implement optimized data structures and algorithms.

### 3.1.2 Performing Position Lookups: indexfetchjoin

A frequently used SRA operator is the selection operator ($\sigma$). This operator selects all regions from a region set that match a set of conditions, e.g. all term regions with the name *information*. At the physical level, this is accomplished by returning a set of collection positions. Let’s assume that the collection positions of the term *information* have to be determined. For this, we have to perform the...
What are the collection positions of the term information?

\[ \text{collection positions} \]

\begin{align*}
\text{tj\_globalTerms} & \quad \text{tid (oid)} \quad \text{term name (str)} \\
& \quad \ldots \quad \ldots \\
& \quad 13 \quad \text{information} \\
& \quad 14 \quad \text{retrieval} \\
& \quad 12 \quad \text{XML} \\
& \quad \ldots \quad \ldots \\
\end{align*}

\begin{align*}
\text{tj\_PFX\_TermIndex} & \quad \text{tid (void)} \quad \text{offset (oid)} \\
& \quad \ldots \quad \ldots \\
& \quad 12 \quad 540 \\
& \quad 13 \quad 1060 \\
& \quad 14 \quad 2001 \\
& \quad \ldots \quad \ldots \\
\text{tj\_PFX\_Terms} & \quad \text{offset (void)} \quad \text{collection position (oid)} \\
& \quad \ldots \quad \ldots \\
& \quad 1060 \quad \text{position 1 (term 13)} \\
& \quad 1061 \quad \text{position 2} \\
& \quad \ldots \quad \ldots \\
& \quad 2000 \quad \text{position 941} \\
& \quad 2001 \quad \text{position 1 (term 14)} \\
& \quad \ldots \quad \ldots \\
\end{align*}

The tj\_PFX\_TermIndex has a void head column corresponding with the term identifiers, so this operation can be performed by positional lookup of each of the terms.

\begin{align*}
& \text{1} \quad \text{Look up the term information in the tj\_globalTerms table. This will determine the term’s identifier (tid).} \\
& \quad \text{This operation can be performed in a single table scan for multiple terms.} \\
& \text{2} \quad \text{Look up the starting position of the term in the tj\_PFX\_TermIndex table, using the tid found in the previous step. This will determine an offset into the tj\_PFX\_Terms table (start).} \\
& \quad \text{Then look up the starting position of the next term (always tid + 1) in the TermIndex table. This will determine an offset into the tj\_PFX\_Terms table (next).} \\
& \text{The tj\_PFX\_TermIndex has a void head column corresponding with the term identifiers, so this operation can be performed by positional lookup of each of the terms.} \\
& \text{3} \quad \text{Retrieve the term positions from the tj\_PFX\_Terms table: slice all rows in tj\_PFX\_Terms from position start up to next – 1.} \\
& \quad \text{This operation can also be performed by positional lookup (taking a slice from start up to next – 1).} \\
\end{align*}

This procedure can be performed to find the collection positions of a set of different terms in one pass. For fast computation, steps 2 and 3 have been implemented in a C function, as a plug-in to MonetDB. This function is called indexfetchjoin\(^2\).

\(^2\)A better name might be offsetindexjoin.
3.2 SRA Implementation – Reducing Intermediate Result Sizes

In this section we describe the physical level implementation of SRA on the PF/Tijah system. It has to be noted here that this implementation is similar or identical in many ways to the implementation of SRA on TIJAH, reported among others in [24]. Sometimes the implementation from TIJAH has been taken without change to PF/Tijah: the combination operators do not access any of the tables from the index, only the intermediate result sets, whose format was not changed in PF/Tijah. The computation and propagation operator implementations have been modified slightly to use the staircase join instead. Finally the selection (σ) operator and the complex selection and scoring operator (α) have been almost completely rewritten to use the staircase join and indexfetchjoin algorithms and the PF/Tijah index tables.

3.2.1 General SRA Implementation Issues

The final stage of the NEXI query compilation process is the translation from logical level SRA algebra to executable MIL code (shown in figure 3.3). A logical level SRA query plan (such as on page 20) is translated virtually line by line to MIL function calls representing the respective SRA operator. For example, the SRA selection operator is implemented at the physical level by the MIL functions select_term and select_node for term and element selection, respectively. At this stage, the decisions the user or administrator has made about e.g. which retrieval model to use are put in effect by translating instances of general SRA operators (e.g. ⊓p) to actual implementations (e.g. ⊓p using language models).

![Figure 3.3: Relationship between SRA operators, algorithms and tables](image)

Figure 3.3 shows the relationship between the logical level operators, their physical level implementations and the algorithms and data structures (tables) from the index these implementations use.
For example, the physical level implementations of the selection operator (\(\sigma\)), select_term and select_node, use the indexfetchjoin algorithm outlined in the previous section. The implementations of the score computation operator and the containment and score propagation operators use the staircase join to compute containment relationships between element regions and term regions, in addition to regular join to retrieve region sizes. Finally, the combination operators only operate on intermediate result region sets.

As indicated in the previous section, the score values associated with each region (\(p\) attribute) are not stored in the index, since this is a transient value only meaningful in the context of a particular query. To associate scores with regions, all of the SRA primitive implementations return transient binary tables that consist of region identifiers in the head and score values in the tail. We use collection positions as region identifiers, since these are guaranteed to be unique, and can easily be used as lookups for other information. The transient binary tables are in effect the physical level representation of region sets. Most of the primitives also accept one or more region sets as arguments, in addition to variables such as the value of \(\lambda\).

### 3.2.2 Running Example

In the following subsections, when illuminating the workings and implementation of the different SRA operators, we refer to a running example, based on the following NEXI query:

```
//article/sec[about(.//title,XML databases)]//p[about(.,information) or about(.,retrieval)]
```

The corresponding SRA query plan using the simple computation operator \((\sqcap_p)\) operator can be expressed procedurally as follows:

**Selection:**

\[
\begin{align*}
\text{article} & := \sigma_{n=\text{article},t=\text{node}(C)} \\
\text{sec} & := \sigma_{n=\text{sec},t=\text{node}(C)} \\
\text{title} & := \sigma_{n=\text{title},t=\text{node}(C)} \\
\text{p} & := \sigma_{n=\text{p},t=\text{node}(C)} \\
\text{XML} & := \sigma_{n=\text{XML},t=\text{term}(C)} \\
\text{databases} & := \sigma_{n=\text{databases},t=\text{term}(C)} \\
\text{information} & := \sigma_{n=\text{information},t=\text{term}(C)} \\
\text{retrieval} & := \sigma_{n=\text{retrieval},t=\text{term}(C)}
\end{align*}
\]

**Containment:**

\[
\begin{align*}
R_1 & := \text{sec} \sqsubseteq \text{article} \\
\text{Containment and implicit combination:}
R_2 & := \text{title} \sqcap_p \text{XML} \\
R_3 & := \text{title} \sqcap_p \text{databases} \\
R_4 & := R_2 \sqcap_p R_3 \\
\text{Upward propagation:}
R_5 & := R_1 \searrow R_4
\end{align*}
\]

**Score computation and explicit combination:**

\[
\begin{align*}
R_6 & := \text{p} \sqsupset_p \text{information} \\
R_7 & := \text{p} \sqsupset_p \text{retrieval}
\end{align*}
\]
3.2. SRA IMPLEMENTATION – REDUCING INTERMEDIATE RESULT SIZES

\[ R_8 := R_6 \sqcup_p R_7 \]  \hspace{1cm} (3.16)

Downward propagation:
\[ R_9 := R_8 \leftarrow R_5 \]  \hspace{1cm} (3.17)

The SRA query plan using the complex selection operator (α) can be expressed procedurally as follows:

Selection:
\[
\begin{align*}
\text{article} & := \sigma_{n=\text{article}, t=\text{node}}(C) \\
\text{sec} & := \sigma_{n=\text{sec}, t=\text{node}}(C) \\
\text{title} & := \sigma_{n=\text{title}, t=\text{node}}(C) \\
p & := \sigma_{n=p, t=\text{node}}(C)
\end{align*}
\]  \hspace{1cm} (3.18)

Containment:
\[ R_1 := \text{sec} \sqsubseteq \text{article} \]  \hspace{1cm} (3.22)

Score computation and implicit combination:
\[ R_2 := \alpha_{n=\text{XML}, d=\text{databases}}(\text{title}) \]  \hspace{1cm} (3.23)

Upward propagation:
\[ R_3 := R_1 \downarrow R_2 \]  \hspace{1cm} (3.24)

Score computation and explicit combination:
\[
\begin{align*}
R_4 & := \alpha_{n=\text{information}}(p) \\
R_5 & := \alpha_{n=\text{retrieval}}(p) \\
R_6 & := R_4 \sqcup_p R_5 \\
R_7 & := R_6 \leftarrow R_3
\end{align*}
\]  \hspace{1cm} (3.25-3.28)

These query plans, although expressed in logical level SRA, reflect the actual physical level query plans that are generated by the NEXI-to-MIL compiler.

3.2.3 Reducing Intermediate Result Sizes

Our stated goal is to devise ways to reduce intermediate result sizes to speed up IR query execution. In relational database systems, this is achieved by performing, if possible, the most selective operator first. For example, by ‘pushing down’ a selection operator through a join operator – i.e. performing selection first and joining on that result – the join operator, which is expensive on large data sets, has less work to perform.

Based on the SRA operator properties, [23] explores some basic optimization possibilities at the logical level, based on query rewriting also used in relational databases. More intensive research of these possibilities is left for future research. In this thesis however, we look at ways to reduce intermediate result sizes by changing the way the physical operators are implemented. These implementations mostly follow the formal definition, while taking advantage of e.g. relational techniques, optimized containment computation.

In principle, all probabilistic operators (computation, combination and propagation) return every region that is passed to them, following the formal definition. This means that when the system is
processing an IR query, since every region is kept, the intermediate result sizes will never decrease. We postulate that reducing intermediate result sizes will result in a system is more efficient: run faster and use less memory. From an IR perspective, the only regions that might be left out of intermediate results are those that are not relevant. The premature removal of irrelevant regions might also benefit the retrieval quality: a retrieval system is considered more useful when it shows fewer irrelevant results.

In the following subsections, we describe the implementations of the five classes of IR operators – region selection, region containment relations, score computation, combination and propagation. These implementations are then examined for opportunities to drop (possibly) irrelevant regions.

### 3.2.4 Selection Operator

The selection operator (\(\sigma\)) filters regions that satisfy a set of constraints from a region set. Selection is used as the starting point for query execution in SRA.

**Implementation**

The implementation of the selection operator is split into two functions: `select_term` selects term regions and `select_node` selects element regions.

According to the logical level specification of the selection operator, it can be used to select (filter) regions from any region set. In practice however, regions are always selected from the entire collection (this can be seen in the running example). This means that `select_term` and `select_node` can be optimized for selecting terms and elements from the collection. The `indexfetchjoin` algorithm plays a pivotal role in this (see subsection 3.1.2).

The intermediate region sets store the collection position and score value for each region in the set. For example, to select all elements named `article` from the collection, the following steps are performed by calling `select_node("article")`:

1. Find the term identifier for the element name `article` in `tj.globalTags`;
2. Call the `indexfetchjoin` algorithm to find the collection positions of the terms with the term identifier from step 1;
3. Create and return a transient intermediate result set consisting of the collection positions from step 2 in the head and default score values in the tail.

Term selection using `select_term` is implemented in the same way.

**Opportunities for Optimization**

When regions are selected from the collection, nothing is yet known of their relevance to the end result. The newly selected regions receive a default starting score, which can be 1 or 0, depending on the retrieval model. This score does not yet give any information about the relevance of these regions. Consequently, it is not possible to leave out any of the regions at selection time.
3.2. Containment Operators

The containment operators containing (\(\sqsupseteq\)) and contained-by (\(\sqsubseteq\)) return all regions from the left-hand-side argument that contain or are contained by regions from the right-hand-side argument, respectively.

Implementation

The running example uses a containment operator to find all sec elements that are contained by article elements (3.9 on page 30). The implementation of the containment operators uses the staircase join algorithm\[14\]. Figure 3.4 shows how this algorithm is used in this case. The contained-by operator returns every region from the left-hand-side argument that is contained by one or more regions in the right-hand-side argument. The implementation of the contained-by operator (\(\sqsubseteq\)) does the following:

1. Compute ancestor-descendant relation using the staircase join. Given two sets of regions, the staircase join returns a binary relation between ancestor and descendant regions. In the case of the contained-by operator we take ancestors from the right-hand-side argument (article elements) and descendants from the left-hand-side argument (sec elements).
2. Get only the descendants from the staircase join results. The staircase join returns a table with ancestors in the head column and their associated descendants in the tail; for contained-by we need the descendants, so the tail column.
3. Re-associate the original scores with the result from the previous step and return this result.

The containing operator (\(\sqsupseteq\)) is implemented in a similar fashion.

Opportunities for Optimization

The containment operators operators are only used when no probabilistic operator (such as score computation) has yet been performed. This means that all regions still have the default score given by the selection operator. Similar to the the selection operator, it is not possible to leave out any regions when computing containment.

3.2.6 Score Computation Operators – \(\sqsupseteq_{p}, \alpha\)

The simple probabilistic containment operator or computation operator (\(\sqsupseteq_{p}\)) computes scores for a set of element regions based on the occurrence of term regions within them, where these term regions are all instances of the same term. In addition, a complex computation operator (\(\alpha\)) is defined, which can score a set of element regions based on their relevance to a set of terms (not term regions). The definition of these computation operators allows the retrieval model that computes scores to be changed independently without changing the semantics of the query, by delegating the actual computation to a auxiliary functions (\(f_{\sqsupseteq}\) and \(f_{\sqsupseteq_{\alpha}}\)).

To date, only two retrieval models (i.e. instantiations of the auxiliary function) have been implemented in the PF/Tijah system. The language modeling approach (LM)[15] has been translated from its
query: sections contained by articles:
\((σ_{n=secftime=\text{node}}) \cap (σ_{n=articleftime=\text{node}})\)

<table>
<thead>
<tr>
<th>position</th>
<th>score</th>
<th>position</th>
<th>score</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>12</td>
<td>1,000</td>
<td>20</td>
<td>1,000</td>
</tr>
<tr>
<td>200</td>
<td>1,000</td>
<td>350</td>
<td>1,000</td>
</tr>
<tr>
<td>540</td>
<td>1,000</td>
<td>400</td>
<td>1,000</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

\(tjPFX\_size\) staircase join

ancestor-descendant relation:

<table>
<thead>
<tr>
<th>ancestor</th>
<th>descendant</th>
</tr>
</thead>
<tbody>
<tr>
<td>position</td>
<td>collection</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>12</td>
<td>20</td>
</tr>
<tr>
<td>200</td>
<td>350</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

reassociate scores

\(\times\)

<table>
<thead>
<tr>
<th>position</th>
<th>score</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>20</td>
<td>1,000</td>
</tr>
<tr>
<td>350</td>
<td>1,000</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

take tail column (descendants)

Figure 3.4: Using the staircase join for region containment
3.2. SRA IMPLEMENTATION – REDUCING INTERMEDIATE RESULT SIZES

TIJAH implementation. The Normalized Log Likelihood Ratio (NLLR)[21] has been implemented from scratch. These two retrieval models are described in the next subsections.

Language Modeling

The language modeling approach to XML information retrieval has been used successfully by the CIRQUID research group to participate in several IR evaluation initiatives, such as INEX and TREC. Language models provide an elegant solution to the problem of determining the relevance of a document with respect to a query. Language models are statistical, they assign a probability to each document, which expresses what the chances are that the given document would generate the query. For more information, see [15]. The language modeling retrieval model we use is defined as follows. For each document \( \text{doc} \) the relevance \( S \) to a query \( q \) (a set of terms) is defined as:

\[
S(\text{doc}|q)^{LMS} = \prod_{t \in q} (\lambda P(t|\text{doc}) + (1 - \lambda)P(t|\text{col}))
\] (3.29)

The foreground probability \( P(t|\text{doc}) \) expresses the probability that the term \( t \) occurs in the document (or element) \( \text{doc} \). Likewise, the background probability \( P(t|\text{col}) \) expresses the probability that the term occurs in the collection. Without the background probability, a score of zero would be assigned to documents that do not contain all of the query words. Not assigning a score of zero expresses the belief that a retrieval model can never be absolutely certain that a document is not relevant, or less relevant than the rest of the collection. This model is known as the Language Modeling approach with linear interpolated smoothing[17]. The model is abbreviated LMS in the experimental system.

While the language modeling approach is defined for documents, it turns out that the model is usable for individual elements as well[24]. The probability values are computed as follows:

\[
P(t|\text{doc}) = \frac{tc(t,\text{doc})}{len(\text{doc})}
\] (3.30)

\[
P(t|\text{col}) = \frac{tc(t,\text{col})}{len(\text{col})}
\] (3.31)

\( tc(t,\text{doc}) \) represents the number of times that the term \( t \) occurs in document (or element) \( \text{doc} \). \( len(\text{doc}) \) represents the length of document or element \( \text{doc} \). Element length is in this case defined as how many words the element contains.

Normalized Log-Likelihood Ratio

A variant of the language modeling approach shown above is the Normalized Log-Likelihood Ratio (NLLR) model. The NLLR function is defined as follows [21]:

\[
S(\text{doc}|q)^{NLLR} = \frac{1}{len(q)} \sum_{t \in q} \log \left( \frac{(1 - \lambda)P(t|\text{doc}) + \lambda P(t|\text{col})}{\lambda P(t|\text{col})} \right)
\] (3.32)

\( ^3 \)The \( \lambda \) term in the divisor is absent in [21]; it has been added to produce zero score values for regions that contain none of the query terms.
The Normalized Log-Likelihood Ratio (NLLR, equation 3.32) formula is very similar to the language modeling (LMS, equation 3.29) formula, except in the following points:

1. The retrieval model will assign a zero score to regions that do not contain any of the query terms. This is because $P(t|\text{doc})$ will be exactly zero for such regions, leaving $\log \left( \frac{\lambda P(t|\text{col})}{\lambda P(t|\text{col})} \right) = 0$.

2. The score values for NLLR are larger than those generated by LMS for the same region and query combinations. $P(t|\text{doc})$ and $P(t|\text{col})$ are defined in the same way as for the LMS model. The first point is especially interesting for our goal; see below.

**Implementation of the Simple Computation Operator**

At the logical level, the different retrieval models are modeled as separate auxiliary functions ($f_\downarrow$), to be instantiated at execution time. When translation from logical to physical takes place, a different instantiation of the entire computation operator operator is chosen: instances of the logical level computation operator ($\sqcup_p$) translate to calls to e.g. $p\_\text{containing}\_t\_\text{LMS}$ or $p\_\text{containing}\_t\_\text{NLLR}$ MIL functions, depending on the retrieval model chosen by the user or administrator. These MIL functions have some overlap in functionality.

There are differences between the definition of the retrieval models above – expressed as scoring function $S$ – and the definition of the (arguments to the) auxiliary score computation function. Some modification is therefore required to use the retrieval model specifications in SRA:

- **Difference**: the $S$ function is defined on a single document, whereas the auxiliary function receives a set of element regions. This is apparent from formal definition of the computation operator ($\sqcup_p$) given in equation 2.5 on page 17.

  **Resolution**: the difference between documents and element regions is not a problem: each individual element region is simply used as the document argument to $S$. The $S$ function is applied to a set of regions by applying the $S$ function to each document individually. This can be done quite efficiently on MonetDB.

- **Difference**: the $S$ function is defined on a set of query terms (i.e. the words themselves), whereas the auxiliary function receives a set of term regions (i.e. the term occurrences or positions). This is also apparent from the formal definition.

  **Resolution**: the fact that term occurrences are passed to the score computation function is actually convenient for its implementation, because it requires query term positions anyway to be able to use the staircase join.

Furthermore, due to the way that queries are expressed in SRA (see below for an example), the auxiliary score computation function ($f_\downarrow$) always receives all the occurrences of only one query term at a time. Performing score computation for each query term separately and then combining the scores using the and operator implemented as a product is therefore mathematically valid, since this results in exactly the product expressed by the product operator ($\prod$) in the $S$ definition.

- **Difference**: the $S$ function assigns new scores to regions, whereas the auxiliary function might receive regions that already have scores assigned.
3.2. SRA IMPLEMENTATION – REDUCING INTERMEDIATE RESULT SIZES

**Resolution:** The implementation of the auxiliary function should take into account the scores already present. For the language modeling function, it is safe to multiply the existing score with the score computed by the computation function: a freshly selected region always has a score of 1.

Suppose that the user chose to use the language modeling retrieval model:

$$
S(doc|q)^{LMS} = \prod_{t \in q} (\lambda P(t|doc) + (1 - \lambda)P(t|col))
$$

Taking into account the points above, the language modeling formula is expressed in an auxiliary score computation function as follows (slightly adapted from [24]):

$$
f^{LMS}_{\triangleright}(r_1, R_2) = r_1.p \cdot \left( \frac{\lambda \sum_{r_2 \in R_2 | r_2 \prec r_1} P_2 - P}{\text{len}(r_1)} + (1 - \lambda) \frac{|R_2|}{\text{len}(col)} \right)
$$

$|R_2|$ is assumed to equivalent with the number of times the term occurs in the collection. Since $R_2$ is always the result of a selection of terms from the entire collection, this assumption is justified (see lines 3.5 and 3.10 of the running example on page 30). $\text{len}(col)$ is the collection size: the number of terms in the entire collection. The product with $r_1.p$ assures that the existing score on $r_1$ is taken into account.

Our implementation of the containment function ($\prec$) uses the staircase join algorithm to determine if an element contains a certain term, in the same manner as the containment operators (see subsection 3.2.5). The term count for each element can be computed by taking the result of the staircase join computation and counting for each ancestor region how many descendant regions it has. This is represented in the formula above by summing the existing scores – on freshly selected regions, these are always 1. The length $\text{len}$ of an element can be taken from the $tj.PFX.size$ table directly. Using this information the scores can now be computed for all elements that are present in the staircase join result (i.e. the ancestor elements that have one or more descendants).

Ancestor regions that have no descendant will not be present in the staircase join result. To comply with the formal definition of the computation operator, the implementation will have to return every region, whether it has descendant (i.e. contains the term) or not. To this end, the regions that are present in the left-hand-side argument but not in the staircase join result, are added to the result region set. They receive a background score ($P(t|col)$).

**Implementation of the Complex Computation Operator**

In the previous chapter we described the complex score computation operator ($\alpha$), which combines selection ($\sigma$), score computation ($\triangleright_p$) and score combination ($\sqcup_p$ or $\sqcap_p$). The most important advantage of the complex computation operator is that it has the potential to be much faster. As shown in the previous chapter and in the running example, when the simple computation operator ($\triangleright_p$) is used, relevance is computed for each region set and term combination separately. In contrast, the complex computation operator computes scores for several terms at once. This contrast can be seen in lines 3.10 and 3.11 versus 3.23 in the running example (page 30).

For queries with a lot of query terms, using the simple computation operator becomes computationally expensive. If the retrieval model that is used for score computation uses a region containment algorithm in order to see if the term *information* is contained in the regions from the supplied region...
set, this algorithm is executed for each set of term occurrences separately – twice in this example. Since this algorithm might support computing containment for multiple sets of terms at the same time, the complex computation operator only has to execute the containment algorithm once for all of the terms occurrences. So the performance advantage of using the complex computation operator increases when the number of search terms per search region increases.

In [24], the author asserts that the major disadvantage of the complex computation operator (α) is its complexity. In addition to performing selection, computation and combination, the definition on both the logical and physical levels has to be modified when features such as phrase search are added. On the other hand, if a retrieval model uses methods (such as probabilistic relationships between different query terms) that cannot be expressed using simple probabilistic containment alone, using this model in SRA would require introduction of additional operators (modification of the conceptual level), whereas this might be expressed inside the auxiliary scoring function \( f_{\alpha, \sqcap} \) for the complex computation operator without change to the conceptual level.

To sum up, the choice between using the simple or complex computation operator is very much dependent on the requirements. Fortunately, these two variants are not mutually exclusive; they might even be used in the same query plans without problems. This might for example be of use in queries that combine text search with multimedia search: the complex computation operator can be used to perform text search, while the simple computation operator, which is in principle oblivious to region types, can be used for other types.

**Opportunities for Optimization**

The implementations of the computation operators described above precisely implement their respective formal definitions. In the case of the simple computation operator \( \sqcap_p \), every region from the left-hand-side argument is returned, with an associated score based on the occurrence of regions from the right-hand-side argument. The implementation even explicitly adds regions that are not in the result of the containment join: these regions do not contain the query term(s), but according to the formal definition, should still occur in the result of the computation operator. Similarly the complex computation operator \( \alpha \) returns every region that satisfies the name and type test, regardless of whether this region contains any of the query terms.

From the perspective of the implementation, it makes sense to only return regions that contain one or more query terms: this makes the explicit adding of regions that contain none of the terms superfluous. This would also reduce intermediate result sizes.

For the NLLR retrieval model, this might even be a ‘natural’ optimization, since it assigns score of zero to regions that do not contain query words. A score of zero means that a region is absolutely not relevant, so they could be ‘pruned’ from intermediate regions anyway. For LMS, however this is not so natural, since it still assigns a background score.

To sum up, we identify two implementation variants of the computation operator:

- The **correct** variant, which follows the formal definition exactly, returning all regions, even if they contain none of the query words;
- The **optimized** variant, which only returns regions that contain one or more query words.

\(^4\)a probability \( P(A) = 0 \) means that \( A \) is impossible
3.2. Score Combination Operators

The combination operators and and or (\(\cap_p\) and \(\cup_p\)) are used to combine scores from instances of the score computation operator (\(\sqcap_p\) or \(\alpha\)). \(\cap_p\) is used for and combinations, \(\cup_p\) is used for or. See lines 3.12 and 3.16 from the running example (page 30).

\(R_1 \cap_p R_2\) returns only regions that are present in both \(R_1\) and \(R_2\), i.e. regions that are present in the intersection \(R_1 \cap R_2\). The resulting score values are determined by an auxiliary function (\(\otimes\)). \(R_1 \cup_p R_2\) returns regions that are present in any of the two regions sets. For regions that are present in both sets, the scores are defined by an auxiliary function (\(\oplus\)), otherwise the regions retain their original score.

Similar to the score computation operators, the actual combination function can be chosen independently, by delegation to auxiliary functions \(\otimes\) and \(\oplus\). These might be defined as \(\otimes = \times\) and \(\oplus = +\), or \(\otimes = \max\) and \(\oplus = \avg\).

Implementation

Based on the decision made by the user or administrator about which auxiliary functions to use, instances of the combination operators (\(\cap_p\) and \(\cup_p\)) are translated to calls to e.g. \(\text{and}_\text{prod}(\otimes = \times)\) and \(\text{or}_\text{sum}(\oplus = +)\) MIL functions.

MonetDB supports multiplexed arithmetic operators that, given one or more tables, compute the operator result of the tail values of these tables. The multiplexed product operator \([*]\) for example takes two tables and multiplies the tail values of every combination of rows that have identical head values (equi-join). Any rows in one table that are not present in the other are not returned, so the result is an intersection of the two tables. See figure 3.5. Operators such as \([+]\) and \([-]\) are implemented in the same fashion, also returning an intersection.

It should be noted here that at the logical level, the combination operators return exactly the same results (except for their scores, defined by \(\otimes\) and \(\oplus\)). This is the case because the computation operator returns every region that it receives. This means that the two arguments to the combination operators always contain the same regions. The intersection of two sets that contain the same elements is the same as a union of those sets.
The fact that the arguments to the combination operators always contain the same regions can be demonstrated by the running example, lines 3.10 through 3.12 (page 30). The two instances of the computation operator (\(\cap_p\)) each have the same region set as their left-hand-side argument: title elements. \(R_1\) and \(R_2\) contain the same regions, if and only if the computation operator returns all regions from the selection \(\sigma_{n=article,j=node(C)}\). In practice this is always the case when combining all regions: multiple about expressions combined with an and or or always have the same context region set. The same goes for the implicit combination between the search terms of the same about expression.

The correct implementation of the computation operator returns every region, following the formal definition precisely. Because in this case there is no difference between union and intersection, the original TIJAH implementations of the combination operators used the multiplex operator (intersection) for both and and or combinations. In subsection 3.2.6 however, we have introduced the possibility that the implementation of the computation operator will not return every region, only those containing the query terms. Since the arguments to the or operator (\(\sqcup_p\)) might now contain different regions, when using a multiplex operator (intersection) this operator is no longer semantically correct: it drops regions that are present in only one of the argument sets. To resolve this, we have implemented the or operator to follow the formal definition precisely: regions from \(R_1\) or \(R_2\) that are not in the intersection \(R_1 \cap R_2\) are added to the result (these regions are in the symmetric difference \(\Delta\)). It would however be interesting to see the effect this ‘correction’ has on the effectiveness and performance of the system.

Using just a multiplex operator to implement the and operator (\(\cap_p\)) will follow the formal definition precisely. However, it may be that now too many regions will be omitted. Consider the first about expression in the running example: this searches title elements for the term XML and databases. If a certain title element does not contain the term XML, but it does contain the other term database, the system will still not return it: the computation operator that processed the term XML has discarded this title, and it will never be added to the result of an and combination. It might however be argued that this title element might still be relevant.

To investigate this issue, we added an implementation of the and operator that works in the same way as the correct implementation of the or operator: regions not in the intersection are added to the result. Because the auxiliary function for and (\(\otimes\)) is usually implemented as a product – this was shown to produce better retrieval effectiveness – we have to compensate the regions that contain only one of the terms. Consider the following sets of regions:

\[
\begin{align*}
R_1 &= \{(s = 1, ..., p = 0.5),(s = 2, ..., p = 0.5)\} \\
R_2 &= \{(s = 2, ..., p = 0.5),(s = 3, ..., p = 0.5)\} \\
R_3 &= R_1 \cap_p R_2
\end{align*}
\]

\(R_1\) and \(R_1\) are the results of a computation operator: the presence of regions in these sets indicates that they contained the term in question. \(R_3\) then contains the result of the and combination of these two sets.

The correct and operator implementation, using \(\otimes = \times\) would produce the following result in \(R_3\):

\[
R_3 = \{(s = 2, ..., p = 0.25)\}
\]

This is in accordance with the formal definition. In contrast, a naive or-like implementation that adds the elements not in the intersection without compensation would give:

\[
R_3 = \{(s = 1, ..., p = 0.5),(s = 3, ..., p = 0.5),(s = 2, ..., p = 0.25)\}
\]
The region with \( s = 2 \) receives a lower score than the others, even though it is contained in both sets (i.e. contains both terms)! This would show \( s = 2 \) below all other regions in the ranking: this is clearly not the intention. Alternatively, we could define the operator to return the following:

\[
R_3 = \{(s = 2, ..., p = 0.25), (s = 1, ..., p = 0.02), (s = 3, ..., p = 0.02), \}
\]

We ensured that the regions not in the intersection receive a smaller score, and the ordering between those regions is retained:

\[
R_1 \cap_{alt} R_2 = (R_1 \cap_{p} R_2) \cup \{ (r_1, s_1, e, r_1, n, r_1, t, r_1, p \cdot \frac{1}{2} \cdot \minscore(R_1 \cup_{p} R_2) | (r_1 \in R_1 \lor r_1 \in R_2) \land r_1 \notin (R_1 \cap R_2) \}
\]

where

\[
\minscore(R) = \min(\{r, p | r \in R\})
\]

### Opportunities for Optimization

The implementation of the and operator \((\cap_p)\) already reduces intermediate result sizes somewhat: the intersection of two region sets is always as large as or smaller than the smallest of the two regions \((|R_1 \cap R_2| \leq \min(|R_1|, |R_2|))\). With the or operator \((\cup_p)\) result sizes are never reduced: the result is always as large as or larger than the largest of the two regions \((|R_1 \cap R_2| \geq \max(|R_1|, |R_2|))\).

It would not be useful to explicitly drop regions that have a zero score as a result of the score combination. In the case of a product implementation, this would mean that one (or both) of the regions from \(R_1\) or \(R_2\) has a zero score after score computation. These regions are better filtered at score computation time.

There are then no real opportunities for optimization. In the previous subsection we did identify two variants for each of the propagation operators:

- For the or operator we identified:
  - the old incorrect\(^5\) variant as it was implemented in TIJAH using just a multiplex operator, and
  - the new correct variant, which follows the formal definition exactly, explicitly adding regions that are not in the intersection

- For the and operator we identified:
  - the correct variant, which follows the formal definition exactly, returning an intersection, and
  - an alternative variant, which explicitly adds regions not in the intersection, while compensating those regions to have a lower score than regions that are in the intersection.

\(^5\)Note that the implementation is only incorrect in the context of computation operators that do not return every region
CHAPTER 3. SRA AT THE PHYSICAL LEVEL

3.2.8 Score Propagation Operators

The upward propagation operator $R_1 \triangleright R_2$ propagates scores from contained (descendant) regions from $R_2$ to containing (ancestor) regions from $R_1$ (3.13 on page 30). It might be that a sec element contains multiple title elements. The scores assigned to these title elements must be combined and propagated to their common ancestor sec element.

The downward propagation operator $R_1 \triangleleft R_2$ propagates scores from containing (ancestor) regions from $R_2$ to contained (descendant) regions from $R_1$ (3.17 on page 31). It might be that a sec element has multiple ancestor article elements. Scores assigned to these article elements must be combined to their common descendant sec element.

Similar to the score computation operators, the actual propagation function is delegated to auxiliary functions ($f_{\triangleright}$ and $f_{\triangleleft}$). The auxiliary function for upward propagation from contained to containing regions might be defined as follows [24]:

$$f_{\text{sum}}^\triangleright (r_1, R_2) = r_1 \cdot p \cdot \sum_{r_2 \in R_2 \mid r_2 \prec r_1} r_2 \cdot p$$ (3.37)

This function sums the scores of all the regions from $R_2$ that are descendants of region $r_1$. This sum is then multiplied with the score already present on $r_1$.

Our implementation of the propagation operators again uses the staircase join algorithm to evaluate the containment relation expressed by the containment function ($\prec$) in e.g. the auxiliary function for upward propagation ($f_{\text{sum}}^\triangleright$).

Similar to the score computation operator ($\sqcap p$), the formal definition of the propagation operators specifies that all regions from the left-hand-side argument must be present in the result. The auxiliary function expressed above assigns a score of zero to regions that have no descendants in the left-hand-side argument, thereby expressing the view that regions that have no descendants are not relevant. When an element has a score of zero, that means that that element is absolutely not relevant. In this case, the element might as well be removed from the result.

To sum up, we identify two implementation variants for both of the propagation operators:

- The correct variant returns every region, even if it is outside the containment relation specified by the propagation operator. Such a region receives a score of zero in accordance with the formal definition.
- The optimized variant leaves out regions that are outside the containment relation (i.e. regions that would receive a zero score).

3.3 Conclusion

In this chapter we have examined the physical level implementation of the SRA operators. We have looked at ways to increase efficiency by reducing intermediate result sizes. We have identified two

---

6 A single sec element having multiple article ancestors is of course not likely: a section is only part of a single article. However, there exist schemas that allow elements of the same name to be nested arbitrarily, e.g. the div element in (X)HTML.
classes of score operators – computation, combination and propagation – to be likely candidates for optimization.

The optimizations we have identified can be summarized as follows:

– For the computation operators ($\square_p$ and $\alpha$), regions that contain none of the query terms are candidates to be omitted from the result. In addition, the NLLR retrieval model might be naturally suited for this optimization.

– The combination operators $\text{and}$ and $\text{or}$ ($\cap_p$ and $\cup_p$), while not offering a lot of opportunities for optimization themselves, are nevertheless affected by changes made to the computation operators. We have to be careful when implementing these combination operators, so that possibly relevant regions are not removed from the intermediate results.

– The upward and downward propagation operators ($\uparrow$ and $\downarrow$) can omit regions that have no descendants or ancestors, respectively.

Any decision made on the physical level to deviate from the formal definitions will have consequences on efficiency and retrieval performance. In the next chapter we describe how we have evaluated the possible implementations of these operators with respect to these factors.
Chapter 4

Investigating Retrieval Effectiveness and Efficiency

In the previous chapter we have identified some possible modifications to the implementations of the SRA operators, with the purpose to improve efficiency of the system. These modifications might have have an effect on two factors:

- **Retrieval effectiveness:** retrieval effectiveness, retrieval quality or retrieval performance of an IR system is usually measured by determining how many relevant results the system returns for a given query. We hope our optimizations do not adversely affect retrieval effectiveness.

- **Efficiency:** in this case, we look at memory usage and speed as indicators of efficiency. We are undertaking this effort to achieve a more efficient system through our modifications.

In this chapter we examine how we evaluate the modifications suggested by the previous chapter according to these two criteria.

For database systems, it is customary to evaluate performance using **benchmarks**. These benchmarks prescribe a data set and a set of queries. Measuring the time it takes these queries on a certain database system allows a fair performance comparison on with other database systems. These benchmarks can also be used to perform performance testing of a single database, to determine the improvement in performance of e.g. a new index structure.

For the evaluation of information retrieval systems, a similar approach is taken. IR evaluation initiatives such as TREC and INEX supply a set of documents (**test collection**) and a set of queries (**topics**). The results that these queries produce is then compared with **relevance assessments**, to determine the retrieval performance. These assessments are sometimes made by the other participants in the initiative. The focus of IR retrieval evaluation is mostly on retrieval effectiveness, in contrast to database system evaluation, which focuses mainly on performance.

Our approach can be divided into two activities: **small-scale testing** and **large-scale evaluation**. These approaches are reported in the next sections.
4.1 Small-Scale Testing

For small-scale testing, we use an XML document together with simple NEXI queries to illustrate the effect that decisions at the physical level can have on result rankings. We will show the difference in ranking between correct and optimized implementations of the three classes of SRA operators.

The structure of the test document, shown in figure 4.1, follows the structure of a simple thesis: chapters, sections, titles and paragraphs are all represented using `chapter`, `section`, `title` and `para` elements, respectively. When NEXI queries on this small XML example are explained, the result element ids are reported, instead of their paths or contents.

We use the small document together with simple NEXI queries to test each SRA operator. These queries are shown in table 4.1. The queries and the example document are actually used in the PF/Tijah source code to perform automated testing.

<table>
<thead>
<tr>
<th>SRA operator</th>
<th>NEXI queries</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Computation:</strong></td>
<td></td>
</tr>
<tr>
<td>$\sqcup_p$ and $\alpha$</td>
<td>//section[about(.,information_retrieval)]</td>
</tr>
<tr>
<td><strong>Combination:</strong></td>
<td></td>
</tr>
<tr>
<td>$\sqcup_p$ (or)</td>
<td>//title[about(.,Pathfinder) or about(.,TIJAH)]</td>
</tr>
<tr>
<td>$\sqcap_p$ (and)</td>
<td>//title[about(.,Pathfinder) and about(.,TIJAH)]</td>
</tr>
<tr>
<td><strong>Propagation:</strong></td>
<td></td>
</tr>
<tr>
<td>$\uparrow$ (up)</td>
<td>//section[about(.,//title,information_retrieval)]</td>
</tr>
<tr>
<td>$\downarrow$ (down)</td>
<td>//chapter[about(.,information_retrieval)]//section[about(.,database)]</td>
</tr>
</tbody>
</table>

Table 4.1: NEXI queries on the small test document

4.1.1 Score Computation Operators

We will consider two implementations of the computation operator:

- The **correct** variant returns all regions, regardless of whether they contain the words or not; background scores might be assigned to regions that do not any query words;

- The **optimized** variant leaves out regions that do not contain any query words (i.e. regions that would receive a background score);

To test the implementation of these variants we used the following query:

//section[about(.,information_retrieval)]

This query should retrieve section elements that are about information and retrieval, with the most relevant section at the top of the ranking. We consider the section elements $s_2$, $s_4$, $s_5$ and $s_1$ to be relevant to this query. The query plan using the complex selection and computation operator ($\alpha$) is as follows:

\[
\text{section} := \sigma_{n=\text{section}} \circledast \text{node}(C) \tag{4.1}
\]
\[
R_1 := \alpha_{n=.} \sqcup \text{[information.retrieval]}(\text{section}) \tag{4.2}
\]
CHAPTER 4. INVESTIGATING RETRIEVAL EFFECTIVENESS AND EFFICIENCY

<?xml version="1.0"?>
<thesis id="thesis"><title id="t1">PF/Tijah</title>
  <chapter id="c1"><title id="t2">Structured information storage and retrieval</title>
    <para id="p1">This chapter is about XML databases and retrieval systems.</para>
    <section id="s1"><title id="t3">XML databases</title>
      <para id="p2">This section is about XML databases.</para>
      <para id="p3">It contains the words XML and databases many times,
        more than say information and retrieval.
        It should be more relevant to XML databases than
        the next section.</para>
    </section>
  </chapter>
  <chapter id="c2"><title id="t5">Pathfinder and TIJAH</title>
    <para id="p6">This chapter is about Pathfinder and TIJAH.</para>
    <section id="s3"><title id="t6">Pathfinder</title>
      <para id="p">This section is about the Pathfinder XML database. This section is
        more about Pathfinder than any other section is about Pathfinder.</para>
    </section>
    <section id="s4"><title id="t7">TIJAH</title>
      <para id="p8">This section is about the TIJAH XML information retrieval
        system. This section is more about TIJAH than any other section is
        about TIJAH.</para>
    </section>
  </chapter>
  <appendix id="a1">
    <section id="s5"><title id="t8">XML definitions</title>
      <para id="p9">This section contains definitions that are used in XML databases
        and XML information retrieval</para>
    </section>
  </appendix>
</thesis>

Figure 4.1: Test document
Our experimental system produces the following rankings for the correct and optimized implementations:

<table>
<thead>
<tr>
<th>Rank</th>
<th>Element</th>
<th>Correct?</th>
<th>Score</th>
<th>Element</th>
<th>Relevant?</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>s2</td>
<td>+</td>
<td>0.0385</td>
<td>s2</td>
<td>+</td>
<td>0.0385</td>
</tr>
<tr>
<td>2</td>
<td>s5</td>
<td>+</td>
<td>0.0175</td>
<td>s5</td>
<td>+</td>
<td>0.0175</td>
</tr>
<tr>
<td>3</td>
<td>s4</td>
<td>+</td>
<td>0.0162</td>
<td>s4</td>
<td>+</td>
<td>0.0162</td>
</tr>
<tr>
<td>4</td>
<td>s1</td>
<td>+</td>
<td>0.0122</td>
<td>s1</td>
<td>+</td>
<td>0.0122</td>
</tr>
<tr>
<td>5</td>
<td>s3</td>
<td>−</td>
<td>0.0054</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The correct ranking also shows section s3, which is not relevant to the query – it does not contain the search terms. This is because in the correct implementation, the computation operator returns every region, regardless of whether it contains the search terms. Section s3 received a background score, which is much smaller than the scores that the relevant regions received.

Precision and recall values for these rankings are as follows (see subsection 4.2.1 for an explanation of these measures):

<table>
<thead>
<tr>
<th>Measure</th>
<th>Correct</th>
<th>Optimized</th>
</tr>
</thead>
<tbody>
<tr>
<td>Recall</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Precision</td>
<td>$\frac{4}{7}$</td>
<td>1</td>
</tr>
<tr>
<td>Average precision</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Average precision is not affected since the ranking of relevant elements is not different: the correct ranking only shows one irrelevant result at the bottom, below all relevant elements.

4.1.2 Score Combination Operators

As we have seen in the previous chapter, the results of combination operator application is affected by the computation operator implementation. We could not identify meaningful optimization opportunities for these operators. We are however interested in the correctness of the implementation of the combination operators when the computation operator is optimized.

**or Combination**

For the **or** operator ($\sqcup_p$), we identified two different implementations in the previous chapter:

- The **correct** implementation, using the multiplex operator and explicitly adding elements not in the intersection;
- The **incorrect** implementation, using just a multiplex operator (intersection). We reasoned that in case of an optimized computation operator implementation, this optimized combination operator implementation will leave out possibly relevant elements.

To demonstrate the effect of these different implementations, we use the following NEXI query:

```nxml
//title[about(.,Pathfinder) or about(.,TIJAH)]
```
This query should return titles that contain *Pathfinder* or *TIJAH* or both. We consider titles t5, t6, t7 and t1 to be relevant to this query. We consider titles that contain both terms (t5) to be more relevant than titles that contain only one of the terms. The query plan using the complex selection operator (\(\alpha\)) is as follows:

\[
\begin{align*}
title & := n_{=\text{title} \cdot \text{node}}(C) \\
R_1 & := n_{=\text{node}}(\{\text{Pathfinder}\}) \cdot (\text{title}) \\
R_2 & := n_{=\text{node}}(\{\text{TIJAH}\}) \cdot (\text{title}) \\
R_3 & := R_1 \cup_p R_2
\end{align*}
\]

Our experimental system produced the following rankings:

<table>
<thead>
<tr>
<th>Rank</th>
<th>Computation: Correct</th>
<th>Correct Score</th>
<th>Incorrect Score</th>
<th>Optimized Correct</th>
<th>Optimized Incorrect Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>t6</td>
<td>0.5793</td>
<td>0.5793</td>
<td>t6</td>
<td>0.5793</td>
</tr>
<tr>
<td>2</td>
<td>t7</td>
<td>0.5793</td>
<td>0.5793</td>
<td>t7</td>
<td>0.5427</td>
</tr>
<tr>
<td>3</td>
<td>t5</td>
<td>0.5793</td>
<td>0.5793</td>
<td>t5</td>
<td>0.5366</td>
</tr>
<tr>
<td>4</td>
<td>t1</td>
<td>0.3293</td>
<td>0.3293</td>
<td>t1</td>
<td>0.2927</td>
</tr>
<tr>
<td>5</td>
<td>t8</td>
<td>0.0793</td>
<td>0.0793</td>
<td>t8</td>
<td>0.0793</td>
</tr>
<tr>
<td>6</td>
<td>t2</td>
<td>0.0793</td>
<td>0.0793</td>
<td>t2</td>
<td>0.0793</td>
</tr>
<tr>
<td>7</td>
<td>t3</td>
<td>0.0793</td>
<td>0.0793</td>
<td>t3</td>
<td>0.0793</td>
</tr>
<tr>
<td>8</td>
<td>t4</td>
<td>0.0793</td>
<td>0.0793</td>
<td>t4</td>
<td>0.0793</td>
</tr>
</tbody>
</table>

As expected, the rankings do not differ when the computation operator is implemented correctly, returning all regions (1 and 2). These rankings do however show a small anomaly: the title with both query words, t5, is shown below the titles with only one query word. This is due to the score that the computation operator assigned to the titles: because t6 and t7 contain only one word, and that word is also the query word, the score is higher than for t5, which contains three words, only one of which is the query word.

The other two rankings do show a significant difference: ranking 3 using the incorrect or combination only shows title t5: this title contains both search terms *Pathfinder* and *TIJAH*. Ranking 3 does not contain the the titles that have only one of the query words; these are present in ranking 2. This is reflected in the difference in recall between rankings 2 and 3:

| Recall: Correct | 1 | 1 |
| Precision: Correct | 4/8 | 4/8 |
| Average precision: Correct | 1 | 1 |

In this case, the correct or operator delivers the best retrieval performance.

and **Combination**

In the previous chapter we showed that the and operator definition (\(\cap_p\)) is already somewhat optimized when implemented correctly. We examine a variant similar to the correct or implementation,
4.1. SMALL-SCALE TESTING

defined by equation 3.35 on page 41 which we call alternative. This implementation is not technically correct, because it does not follow the formal definition. We used the following query to examine the differences between these implementations:

//title[about(.,Pathfinder) and about(.,TIJAH)]

This query should return titles that contain both Pathfinder and TIJAH. We consider title t5 to be relevant to this query. The query plan using the complex selection and computation operator (α) is almost identical to the previous plan for the or operator:

\[
\begin{align*}
\text{title} & := \sigma_{n=title, t=node}(C) \quad (4.7) \\
R_1 & := \alpha_\exists[\text{Pathfinder}](\text{title}) \quad (4.8) \\
R_2 & := \alpha_\exists[\text{TIJAH}](\text{title}) \quad (4.9) \\
R_3 & := R_1 \cap P R_2 \quad (4.10)
\end{align*}
\]

Our experimental system produced the following rankings:

<table>
<thead>
<tr>
<th>Computation:</th>
<th>Correct</th>
<th>Optimized</th>
</tr>
</thead>
<tbody>
<tr>
<td>Combination:</td>
<td>Alternative</td>
<td>Correct</td>
</tr>
<tr>
<td>Rank</td>
<td>Score</td>
<td>Score</td>
</tr>
<tr>
<td>1</td>
<td>t5</td>
<td>0.0839</td>
</tr>
<tr>
<td>2</td>
<td>t6</td>
<td>0.0229</td>
</tr>
<tr>
<td>3</td>
<td>t7</td>
<td>0.0199</td>
</tr>
<tr>
<td>4</td>
<td>t1</td>
<td>0.0107</td>
</tr>
<tr>
<td>5</td>
<td>t8</td>
<td>0.0016</td>
</tr>
<tr>
<td>6</td>
<td>t2</td>
<td>0.0016</td>
</tr>
<tr>
<td>7</td>
<td>t3</td>
<td>0.0016</td>
</tr>
<tr>
<td>8</td>
<td>t4</td>
<td>0.0016</td>
</tr>
</tbody>
</table>

The rankings are all very similar to the or combination results. The alternative implementation of the and operator works as expected: it produces rankings identical to the correct or implementation; only the scores differ.

<table>
<thead>
<tr>
<th>Computation:</th>
<th>Correct</th>
<th>Optimized</th>
</tr>
</thead>
<tbody>
<tr>
<td>Combination:</td>
<td>Alternative</td>
<td>Correct</td>
</tr>
<tr>
<td></td>
<td>Score</td>
<td>Score</td>
</tr>
<tr>
<td>Recall:</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Precision:</td>
<td>(\frac{1}{8})</td>
<td>(\frac{1}{8})</td>
</tr>
<tr>
<td>Average precision:</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

The correct implementation in this case produces the highest precision; recall and average precision are unaffected by the choice of implementations.

4.1.3 Score Propagation Operators

The results of queries using propagation are influenced by the implementation of the computation operator. We have therefore four different rankings to examine. As with the computation operator, we examine two implementation variants:
– The *correct* variant returns every region, even if it is outside the containment relation specified by the propagation operator. Such a region receives a score of zero in accordance with the formal definition.

– The *optimized* variant leaves out regions that are outside the containment relation (i.e., regions that would receive a zero score).

In the following subsections, we look at upward and downward propagation separately.

**Upward Propagation**

To examine upward propagation, we used the following query:

```xml
//section[about(.//title, information retrieval)]
```

We consider section s2 to be relevant to this query. The query plan using the complex computation operator (α) is as follows:

\[
\begin{align*}
\text{section} & := \sigma_{n=section, t=node}(C) \quad (4.11) \\
\text{title} & := \sigma_{n=title, t=node}(C) \quad (4.12) \\
R_1 & := \alpha_{n=\text{title}, t=node}(\text{information.retrieval}) \quad (4.13) \\
R_2 & := \text{section} \triangleright \text{title} \quad (4.14)
\end{align*}
\]

This generates the following rankings:

<table>
<thead>
<tr>
<th>Computation: Propagation</th>
<th>Correct</th>
<th>Optimized</th>
<th>Correct</th>
<th>Optimized</th>
<th>Optimized</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rank</td>
<td>Score</td>
<td>Score</td>
<td>Score</td>
<td>Score</td>
<td>Score</td>
</tr>
<tr>
<td>1</td>
<td>s2</td>
<td>0.0955</td>
<td>s2</td>
<td>0.0955</td>
<td>s2</td>
</tr>
<tr>
<td>2</td>
<td>s5</td>
<td>-0.0054</td>
<td>s5</td>
<td>-0.0054</td>
<td>s5</td>
</tr>
<tr>
<td>3</td>
<td>s4</td>
<td>-0.0054</td>
<td>s4</td>
<td>-0.0054</td>
<td>s4</td>
</tr>
<tr>
<td>4</td>
<td>s3</td>
<td>-0.0054</td>
<td>s3</td>
<td>-0.0054</td>
<td>s3</td>
</tr>
<tr>
<td>5</td>
<td>s1</td>
<td>-0.0054</td>
<td>s1</td>
<td>-0.0054</td>
<td>s1</td>
</tr>
</tbody>
</table>

Note that the rankings for correct computation do not differ (① and ②). This is because the computation operator (4.13) returns every title, even those not containing any of the query terms. The titles not containing any terms are assigned a background score. The propagation operator (4.14) then does not leave out any sections, since every section contains a title with a non-zero score.

The rankings do differ in the case of the optimized computation implementation (③ and ④). In this case, the computation operator (4.13) returns only titles that contain one or more of the query terms. The correct propagation operator implementation ③ assigns zero scores to sections that do not contain any titles. The optimized propagation operator implementation ④ leaves out those sections, returning only the section that has a title that matches the query terms.

Precision and recall values for these rankings are as follows:
4.1. SMALL-SCALE TESTING

<table>
<thead>
<tr>
<th>Computation: Propagation</th>
<th>Correct</th>
<th>Optimized</th>
<th>Optimized</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Correct</td>
<td>Optimized</td>
<td>Optimized</td>
</tr>
<tr>
<td>Recall:</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Precision:</td>
<td>$\frac{1}{5}$</td>
<td>$\frac{1}{5}$</td>
<td>$\frac{1}{5}$</td>
</tr>
<tr>
<td>Average precision:</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**Downward Propagation**

To examine the downward operator implementation we use the following NEXI query:

```
//chapter[about(.,information retrieval)]//section[about(.,database)]
```

We consider the following sections to be relevant: $s_1$, $s_2$ and $s_3$. Section $s_5$ does contain the word *database*, however it is contained in an *appendix* element. Section $s_4$ is contained in a chapter about *information* and *retrieval*, however it does not contain the word *database*.

The resulting SRA query plan is as follows:

```
chapter := $\sigma_{n=\text{chapter},t=\text{node}}(C)$  \hspace{1cm} (4.15)
section := $\sigma_{n=\text{section},t=\text{node}}(C)$  \hspace{1cm} (4.16)
R_1 := $\alpha_{n=\text{chapter}}[\text{information.retrieval}](\text{chapter})$  \hspace{1cm} (4.17)
R_2 := $\alpha_{n=\text{section}}[\text{database}](\text{section})$  \hspace{1cm} (4.18)
R_3 := section ◄ chapter  \hspace{1cm} (4.19)
```

This generates the following rankings:

<table>
<thead>
<tr>
<th>Computation: Propagation:</th>
<th>Correct</th>
<th>Optimized</th>
<th>Optimized</th>
<th>Correct</th>
<th>Optimized</th>
<th>Optimized</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rank</td>
<td>Score</td>
<td>Score</td>
<td>Score</td>
<td>Score</td>
<td>Score</td>
<td>Score</td>
</tr>
<tr>
<td>1</td>
<td>$s_1$</td>
<td>+ 0.0038</td>
<td>+ 0.0038</td>
<td>$s_1$</td>
<td>+ 0.0038</td>
<td>+ 0.0038</td>
</tr>
<tr>
<td>2</td>
<td>$s_2$</td>
<td>+ 0.0017</td>
<td>+ 0.0017</td>
<td>$s_2$</td>
<td>+ 0.0017</td>
<td>+ 0.0017</td>
</tr>
<tr>
<td>3</td>
<td>$s_3$</td>
<td>+ 0.0008</td>
<td>+ 0.0008</td>
<td>$s_3$</td>
<td>+ 0.0008</td>
<td>+ 0.0008</td>
</tr>
<tr>
<td>4</td>
<td>$s_4$</td>
<td>- 0.0004</td>
<td>- 0.0004</td>
<td>$s_5$</td>
<td>- 0</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>$s_5$</td>
<td>- 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Because the correct implementation of the propagation operator \([4.19]\) assigns a zero score to elements outside the containment relation, rankings 1 and 2 contains section $s_5$ at the bottom of the ranking: $s_5$ does contain the word *database*, however it is contained in an *appendix* instead of a *chapter*. Section $s_4$, which does not contain the word *database*, is present in the rankings of the correct containment operator because it returns all regions: $s_4$ received a background score.

Precision and recall values for these rankings are as follows:

<table>
<thead>
<tr>
<th>Computation: Propagation:</th>
<th>Correct</th>
<th>Optimized</th>
<th>Optimized</th>
<th>Correct</th>
<th>Optimized</th>
<th>Optimized</th>
</tr>
</thead>
<tbody>
<tr>
<td>Recall:</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Precision:</td>
<td>$\frac{3}{5}$</td>
<td>$\frac{3}{5}$</td>
<td>$\frac{3}{5}$</td>
<td>$\frac{3}{4}$</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Average precision:</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
4.1.4 Conclusion

From the small-scale testing we can draw the following conclusions:

– The operators are functioning as expected: the correct implementations comply exactly with the formal SRA definitions;
– The fully optimized operator implementations produce rankings that have higher precision: they return fewer irrelevant regions;
– With computation and propagation, the optimized implementations do not reduce recall;
– The incorrect or operator reduces recall significantly. This is however only a problem when the computation operator is optimized.
– In general, when using the optimized variants of the operators, the precision of the entire ranking is improved: fewer irrelevant results are retrieved.

These small-scale tests are of course constructed to show the properties of the different operator implementations. Also, the relevance assessments are somewhat simplistic: if an element contains the query word we consider it relevant. Relevance assessment is more complicated on larger collections. We would like to know the effects on more real-life queries. To this end, we also performed a large-scale evaluation, reported in the next section.

4.2 Large-Scale Evaluation

For the large-scale experiments we performed we used the test collection and queries (topics) of the INitiative for the Evaluation of XML Retrieval (INEX), specifically, those used in the 2004 and 2005 conferences. [23, 22] give overviews of the methods used in this initiative. Our method is very similar to the one used in [25].

The INEX 2004 and 2005 test collections consist of the full text of scientific articles from a number of IEEE Computer Society publications. These articles are are supplied in XML markup. The INEX 2004 collection size is about 494 MB, the INEX 2005 collection is about 764 MB.

Test queries or topics are supplied by the INEX participants. These topics are based on information needs that might be expected in the real world based on the collection. The topics are also supplied in XML format. The information need represented by these topics is expressed in several ways. The title contains a NEXI query. The description gives a short summary of the information need, while the narrative gives an overview of the problem domain. Finally, a set of keywords is supplied. For INEX 2004 there were 40 content-only (CO) topics and 34 content-and-structure (CAS) topics. Of the 34 CAS topics, 26 have accompanying relevance assessments. In this case, we allow the system to execute all 34 topics, but we can only report retrieval effectiveness based on the 26 topics for which we have relevance assessments. INEX 2005 had 47 CAS topics, of which only 10 are used for relevance assessment.

We used this approach because INEX focuses on structured retrieval, which requires the full expressive power of SRA. For the same reason, we only examine the content-and-structure topics.
4.2. LARGE-SCALE EVALUATION

4.2.1 Measuring Retrieval Effectiveness – Recall and Precision

As we have seen before, the focus of information retrieval system evaluation is on the relevance of the results it returns. A retrieval system is considered to be of higher quality if it retrieves more relevant documents than another system, and if it ranks those relevant documents higher than irrelevant ones. Factors such as the time that it took to retrieve the results are usually not taken into account, because they are difficult to measure across different systems.

In IR, evaluation of retrieval effectiveness is typically performed using recall-precision graphs and average precision figures. Appendix A explains these concepts in detail. Recall for a given query is the ratio of the number of relevant documents retrieved to the number of relevant documents in the collection. Precision is the ratio of the number of relevant documents retrieved to the number of documents retrieved (irrelevant or relevant). By going down a result ranking and computing the recall and precision at every rank, a set of recall-precision combinations is formed. These values can be plotted in a recall-precision graph. By showing the curves for different systems in a single graph, the retrieval performance of these systems can be compared.

To evaluate retrieval effectiveness it is also useful to derive a single figure that indicates the effectiveness of a single figure. Average precision is often used for this. Average precision is determined by taking the average of the precision values at every relevant document in a ranking.

Using a single query to compare retrieval effectiveness of different systems is not scientifically correct. For a more consistent figure to use for comparison, the average precision values of several different queries are combined into a mean average precision (MAP) figure. These MAP figures are reported e.g. for the TREC evaluation initiative.

Because in structured retrieval, it is possible that (possibly nested) parts of documents are returned, the concept of relevance and the method of relevance assessment have to be modified slightly. We use the INEX assessment method here: when assessing elements for relevance to a query, these elements receive an assessment based on two dimensions: exhaustivity and specificity. These are both expressed on a scale of 0 to 3, where 0 is not relevant and 3 is highly relevant. At evaluation time, these two assessments are converted into a single relevance value on a scale of 0 to 1 using quantization functions: each quantization assigns a different weight to the two dimensions and their values. INEX 2004 used five different quantizations. For example, the strict quantization function only considers elements to be relevant that score 3 on both exhaustivity and specificity. The mean average precision (MAP) values over all topics based on each quantization are aggregated to form a single average MAP value. Appendix A contains a more detailed discussion of relevance assessments at INEX 2004. In later INEX conferences, more advanced approaches to evaluation were taken. INEX 2005 used the normalized extended Cumulated Gain method (nxCG), in addition to several other metrics such as the mean average effort-precision (MAep). Discussion of these concepts is outside the scope of this thesis; the interested reader is referred to [20].

In this thesis we report the evaluation results based on the generalized quantization. We report mean average precision (MAP) in addition to precision at three ranks (10, 25 and 50). For the INEX 2005 MAP we use the mean average effort-precision (MAep) values; for the precision at rank-values we used the normalized extended Cumulated Gain (nxCG) measure.
4.2.2 Experiments

In the previous chapter, we identified three classes of SRA operators that might be modified to return fewer intermediate regions. In the previous section we examined the computation operators ($\alpha$ and $\sqcap_p$), the combination operators and $\alpha$ ($\sqcap_p$ and $\sqcup_p$) and the operators for upward and downward propagation (↑ and ↓). For each of these operators, we can define two implementation variants. To be able to judge the merits of each of these implementations, we would have to run an experiment for each combination of these implementations. This would yield $2^5 = 32$ experiments. We also like to examine the difference between LMS and NLLR. We would need to run and evaluate a total of 64 experiments.

Fortunately, in the previous section we have seen that some combinations of operator implementations do not produce different results. The only operator that showed different results with the correct computation operator implementation was the downward propagation operator. We can therefore reduce the number of experiments somewhat: we now need to run 18 experiments for each retrieval model.

Our experiments for the LMS retrieval model are shown in table 4.2; the experiments for NLLR are identical. The run identifiers are defined according to the variables: cO in LMS, oC in LMS, aC in LMS, uC in LMS, dC in LMS. Means the Optimized implementation of the computation operator. We use the complex selection and computation operator ($\alpha$) for these experiments. For reference, we also perform two runs using the simple computation operator ($\sqcap_p$). These runs are also show in table 4.2, prefixed with ASP (for ASPECT, the internal name for this method of querying).

| Run identifier | Retrieval Model | Computation $\sqcap_p$ | Implementation variant for operator $\sqcup_p$ (or) $\sqcap_p$ (and) Propagation ▲ (up) ◄ (down) |
|----------------|-----------------|------------------------|-------------------------------------------------|--|--|
| LMS_cO_cO_cO_cO_cO_dC | LMS | correct | correct | correct | correct | correct |
| LMS_cO_cO_cO_cO_dO | LMS | correct | correct | correct | correct | optimized |
| LMS_cO_cO_cO_cO_dC | LMS | optimized | correct | correct | correct | optimized |
| LMS_cO_cO_cO_cO_dO | LMS | optimized | correct | correct | optimized | optimized |
| LMS_cO_cO_cA_cO_dC | LMS | optimized | correct | alternative | correct | optimized |
| LMS_cO_cO_cA_cO_dO | LMS | optimized | correct | alternative | optimized | optimized |
| LMS_cO_cO_cO_cA_dC | LMS | optimized | incorrect | correct | correct | optimized |
| LMS_cO_cO_cO_cA_dO | LMS | optimized | incorrect | correct | optimized | optimized |
| LMS_cO_cO_cA_cA_dC | LMS | optimized | incorrect | alternative | correct | optimized |
| LMS_cO_cO_cA_cA_dO | LMS | optimized | incorrect | alternative | optimized | optimized |
| ASP_LMS_cO_cO_cO_cO_dC | LMS (↓) | correct | correct | correct | correct | optimized |
| ASP_LMS_cO_cO_cO_dO | LMS (↓) | optimized | incorrect | correct | optimized | optimized |

Table 4.2: Experimental run identifiers and variables for the LMS model.

All the changes in the query process that are necessary for these experiments need to be implemented at the physical level. For example, the correct variant of the and implementation using product for
4.2. LARGE-SCALE EVALUATION

score combination, \texttt{and} \_\texttt{prod}, is supplemented with an alternative variant called \texttt{and} \_\texttt{sum} \_\texttt{alternative}. At query time, this variant can be selected by setting a configuration parameter from XQuery.

For each of the runs described above, we create an XQuery expression that executes all the CAS topics from INEX 2004 on the collection. These XQuery scripts produce results in the INEX submission XML format. We then use the \texttt{inex} \_\texttt{eval} tool to evaluate the results with the assessments. In this thesis, we only execute and evaluate the CAS queries, since the structural aspect is important in these queries.

Since we are not so much interested in determining the optimum settings for other variables, we take the following default settings:

- The \( \lambda \) parameter to the retrieval models is set to 0.5;
- The \texttt{and} combination operator \((\land_p)\) is implemented using product \(\otimes = \times\);
- The \texttt{or} combination operator \((\lor_p)\) is implemented using sum \(\oplus = +\);
- For upward propagation \((\uparrow)\) we use weighted sum;
- For downward propagation \((\downarrow)\) we use sum.

For experiments that describe the influence of these factors on the retrieval process, the reader is referred to [24].

4.2.3 Investigating Efficiency

To measure the gains in efficiency of our modifications at the physical level, we use the same collection and queries as for the retrieval performance experiments described in the previous section. However, because we are only interested in memory usage and execution time of the IR primitives – excluding the XQuery overhead – we will measure only the execution of the IR part of the queries.

In the previous chapter we demonstrated the architecture of our experimental system PF/Tijah (page 24). At the last stage of the NEXI compilation process, the logical SRA expression is translated to an MIL code fragment, which is then executed on the database. To capture only the IR part of the queries, we modify the compilation process so that the generated MIL code to be executed is also appended to a file. This file then contains the physical level query plans of several consecutive NEXI queries that were executed by a single XQuery expression. We can now execute this file on a fresh database process, while measuring memory usage and execution time. This way we ensure that these measurements are not affected by any XQuery overhead.

There are two aspects by which we measure efficiency:

- \textit{Memory usage}: we use the externally observable virtual memory (VM) size of the MonetDB database process. The virtual memory size includes the code size, data size, shared libraries and all memory that has been swapped to the disk.

  The VM size is easy to determine: on Unix platforms, this information is supplied by the \texttt{ps (1)} program. By determining the size of the VM size of the database process at fixed intervals (e.g. one second), we can create a memory usage graph and determine maximum and average memory usage.
– Execution time: we measure the time it takes for the entire run to complete. We derive the execution time from the memory usage data.

The time it takes for a computation intensive process to complete is in general influenced by the load on the experimental computer. To achieve consistent results, we perform each run three times and take the average completion time.

### 4.2.4 Results

Table 4.3 present an overview of some of the interesting runs from our experiments. For the results of all experiments the reader is referred to the tables in the appendix (page 79 and further).

Note that because INEX 2004 and INEX 2005 each uses their own set of metrics, results from INEX 2004 cannot be compared with results from INEX 2005. As we have stated before, the memory usage and execution time figures are averages over three separate runs. Analysis of the runs show that the three execution times for individual runs have a standard deviation of at most 1.4 seconds.

<table>
<thead>
<tr>
<th>Run identifier</th>
<th>Memory usage (MB)</th>
<th>Time (s)</th>
<th>MAP</th>
<th>Pr@10</th>
<th>Pr@25</th>
<th>Pr@50</th>
</tr>
</thead>
<tbody>
<tr>
<td>LMS_cC_oC_aC_uC_dC</td>
<td>1492</td>
<td>76</td>
<td>0.0442</td>
<td>0.40</td>
<td>0.34</td>
<td>0.25</td>
</tr>
<tr>
<td>LMS_cO_oC_aC_uC_dC</td>
<td>1244</td>
<td>54</td>
<td>0.0411</td>
<td>0.38</td>
<td>0.31</td>
<td>0.24</td>
</tr>
<tr>
<td>LMS_cO_oC_aC_uO_dO</td>
<td>1186 (894)</td>
<td>29</td>
<td>0.0409</td>
<td>0.38</td>
<td>0.32</td>
<td>0.25</td>
</tr>
<tr>
<td>LMS_cO_oI_aC_uO_dO</td>
<td>1210 (899)</td>
<td>28</td>
<td>0.0400</td>
<td>0.40</td>
<td>0.33</td>
<td>0.25</td>
</tr>
<tr>
<td>LMS_cO_oI_aA_uO_dO</td>
<td>1258</td>
<td>33</td>
<td>0.0406</td>
<td>0.40</td>
<td>0.32</td>
<td>0.24</td>
</tr>
</tbody>
</table>

**INEX 2004**

<table>
<thead>
<tr>
<th>Run identifier</th>
<th>Memory usage (MB)</th>
<th>Time (s)</th>
<th>MAP</th>
<th>Pr@10</th>
<th>Pr@25</th>
<th>Pr@50</th>
</tr>
</thead>
<tbody>
<tr>
<td>LMS_cC_oC_aC_uC_dC</td>
<td>1502</td>
<td>159</td>
<td>0.0387</td>
<td>0.37</td>
<td>0.31</td>
<td>0.23</td>
</tr>
<tr>
<td>LMS_cO_oC_aC_uO_dO</td>
<td>1487</td>
<td>143</td>
<td>0.0383</td>
<td>0.37</td>
<td>0.31</td>
<td>0.23</td>
</tr>
<tr>
<td>NLLR_cC_oC_aC_uC_dC</td>
<td>1508</td>
<td>73</td>
<td>0.0407</td>
<td>0.34</td>
<td>0.28</td>
<td>0.22</td>
</tr>
<tr>
<td>NLLR_cO_oC_aC_uC_dC</td>
<td>1244</td>
<td>54</td>
<td>0.0407</td>
<td>0.34</td>
<td>0.28</td>
<td>0.21</td>
</tr>
<tr>
<td>NLLR_cO_oC_aC_uO_dO</td>
<td>1187</td>
<td>28</td>
<td>0.0412</td>
<td>0.40</td>
<td>0.33</td>
<td>0.26</td>
</tr>
<tr>
<td>NLLR_cO_oI_aC_uO_dO</td>
<td>1234 (893)</td>
<td>29</td>
<td>0.0404</td>
<td>0.40</td>
<td>0.33</td>
<td>0.25</td>
</tr>
<tr>
<td>NLLR_cO_oI_aA_uO_dO</td>
<td>1258</td>
<td>33</td>
<td>0.0398</td>
<td>0.37</td>
<td>0.30</td>
<td>0.23</td>
</tr>
</tbody>
</table>

**INEX 2005**

<table>
<thead>
<tr>
<th>Run identifier</th>
<th>Memory usage (MB)</th>
<th>Time (s)</th>
<th>Gen. quant.; precision using nxCG</th>
</tr>
</thead>
<tbody>
<tr>
<td>LMS_cC_oC_aC_uC_dC</td>
<td>1618</td>
<td>44</td>
<td>MAP 0.0473 Pr@10 0.22 Pr@25 0.19 Pr@50 0.19</td>
</tr>
<tr>
<td>LMS_cO_oC_aC_uO_dO</td>
<td>1486</td>
<td>27</td>
<td>MAP 0.0473 Pr@10 0.23 Pr@25 0.19 Pr@50 0.20</td>
</tr>
<tr>
<td>LMS_cO_oI_aC_uO_dO</td>
<td>1531</td>
<td>26</td>
<td>MAP 0.0473 Pr@10 0.23 Pr@25 0.19 Pr@50 0.20</td>
</tr>
<tr>
<td>LMS_cO_oI_aA_uC_dO</td>
<td>1574</td>
<td>31</td>
<td>MAP 0.0476 Pr@10 0.23 Pr@25 0.19 Pr@50 0.20</td>
</tr>
<tr>
<td>NLLR_cC_oC_aC_uC_dC</td>
<td>1574</td>
<td>43</td>
<td>MAP 0.0484 Pr@10 0.20 Pr@25 0.17 Pr@50 0.19</td>
</tr>
<tr>
<td>NLLR_cO_oC_aC_uO_dO</td>
<td>1574</td>
<td>27</td>
<td>MAP 0.0484 Pr@10 0.23 Pr@25 0.19 Pr@50 0.20</td>
</tr>
<tr>
<td>NLLR_cO_oI_aC_uO_dO</td>
<td>1574</td>
<td>27</td>
<td>MAP 0.0484 Pr@10 0.23 Pr@25 0.19 Pr@50 0.20</td>
</tr>
<tr>
<td>NLLR_cO_oI_aA_uC_dO</td>
<td>1530</td>
<td>32</td>
<td>MAP 0.0489 Pr@10 0.23 Pr@25 0.19 Pr@50 0.20</td>
</tr>
</tbody>
</table>

Table 4.3: Experimental results

Based on these results we can make the following observations:
4.2. LARGE-SCALE EVALUATION

Utilizing optimized operator implementations increases speed and reduces memory usage. This effect is visible with both retrieval models and in both INEX 2004 and INEX 2005. The best example is the ‘most optimized’ run, in which all operators are optimized: run LMS_cO_oI_aC_uO_dO (Θ for INEX 2004 and Θ for INEX 2005).

In the INEX 2004 experiments, the ‘most correct’ run (all operators are implemented correctly) shows the best retrieval effectiveness (LMS_cC_oC_aC_uC_dC Θ). This run is also the slowest and memory-intensive.

In the INEX 2005 experiments, the differences in retrieval effectiveness between runs is not so pronounced, however it seems to work the other way around.

Using the complex computation operator (α) instead of the simple computation operator (⊔p) dramatically decreases the execution time. Compare for example run ASP_LMS_cC_oC_aC_uC_dC with LMS_cC_oC_aC_uC_dC Θ: the former uses the simple computation operator, the latter the complex computation operator. In addition, using optimized operator implementations together with the simple computation operator results in only a limited reduction in execution time.

The largest difference in retrieval effectiveness stems from the decision between the correct and optimized computation operator implementations. This can be seen in the difference in retrieval effectiveness between INEX 2004 runs LMS_cC_oC_aC_uC_dC Θ and LMS_cO_oC_aC_uC_dC Θ, where only the computation operator implementation is different.

After the computation operator, the use of optimized propagation operators seems to have the largest effect on efficiency. In all cases, runs with optimized implementations for both upward and downward propagation are faster. Compare for example INEX 2004 LMS_cO_oC_aC_uC_dC Θ and LMS_cO_oC_aC_uO_dO Θ: the first uses correct implementations of upward and downward propagation, the second uses optimized implementations.

The difference between retrieval effectiveness between ‘most correct’ and ‘most optimized’ runs is due to only some of the topics; see tables Θ for INEX 2004 and Θ for INEX 2005 in the appendix for a topic-by-topic analysis of these two extremes. Most topics show only minor differences or none at all.

There is a strong correlation between memory usage and execution time in the INEX 2004 runs. As memory usage goes down, so does execution time. The correlation coefficient ρ between average memory usage and execution time is 0.75 for the 2004 LMS runs. For the INEX 2005 experiments using LMS, the correlation coefficient is only 0.3: still somewhat correlated, but not as strongly.

The incorrect or operator implementation has only a limited influence on retrieval effectiveness. In the INEX 2004 runs, retrieval effectiveness seems to increase slightly when the incorrect implementation is used, contrary to our expectations. Compare INEX 2004 runs LMS_cO_oC_aC_uO_dO Θ and LMS_cO_oI_aC_uO_dO Θ in which only the or implementation differs. In the INEX 2005 runs, the effect on retrieval effectiveness is not visible.

The alternative and operator implementation does not produce significantly different results than the correct implementation. Execution times are increased. Compare for example INEX 2004 LMS_cO_oI_aC_uO_dO Θ and LMS_cO_oI_aA_uO_dO Θ, in which only the and implementation differs. As with the or implementation, the effect is not visible in the INEX 2005 runs.
The use of NLLR does not result in a decrease in execution time or memory usage: these are virtually identical to the LMS runs. Retrieval effectiveness is slightly higher for LMS in some cases and slightly higher for NLLR in others. Compare for example INEX 2004 runs $LMS_{CC,OC,AC,UC,DO}$ and $NLLR_{CC,OC,AC,UC,DO}$, where LMS has the advantage and $LMS_{CO,OC,AC,UC,DO}$ and $NLLR_{CO,OC,AC,UC,DO}$ where NLLR scores better. In both of these combinations, only the retrieval model differs. In the INEX 2005 runs, NLLR seems to perform better than LMS in most cases, however the difference is minimal.

Figure 4.2 shows a graph of the virtual memory usages of two INEX 2004 runs: the slowest and most memory-intensive run $LMS_{CC,OC,AC,UC,DO}$ and the fastest run $LMS_{CO,OC,AC,UC,DO}$. The graph shows that the former run has higher peaks in memory usage. It also takes longer to complete. Figure 4.3 shows the same runs using INEX 2005: the difference memory usage between the two runs is less pronounced there, however execution time is still reduced significantly.

Figure 4.4 show three recall-precision curves, showing the retrieval effectiveness of the ‘most correct’ run $LMS_{CC,OC,AC,UC,DO}$ and the ‘most optimized’ run $LMS_{CO,OC,AC,UC,DO}$ based on the INEX 2004 collection. In addition, the graph shows the curve that the TIJAH system achieved for INEX 2004. The graph shows that the difference in retrieval effectiveness between correct and optimized operator implementations is not as great as the difference between the PF/Tijah and TIJAH. This last difference is mostly due to the absence in PF/Tijah of advanced search techniques, such as relevance feedback and phrase search, which TIJAH did support.

![Figure 4.2: Virtual memory usage graph: INEX 2004 runs $LMS_{CC,OC,AC,UC,DO}$ and $LMS_{CO,OC,AC,UC,DO}$](image)

### 4.3 Conclusion

In this chapter we outlined the method by which we investigate the effect that optimizations at the physical level have on retrieval effectiveness (precision) and efficiency (memory usage and execution
Figure 4.3: Virtual memory usage graph: INEX 2005 runs LMS_cC_oC_aC_uC_dO and LMS_cO_oI_aC_uO_dO

Figure 4.4: Recall/precision curve using generalized quantization: INEX 2004 runs LMS_cC_oC_aC_uC_dC and LMS_cO_oI_aC_uO_dO and a submission to INEX 2004 by the TIJAH system.
time). We performed small-scale correctness testing of each of the SRA operators. We also performed a large-scale experiment, based on the method used at the INitiative for the Evaluation of XML Retrieval (INEX), in our case the 2004 installment. We have reported the results of these experiments.

In the next chapter, we draw conclusions based on these results and will try to answer the research questions.
Chapter 5

Conclusions and Recommendations

In this thesis we have examined the impact the physical level implementation of an IR system can have on retrieval effectiveness and performance. We described how optimized data structures and algorithms based on relational techniques can be employed to build a fast structured information retrieval system. In addition, we attempted to answer these research questions:

1. What steps can be taken at the physical level of an XML-IR system based on SRA to reduce intermediate result sizes during IR query execution?

   For these steps we would like to know:
   
   (a) what is the (expected and actual) effect on memory usage and execution speed?
   
   (b) what is the (expected and actual) effect on retrieval effectiveness?

2. What is the effect of introducing retrieval models that are not based on probabilities but on logarithmic likelihoods?

   (a) What is the effect on retrieval effectiveness?
   
   (b) Can these models be used to reduce intermediate result sizes?
   
   (c) How does this influence combination and propagation?

In chapter 3 we examined each of the SRA operators for opportunities for intermediate result size reduction. In chapter 4 we report on the results of small-scale tests and large-scale experiments to show the effects that these optimizations have on retrieval performance and efficiency. Based on these experiments, we can draw the following conclusions:

- Using the complex (‘coarse’) computation operator ($\alpha$, processing a set of terms at a time) instead of the simple computation operator ($\square_p$, processing each term separately) results in a dramatic increase of execution speed. When using optimized operator implementations, using the complex computation operator instead of the simple computation operator is more than five times faster. Using optimized operator implementations with the simple computation operator has only a limited effect. We were able to achieve retrieval effectiveness approaching that achieved by TIJAH at INEX 2004.
The use of an optimized computation operator – a retrieval model that leaves out regions that do not contain any of the query terms – results in a significantly faster system, especially when combined with optimized propagation operators. We expected this behavior based on our small-scale tests. In our large-scale experiments, this resulted in a system that was more than twice as fast, with only a minor reduction in retrieval effectiveness. Memory usage was also significantly reduced.

The choice of combination operator implementation is of marginal influence on efficiency, however it is interesting to note that choosing the ‘incorrect’ \texttt{or} implementation, which produces an intersection instead of the ‘correct’ union, does not diminish retrieval effectiveness appreciably. This is not what we expected based on our small-scale tests. The explanation for this limited influence might be found in the fact that only some of the topics use explicit \texttt{and} or \texttt{or} combinations.

The main research has been performed using the Language Modeling retrieval model, with background smoothing (LM or LMS). We also looked at the Normalized Log-Likelihood Ratio (NLLR) retrieval model to see if it might contribute to a further optimized system. We considered NLLR a good candidate because it naturally assigns a score of zero to regions that it considers irrelevant – regions that do not contain any of the query terms. With regard to the use of NLLR, based on our experiments, we can draw the following conclusions:

- When using NLLR, retrieval effectiveness is about the same as when using LMS. LMS has higher retrieval effectiveness in some cases: using correct implementations for all the operators in combination with LMS produces better retrieval effectiveness than can be achieved with NLLR. However, in some optimized cases, the NLLR model performs better than LMS.

- In our experiments, NLLR does not reduce intermediate results more than does the LMS retrieval model. This is due to the fact that NLLR and LMS both leave out exactly those regions that do not contain any of the query terms: this is the same set for each model. The decision is not based on score values.

- We could not show any effects that combination and propagation operators have on retrieval effectiveness using NLLR: there was too little variation between experimental runs to give any meaningful information on this point.

**Recommendations**

In our experiments, especially those using the INEX 2004 collection, we have seen a clear trade-off between retrieval effectiveness on one hand and efficiency – memory usage and speed – on the other. We can characterize the two ends of this spectrum as follows:

- To achieve a system that delivers the highest retrieval performance possible, use the Language Modeling retrieval model (LMS), with every operator implemented correctly. This system will not be the most efficient in terms of memory usage and execution speed. This can be used for e.g. experimental situations, where efficiency is less of an issue.

- To achieve a system that is faster and more memory-efficient, use optimized implementations of each operator. In this case, the decision for one of the retrieval models – LMS or NLLR – is not
that significant. This will result in a system that still shows adequate retrieval effectiveness. This can be used for e.g. demonstrations or production systems, where efficiency is more important. Our experiments show that the PF/Tijah system achieves retrieval effectiveness similar to the TIJAH system used for e.g. INEX 2004 and 2005. In some cases the results with PF/Tijah are slightly worse. The difference in retrieval effectiveness is caused among others by the following factors:

- *absence of priors*: research has shown that using for example the size of an element to compute a prior probability for that element can increase retrieval effectiveness significantly\[26, 19\]. Although priors were implemented in TIJAH we have not used them in our experiments since we did not yet have priors that worked correctly when using the NLLR model. We recommend that priors are added that support the NLLR retrieval model as well.

- *absence of more advanced IR techniques*: because our system uses the complex selection and computation operator (α) exclusively, some features such as term weighting, phrase search and negation have not been used. These features are at this time only available when using the simple computation operator (⊐). Yet more advanced features such as relevance feedback have not yet been implemented at all. We recommend therefore that advanced search techniques also be implemented on PF/Tijah, using both the simple and complex computation operators.

These features should be added to PF/Tijah to achieve an even more effective system; perhaps these features also give rise to more optimization possibilities.

In this thesis we could not fully investigate the effect that the use of the NLLR retrieval model has on the retrieval process. For example, the use of other combination and propagation functions might increase retrieval effectiveness when using NLLR.

This thesis shows again the power of using layered approaches in designing (database) systems. The use of an algebra at the logical layer makes reasoning about query semantics easier. The use of a separate physical layer with its own data model, algorithms and structures enables many optimizations. One can even choose to deviate from the formal semantics of the IR primitives to achieve even higher performance.

We also demonstrated that a small sacrifice in retrieval effectiveness can result in a large increase in efficiency of an IR system. The use of SRA and a layered system design allows the search parameters to be adjusted according to the needs of the user: for experimental evaluation, the trade-off can be shifted towards higher retrieval effectiveness, at the cost of some efficiency. For a production system, the administrator can choose higher performance at the cost of some effectiveness.
Bibliography


Appendix A

Information Retrieval Result Evaluation

A.1 Relevance Assessment and Evaluation in Traditional IR

Traditionally, retrieval quality is expressed using the concepts recall and precision. Recall for a given query is the ratio of the number of relevant documents retrieved to the number of relevant documents in the collection. Precision is the ratio of the number of relevant documents retrieved to the number of documents retrieved (irrelevant or relevant):

\[
\text{recall} = \frac{|\{\text{relevant documents}\} \cap \{\text{retrieved documents}\}|}{|\{\text{relevant documents}\}|}
\]

\[
\text{precision} = \frac{|\{\text{relevant documents}\} \cap \{\text{retrieved documents}\}|}{|\{\text{retrieved documents}\}|}
\]

See also figure A.1 for a diagrammatic explanation.

Suppose we have three systems A, B and C. For each of these systems we receive a list of ten documents in response to the same query. Suppose also that each system has the same collection: it contains 2000 documents, of which 12 are relevant to this query. The following table lists the result rankings for each of these systems. In a Venn diagram this is shown in figure A.1.

<table>
<thead>
<tr>
<th>Rank</th>
<th>System A</th>
<th>System B</th>
<th>System C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Relevant?</td>
<td>Precision</td>
<td>Recall</td>
</tr>
<tr>
<td>1</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>2</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>+</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>4</td>
<td>-</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>5</td>
<td>+</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>6</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>7</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>8</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>9</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Avg. prec.: 0.25 0.40 0.29

By looking at the list of retrieved documents that a system has produced for a given query, the precision at a certain rank can be determined. For example, for system A, when three documents have been
APPENDIX A. INFORMATION RETRIEVAL RESULT EVALUATION

**Recall**
\[
\text{recall} = \frac{|\{\text{relevant documents}\} \cap \{\text{retrieved documents}\}|}{|\{\text{relevant documents}\}|}
\]

**Precision**
\[
\text{precision} = \frac{|\{\text{relevant documents}\} \cap \{\text{retrieved documents}\}|}{|\{\text{retrieved documents}\}|}
\]

---

Figure A.1: Venn diagram of relevance, precision and recall
A.1. RELEVANCE ASSESSMENT AND EVALUATION IN TRADITIONAL IR

examined of which only one is relevant, the precision at rank 3 is \( \frac{1}{3} \) (\( \Theta \)). Note that the precision of the entire ranking is equal to the precision at the last rank: in this case, \( \frac{1}{5} \).

The recall at a certain rank is determined by dividing the number of relevant documents retrieved at that point by the number of relevant documents in the collection. Similar to precision, this can also be computed for the entire ranking: in the case of system A, the recall is \( \frac{1}{12} \) (\( \Theta \)).

The average precision for a ranking is determined by averaging the precisions at every relevant document:

\[
\text{Average precision} = \frac{\sum_{r=1}^{N} (P(r) \cdot \text{rel}(r))}{|\text{relevant documents}|}
\]

The function \( \text{rel}(r) \) returns 1 when the document at rank \( r \) is relevant, 0 otherwise. \( P(r) \) is the precision at rank \( R \).

In our example, the average precision for system A is \( \frac{1+\frac{1}{2}+\frac{1}{4}+\frac{1}{2}+\frac{1}{2}}{10} \approx 0.25 \) (\( \Theta \)). In contrast, the average precision of system B is much higher (\( \Theta \)). Note that the difference in average precision between systems A and B is only due to the order in which the results were ranked: system A and B have retrieved the same number of relevant documents – in other words, their recall is identical. System C returned more relevant documents – its recall is slightly higher – but it displayed these relevant documents lower in the ranking than did systems A and B. This is reflected in the average precision being lower than system B(\( \Theta \)).

Because average precision expresses both the ratio of relevant to irrelevant documents and the order in which these are returned, average precision figures are used frequently in IR retrieval evaluation to compare the performance of IR systems. Comparing the overall performance of IR systems based on a single query is not a good measure. To arrive at well-founded conclusions, several queries are run, their average precisions are determined, and the mean average precision over these queries is used to compare.

The precision and recall values of a certain query can be plotted in a graph. It is customary to express both recall and precision on a scale of 0 to 1. For example, the precision at recall point \( \frac{1}{12} \) for system A is \( \frac{1}{5} \) (at rank 4). For system B the precision at recall point \( \frac{1}{17} \) is 1 (at rank 2). Plotting the PR-curves for two retrieval systems in a single graph can show the difference in retrieval quality between those two systems for that query: when the line of system A is above the line of system B at every point, we can say that system A delivers better retrieval performance. See figure [A.2].

Note that in this example we have multiple precision values for some recall levels. Also, the recall levels are dependent on the number of relevant documents; when comparing results from different queries, the different recall values become a problem. Using interpolation, we can get a single precision value for a set of standard recall levels. For each recall value \( R \) in this set (e.g. \( \{0, 0.1, ..., 1\} \)), the precision \( P \) is now determined by the following expression:

\[
P(R) = \max(P'|R' \geq R \wedge (R', P') \in S)
\]

\( S \) is the set of precision-recall combinations generated by the query. This function returns for a certain recall level \( R \) the highest precision value \( P' \) that is associated with any recall level higher than \( R \). Using this function to interpolate the results shown in figure [A.2] we achieve an interpolated recall-precision graph, shown in figure [A.3]. It can now be seen from this graph that system B performs better than both A and C.
Figure A.2: Recall-precision graph for a single query

Figure A.3: Interpolated recall-precision graph for a single query
A.2 Relevance Assessment and Evaluation in Structured IR

The above assumes that the collection consists of documents that are returned entirely. Furthermore, it is assumed that relevance is a binary property: a document is relevant or it is irrelevant. These assumptions do not apply for systems where parts of documents can be returned. Consider a ranking that contains a section element \texttt{sec} and its parent ‘document element’ \texttt{article}. Using binary assessments of entire documents (as above), the article can be considered relevant. However, what to do with the \texttt{sec} element? Because it is part of the article, it can be argued that this is enough reason to consider the section relevant as well, without even looking at its contents. On the other hand, maybe the section discusses an entirely different subject than the subject for which the article was considered relevant.

To solve this issue, INEX has chosen to assess relevance of individual elements on two scales: specificity and exhaustivity. Relevance on specificity with respect to a subject indicates the extent in which the element is only about the subject. In other words, an element is not specific if it discusses many other subjects as well. Relevance on exhaustivity with respect to a subject indicates how many aspects of the subject an element it discusses. In other words, an element is not exhaustive if it discusses only a small part of the subject. See also figure A.4.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{relevanceassessment.png}
\caption{Relevance assessment at INEX: elements are assessed according to their relevance on specificity and exhaustivity.}
\end{figure}

The problem of determining relevance of overlapping elements can be solved relatively easily now. Suppose two elements in an ancestor-descendant relation are to be assessed: an article element, which contains a section element. The article can be more exhaustive than the section, but not more specific: an article can discuss the subject in more depth, but it also contains more on other subjects. Conversely, the section can be more specific than the article but not more exhaustive: it can discuss fewer subjects, but it cannot discuss more aspects of those subjects than the article.

Relevance on the specificity and exhaustivity dimensions is determined each on a four-point scale (0 to 3). the only limitation imposed is that an element that is absolutely not specific about the subject (specificity = 0) is also absolutely not exhaustive (exhaustivity = 0). For the purpose of evaluating a result ranking, these two dimensions have to be combined into a single relevance level for each element, on a continuous scale of zero to one. This is called quantization. For example, the strict quantization defined for INEX only regards as relevant elements with high specificity and exhaustivity. The generalized quantization assigns a continuous value, where exhaustivity is considered more important than specificity. There are several more quantizations, each assigning different importance to the two relevance dimensions.
For each quantization, an average precision figure can be determined, which shows the quality of a given result list. In addition, we can plot graphs such as in the previous subsection. By determining the average precisions of several quantizations and taking the mean, a figure that incorporates several quantizations is determined: the mean average precision (MAP) is reported in INEX 2004 participant papers. More information on the evaluation process used at INEX 2004 can be found in [23]. We also report the MAP figure for our experiments in this thesis. This facilitates comparing our results against e.g. those achieved by the TIJAH system at INEX 2004 [26].
Appendix B

Using PF/Tijah for IR Experiments

This appendix briefly describes the procedure we followed to perform the INEX 2004 and 2005 experiments reported in this thesis.

B.1 Loading the Document Collection

To perform experiments using the PF/Tijah system, it has to be installed first. PF/Tijah is a part of the MonetDB/XQuery distribution.

When PF/Tijah has been installed, the collection that is to be used has to be indexed by PF/Tijah. In our case, the INEX 2004 and 2005 collections required some processing: the following shell script fragment assembles all the individual article files into a single file per volume, replaces any named character entities (e.g. \&euml;) with numeric character entities and drops the DTD definition:

```bash
cd /local/os/Collections/inex-1.9
mkdir processed
# Find all files named volume.xml in the subdirectory xml
for f in 'find -L xml -name "volume.xml"' ; do
  # Generate a filename based on the path to this volume.xml
  # (e.g. xml/it/2002/volume.xml becomes xml-it-2002-volume.xml)
  FN='echo $f | tr / -'
  # Process the file, placing it in the subdirectory processed
  xmllint --path dtd --loaddtd --noent --dropdtd $f > processed/$FN
done
```

This processed collection can now be loaded into PF/Tijah using for example the following MIL code:

```mil
def
module(pftijah);
tj_init_collection("INEX2005sneng",
  new(str,str).insert(“stemmer”,"snowball-english"));
```

1At the time of this writing, loading XML documents into the IR index can also be done using XQuery functions, however this method was not yet available when the experiments reported in this thesis were performed.
var docs := new(str,str);
var path := "/local/os/Collections/inex-1.9/processed/";
docs.insert(path + "xml-it-2002-volume.xml", "xml-it-2002-volume.xml");
docs.insert(path + "xml-it-2001-volume.xml", "xml-it-2001-volume.xml");
docs.insert(path + "xml-it-1999-volume.xml", "xml-it-1999-volume.xml");
# ... other volume files ...
tj_add2collection("INEX2005sneng",docs,true);

This code, saved in a file load.mil, can be executed by running the following command:
Mserver load.mil

B.2 Performing Experiments

The following example is an XQuery expression that performs a series of NEXI queries on a prede-
fined collection.

\[
\text{(: This function determines for a given XML element the path to it from the}
\text{article root. This path expression is a valid XPath—expression, in the form}
\text{/article[1]/sec[1]/p[3] :)}
\]

\[
\text{declare function local:getINEXPath ( Snode as node() ) as xs:string }
\]
\[
\{ \text{let $pathelements := for $a in $node/ancestor-or-self::*}
\text{where ( name($a) != "books" and name($a) != "journal" )}
\text{return if ( name($a) = "article" ) then}
\text{"article[1]"}
\text{else}
\text{concat(}
\text{name($a),}
\text{"[",}
\text{count($a/preceding-sibling::*[name()=name($a)]) + 1,}
\text{"]"})
\text{)}
\text{return string-join( $pathelements, "/" )};}
\]

\[
\text{(: Name of the file that contains the topics that are to be executed. :)}
\]
\[
\text{let $topicfile := "/local/os/Experiments/topics.xml"}
\]

\[
\text{(: Find all topics in the topic file :)}
\]
\[
\text{let $topics := fn:doc($topicfile)//topic}
\]

\[
\text{(: Configuration of the IR subsystem. The meaning of these}
\text{attributes is explained below. :)}
\]
\[
\text{let $options := <TijahOptions}
\text{txtmodel_model="LMS”}
\]
B.2. PERFORMING EXPERIMENTS

```xml
<inex-submission participant-id="123"
  task="VCAS"
  topic-part="T"
  query="automatic"
  type="run"
  run-id="LMS_cCoCaCuCdC">  
  <description></description>

  (: Iterate over each topic in the topic file :) 
  for $topic in $topics  
    (: Find the text of the topic (NEXI query) :) 
    let $topic_text := $topic/text()  
    (: Find the topic number :) 
    let $topic_id := data($topic/@id)  

    (: Perform the IR query, returning a sequence of nodes :) 
    let $nodes := pf:tijah-query( $options, (), fn:string-join($topic_text, "") )  

    (: For each topic, we have to return a topic element :) 
    return <topic topic-id="{$topic_id}"> 
      (: Iterate over all nodes in the IR query result :) 
      for $node at $rank in $nodes  
        (: Find the DOI (unique document ID) of the article that 
```
contains the element :)  
let $doi :=$node/ancestor−or−self::article/doi/text()

( : Return the result element description: 
path to the result element, DOI and rank :) 
return <result>
  <path>/{local:getINEXPath( $node )}</path>
  <doi> ${doi} </doi>
  <rank> ${rank} </rank>
</result>
</topic>
</inex-submission>

Note that this example reflects the user interface of PF/Tijah at the time of the experiments; at the time of this writing, all functions have been moved to the tijah namespace. This query, when saved in a file (e.g. run1.xq) is compiled and run using for example the following command:

pf run1.xq | Mserver --dbinit="module(pftijah);"

This results in the following XML document:

```xml
<?xml version="1.0"?>
<inex-submission run-id="LMS_cCoCaCuCdC" task="VCAS"
  participant-id="123" query="automatic" type="run" topic-part="T">
  <description/>
  <topic topic-id="127">
    <result>
      <path>/article[1]/bdy[1]/sec[6]/p[1]</path>
      <doi>10.1041/M4019s−1995</doi>
      <rank>1</rank>
    </result>
    <!-- ... More topics and their results ... -->
    <result>
      <path>/article[1]/bm[1]/bib[1]/bibl[1]/bb[15]</path>
      <doi>10.1041/K0050s−1997</doi>
      <rank>1500</rank>
    </result>
  </topic>
</inex-submission>
```

INEX submissions are required to report for each result element the name of the file that element originated from. Since we cannot get this information from either Pathfinder or the IR querying extension, we return the unique document identifier (DOI) that is associated with every article element. Since each XML file in the INEX collection contains only a single article, we created a mapping, defined in an XML file between file names and article DOIs. To produce correct INEX submissions, we must then replace all the DOI attributes in the submission with the correct filename. This can be performed with XQuery, XSLT or with a custom script, resulting in the following file:
This file can then be used to determine the retrieval effectiveness for this experiment, using `inex_eval` (INEX 2004) and `EvalJ` (INEX 2005).

### B.3 Description of TijahOptions Attributes

Table B.1 lists the options used for these experiments. Some of these attributes, namely `mil_prelude_file` and `milplan_append` are not present in the released version of PF/Tijah. In addition, the `txtmodel_*comb` and `txtmodel_*prop` attributes work slightly different from the released version. Only `milplan_append` was absolutely necessary to be able to get the generated MIL plans; the other attributes made it possible to add and select different physical level implementations without having to recompile the system.
APPENDIX B. USING PF/TIJAH FOR IR EXPERIMENTS

<table>
<thead>
<tr>
<th>Attribute name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>txtmodel_model</td>
<td>Retrieval model to use (LMS or NLLR).</td>
</tr>
<tr>
<td>txtmodel_collectionLambda</td>
<td>Value of $\lambda$ in the retrieval model.</td>
</tr>
<tr>
<td>txtmodel_orcomb</td>
<td>or combination implementation to use.</td>
</tr>
<tr>
<td>txtmodel_andcomb</td>
<td>and combination implementation to use.</td>
</tr>
<tr>
<td>txtmodel_upprop</td>
<td>Upward propagation operator implementation to use.</td>
</tr>
<tr>
<td>txtmodel_down</td>
<td>Downward propagation operator implementation to use.</td>
</tr>
<tr>
<td>txtmodel_returnall</td>
<td>Specifies whether the retrieval model should return all elements (true) or only those that contain one or more of the query terms (false).</td>
</tr>
<tr>
<td>algebraType</td>
<td>Which algebra variant to use: COARSE2 (using complex computation operator) or ASPECT (simple computation operator).</td>
</tr>
<tr>
<td>scoreBase</td>
<td>Default region score (ZERO or ONE).</td>
</tr>
<tr>
<td>mil_prelude_file</td>
<td>When this attribute is specified, the indicated MIL file is executed before IR querying starts.</td>
</tr>
<tr>
<td>milplan_append</td>
<td>File to append generated MIL plans to.</td>
</tr>
<tr>
<td>use_equivalences</td>
<td>Specifies whether to use equivalence classes (true or false).</td>
</tr>
<tr>
<td>equivalence_class_*</td>
<td>Each instance of this attribute specifies an equivalence class, consisting of a comma-separated list of equivalent element names.</td>
</tr>
<tr>
<td>stem_stop_query</td>
<td>Specifies whether to perform stemming and stop word removal on the query.</td>
</tr>
<tr>
<td>collection</td>
<td>Name of the collection to use.</td>
</tr>
<tr>
<td>returnNumber</td>
<td>Number of elements to return to XQuery.</td>
</tr>
</tbody>
</table>

Table B.1: TijahOptions attributes
Appendix C

Detailed Experimental Results

This appendix contains the results of all our experiments, performed on the INEX 2004 and INEX 2005 collections.

Tables C.1 and C.2 show the results of all our experiments based on the INEX 2004 and 2005 collections, respectively. Tables C.3 and C.4 show experimental results for each separate topic from two runs, namely the ‘most correct’ and ‘most optimized’, based on the INEX 2004 and 2005 collections. Finally, tables C.5 and C.6 list the topics (NEXI queries) used in these experiments.
## APPENDIX C. DETAILED EXPERIMENTAL RESULTS

### INEX 2004 – LMS

<table>
<thead>
<tr>
<th>Run identifier</th>
<th>Memory usage (MB)</th>
<th>Generalized quantization</th>
<th>Time (s)</th>
<th></th>
<th></th>
<th></th>
<th>Pr@10</th>
<th>Pr@25</th>
<th>Pr@50</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASP.LMS.c0.cC.aC.uC.dC</td>
<td>Maximum</td>
<td>Average</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1502</td>
<td>989</td>
<td>159</td>
<td></td>
<td>0.0387</td>
<td>0.37</td>
<td>0.31</td>
<td>0.23</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ASP.LMS.c0.o1.aC.uO.dO</td>
<td>1487</td>
<td>974</td>
<td>143</td>
<td>0.0383</td>
<td>0.37</td>
<td>0.31</td>
<td>0.23</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LMS.c0.cC.aC.uC.dO</td>
<td>1492</td>
<td>983</td>
<td>76</td>
<td>0.0442</td>
<td>0.40</td>
<td>0.34</td>
<td>0.25</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LMS.c0.cC.aC.uC.dO</td>
<td>1508</td>
<td>984</td>
<td>74</td>
<td><strong>0.0442</strong></td>
<td><strong>0.40</strong></td>
<td><strong>0.34</strong></td>
<td><strong>0.25</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LMS.c0.cC.aC.uC.dC</td>
<td>1244</td>
<td>971</td>
<td>54</td>
<td>0.0412</td>
<td>0.38</td>
<td>0.31</td>
<td>0.24</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LMS.c0.cC.aC.uC.dC</td>
<td>1244</td>
<td>987</td>
<td>51</td>
<td>0.0413</td>
<td>0.38</td>
<td>0.31</td>
<td>0.24</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LMS.c0.cC.aC.uC.dC</td>
<td>1244</td>
<td>940</td>
<td>33</td>
<td>0.0411</td>
<td>0.38</td>
<td>0.32</td>
<td>0.24</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LMS.c0.cC.aC.uC.dC</td>
<td>1244</td>
<td>969</td>
<td>54</td>
<td>0.0411</td>
<td>0.38</td>
<td>0.31</td>
<td>0.24</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LMS.c0.cC.aC.uC.dC</td>
<td>1243</td>
<td>982</td>
<td>51</td>
<td>0.0412</td>
<td>0.38</td>
<td>0.31</td>
<td>0.24</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LMS.c0.cC.aC.uC.dC</td>
<td>1244</td>
<td>985</td>
<td>49</td>
<td>0.0411</td>
<td>0.38</td>
<td>0.32</td>
<td>0.25</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LMS.c0.cC.aC.uC.dC</td>
<td>1243</td>
<td>974</td>
<td>54</td>
<td>0.0412</td>
<td>0.38</td>
<td>0.32</td>
<td>0.24</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LMS.c0.cC.aC.uC.dC</td>
<td>1243</td>
<td>984</td>
<td>52</td>
<td>0.0412</td>
<td>0.38</td>
<td>0.32</td>
<td>0.24</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LMS.c0.cC.aC.uC.dC</td>
<td>1244</td>
<td>987</td>
<td>50</td>
<td>0.0409</td>
<td>0.41</td>
<td>0.32</td>
<td>0.24</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LMS.c0.cC.aC.uC.dC</td>
<td>1258</td>
<td>948</td>
<td>33</td>
<td>0.0406</td>
<td>0.40</td>
<td>0.32</td>
<td>0.24</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LMS.c0.cC.aC.uC.dC</td>
<td>1244</td>
<td>977</td>
<td>54</td>
<td>0.0411</td>
<td>0.38</td>
<td>0.32</td>
<td>0.24</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LMS.c0.cC.aC.uC.dC</td>
<td>1244</td>
<td>986</td>
<td>51</td>
<td>0.0412</td>
<td>0.38</td>
<td>0.32</td>
<td>0.24</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LMS.c0.cC.aC.uC.dC</td>
<td>1244</td>
<td>992</td>
<td>50</td>
<td>0.0404</td>
<td>0.41</td>
<td>0.33</td>
<td>0.25</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LMS.c0.cC.aC.uC.dC</td>
<td><strong>1210</strong></td>
<td><strong>899</strong></td>
<td><strong>28</strong></td>
<td><strong>0.0400</strong></td>
<td><strong>0.40</strong></td>
<td><strong>0.33</strong></td>
<td><strong>0.25</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### INEX 2004 – NLLR

<table>
<thead>
<tr>
<th>Run identifier</th>
<th>Memory usage (MB)</th>
<th>Generalized quantization</th>
<th>Time (s)</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>NLLR.c0.cC.aC.uC.dC</td>
<td>Maximum</td>
<td>Average</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1508</td>
<td>980</td>
<td>73</td>
<td></td>
<td>0.0407</td>
<td>0.34</td>
<td>0.28</td>
<td>0.22</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NLLR.c0.cC.aC.uC.dO</td>
<td>Maximum</td>
<td>Average</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1508</td>
<td>984</td>
<td>71</td>
<td></td>
<td>0.0412</td>
<td>0.36</td>
<td>0.29</td>
<td>0.23</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NLLR.c0.cC.aC.uC.dC</td>
<td>1244</td>
<td>966</td>
<td>55</td>
<td>0.0408</td>
<td>0.34</td>
<td>0.28</td>
<td>0.21</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NLLR.c0.cC.aC.uC.dO</td>
<td>1244</td>
<td>979</td>
<td>51</td>
<td>0.0425</td>
<td>0.37</td>
<td>0.30</td>
<td>0.23</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NLLR.c0.cC.aC.uC.dC</td>
<td>1312</td>
<td>1008</td>
<td>53</td>
<td>0.0367</td>
<td>0.29</td>
<td>0.25</td>
<td>0.19</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NLLR.c0.cC.aC.uC.dO</td>
<td>1258</td>
<td>940</td>
<td>33</td>
<td>0.0401</td>
<td>0.37</td>
<td>0.30</td>
<td>0.23</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NLLR.c0.cC.aC.uC.dC</td>
<td>1244</td>
<td>969</td>
<td>54</td>
<td>0.0407</td>
<td>0.34</td>
<td>0.28</td>
<td>0.21</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NLLR.c0.cC.aC.uC.dO</td>
<td>1243</td>
<td>981</td>
<td>51</td>
<td>0.0424</td>
<td>0.37</td>
<td>0.30</td>
<td>0.23</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NLLR.c0.cC.aC.uC.dC</td>
<td>1312</td>
<td>1009</td>
<td>52</td>
<td>0.0379</td>
<td>0.32</td>
<td>0.28</td>
<td>0.22</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NLLR.c0.cC.aC.uC.dO</td>
<td><strong>1187</strong></td>
<td><strong>894</strong></td>
<td><strong>28</strong></td>
<td><strong>0.0412</strong></td>
<td><strong>0.40</strong></td>
<td><strong>0.33</strong></td>
<td><strong>0.26</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NLLR.c0.o1.AU.uD.dC</td>
<td>1244</td>
<td>967</td>
<td>54</td>
<td>0.0416</td>
<td>0.34</td>
<td>0.28</td>
<td>0.21</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NLLR.c0.o1.AU.uD.dC</td>
<td>1244</td>
<td>978</td>
<td>51</td>
<td>0.0433</td>
<td>0.37</td>
<td>0.30</td>
<td>0.23</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NLLR.c0.o1.AU.uD.dO</td>
<td>1312</td>
<td>1009</td>
<td>52</td>
<td>0.0375</td>
<td>0.29</td>
<td>0.25</td>
<td>0.19</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NLLR.c0.o1.AU.uD.dO</td>
<td>1258</td>
<td>943</td>
<td>33</td>
<td>0.0398</td>
<td>0.37</td>
<td>0.30</td>
<td>0.23</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NLLR.c0.o1.AU.uD.dC</td>
<td>1244</td>
<td>969</td>
<td>54</td>
<td>0.0416</td>
<td>0.34</td>
<td>0.28</td>
<td>0.21</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NLLR.c0.o1.AU.uD.dO</td>
<td>1244</td>
<td>981</td>
<td>51</td>
<td>0.0432</td>
<td>0.37</td>
<td>0.30</td>
<td>0.23</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NLLR.c0.o1.AU.uD.dC</td>
<td>1312</td>
<td>1010</td>
<td>52</td>
<td>0.0381</td>
<td>0.33</td>
<td>0.27</td>
<td>0.21</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NLLR.c0.o1.AU.uD.dO</td>
<td><strong>1234</strong></td>
<td><strong>893</strong></td>
<td><strong>29</strong></td>
<td><strong>0.0404</strong></td>
<td><strong>0.40</strong></td>
<td><strong>0.33</strong></td>
<td><strong>0.25</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table C.1: Experimental results – INEX 2004
### INEX 2005 – LMS

<table>
<thead>
<tr>
<th>Run identifier</th>
<th>Memory usage (MB)</th>
<th>Gen. quant.; precision using nxCG</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Maximum</td>
<td>Average</td>
</tr>
<tr>
<td>LMS,cC,aC,uC,dC</td>
<td>1618</td>
<td>1066</td>
</tr>
<tr>
<td>LMS,cC,aC,uC,dO</td>
<td>1581</td>
<td>1065</td>
</tr>
<tr>
<td>LMS,cO,aC,uC,dC</td>
<td>1574</td>
<td>1052</td>
</tr>
<tr>
<td>LMS,cO,aC,uC,dO</td>
<td>1486</td>
<td>1051</td>
</tr>
<tr>
<td>LMS,cO,aA,uC,dC</td>
<td>1574</td>
<td>1057</td>
</tr>
<tr>
<td>LMS,cO,aA,uC,dO</td>
<td>1531</td>
<td>1058</td>
</tr>
<tr>
<td>LMS,cO,aC,uC,dC</td>
<td>1575</td>
<td>1053</td>
</tr>
<tr>
<td>LMS,cO,aC,uC,dO</td>
<td>1575</td>
<td>1050</td>
</tr>
<tr>
<td>LMS,cO,aC,uO,dC</td>
<td>1487</td>
<td>1056</td>
</tr>
<tr>
<td>LMS,cO,aC,uO,dO</td>
<td>1486</td>
<td>1057</td>
</tr>
<tr>
<td>LMS,cO,aA,uC,dC</td>
<td>1574</td>
<td>1051</td>
</tr>
<tr>
<td>LMS,cO,aA,uC,dO</td>
<td>1574</td>
<td>1051</td>
</tr>
<tr>
<td>LMS,cO,aA,uO,dC</td>
<td>1574</td>
<td>1057</td>
</tr>
<tr>
<td>LMS,cO,aA,uO,dO</td>
<td>1574</td>
<td>1064</td>
</tr>
<tr>
<td>LMS,cO,aC,uC,dC</td>
<td>1530</td>
<td>1058</td>
</tr>
<tr>
<td>LMS,cO,aC,uC,dO</td>
<td>1574</td>
<td>1052</td>
</tr>
<tr>
<td>LMS,cO,aC,uO,dC</td>
<td>1574</td>
<td>1057</td>
</tr>
<tr>
<td>LMS,cO,aC,uO,dO</td>
<td>1531</td>
<td>1061</td>
</tr>
</tbody>
</table>

### INEX 2005 – NLLR

<table>
<thead>
<tr>
<th>Run identifier</th>
<th>Memory usage (MB)</th>
<th>Gen. quant.; precision using nxCG</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Maximum</td>
<td>Average</td>
</tr>
<tr>
<td>NLLR,cC,aC,uC,dC</td>
<td>1574</td>
<td>1061</td>
</tr>
<tr>
<td>NLLR,cC,aC,uC,dO</td>
<td>1574</td>
<td>1060</td>
</tr>
<tr>
<td>NLLR,cO,aC,uC,dC</td>
<td>1486</td>
<td>1050</td>
</tr>
<tr>
<td>NLLR,cO,aC,uC,dO</td>
<td>1530</td>
<td>1050</td>
</tr>
<tr>
<td>NLLR,cO,aA,uC,dC</td>
<td>1574</td>
<td>1055</td>
</tr>
<tr>
<td>NLLR,cO,aA,uC,dO</td>
<td>1530</td>
<td>1056</td>
</tr>
<tr>
<td>NLLR,cO,aC,uC,dC</td>
<td>1530</td>
<td>1052</td>
</tr>
<tr>
<td>NLLR,cO,aC,uC,dO</td>
<td>1574</td>
<td>1049</td>
</tr>
<tr>
<td>NLLR,cO,aC,uO,dC</td>
<td>1618</td>
<td>1056</td>
</tr>
<tr>
<td>NLLR,cO,aC,uO,dO</td>
<td>1574</td>
<td>1059</td>
</tr>
<tr>
<td>NLLR,cO,aA,uC,dC</td>
<td>1570</td>
<td>1054</td>
</tr>
<tr>
<td>NLLR,cO,aA,uC,dO</td>
<td>1530</td>
<td>1048</td>
</tr>
<tr>
<td>NLLR,cO,aA,uO,dC</td>
<td>1574</td>
<td>1055</td>
</tr>
<tr>
<td>NLLR,cO,aA,uO,dO</td>
<td>1531</td>
<td>1061</td>
</tr>
<tr>
<td>NLLR,cO,aC,uC,dC</td>
<td>1618</td>
<td>1051</td>
</tr>
<tr>
<td>NLLR,cO,aC,uC,dO</td>
<td>1531</td>
<td>1047</td>
</tr>
<tr>
<td>NLLR,cO,aC,uO,dC</td>
<td>1618</td>
<td>1055</td>
</tr>
<tr>
<td>NLLR,cO,aC,uO,dO</td>
<td>1574</td>
<td>1059</td>
</tr>
</tbody>
</table>

Table C.2: Experimental results – INEX 2005
APPENDIX C. DETAILED EXPERIMENTAL RESULTS

<table>
<thead>
<tr>
<th>Topic</th>
<th>'most correct' LMS_\text{cC,oC,aC,uC,dC}</th>
<th>'most optimized' LMS_\text{cO,ol,aC,uO,dO}</th>
</tr>
</thead>
<tbody>
<tr>
<td>127</td>
<td>0.1737</td>
<td>0.1737</td>
</tr>
<tr>
<td>128</td>
<td>0.0406</td>
<td>0.0406</td>
</tr>
<tr>
<td>129</td>
<td>0.0280</td>
<td>0.0236</td>
</tr>
<tr>
<td>130</td>
<td>0.0142</td>
<td>0.0098</td>
</tr>
<tr>
<td>131</td>
<td>0.0679</td>
<td>0.0392</td>
</tr>
<tr>
<td>132</td>
<td>0.0465</td>
<td>0.0486</td>
</tr>
<tr>
<td>133</td>
<td>0.0088</td>
<td>0.0033</td>
</tr>
<tr>
<td>134</td>
<td>0.0310</td>
<td>0.0344</td>
</tr>
<tr>
<td>135</td>
<td>0.0253</td>
<td>0.0226</td>
</tr>
<tr>
<td>136</td>
<td>0.0338</td>
<td>0.0224</td>
</tr>
<tr>
<td>137</td>
<td>0.0408</td>
<td>0.0269</td>
</tr>
<tr>
<td>139</td>
<td>0.0556</td>
<td>0.0194</td>
</tr>
<tr>
<td>140</td>
<td>0.0212</td>
<td>0.0214</td>
</tr>
<tr>
<td>141</td>
<td>0.0211</td>
<td>0.0205</td>
</tr>
<tr>
<td>142</td>
<td>0.0003</td>
<td>0.0003</td>
</tr>
<tr>
<td>143</td>
<td>0.0086</td>
<td>0.0086</td>
</tr>
<tr>
<td>144</td>
<td>0.0001</td>
<td>0.0001</td>
</tr>
<tr>
<td>145</td>
<td>0.0639</td>
<td>0.0639</td>
</tr>
<tr>
<td>149</td>
<td>0.0484</td>
<td>0.0463</td>
</tr>
<tr>
<td>150</td>
<td>0.0169</td>
<td>0.0151</td>
</tr>
<tr>
<td>151</td>
<td>0.0740</td>
<td>0.0762</td>
</tr>
<tr>
<td>152</td>
<td>0.0739</td>
<td>0.0739</td>
</tr>
<tr>
<td>153</td>
<td>0.1574</td>
<td>0.1578</td>
</tr>
<tr>
<td>155</td>
<td>0.0016</td>
<td>0.0008</td>
</tr>
<tr>
<td>156</td>
<td>0.0924</td>
<td>0.0901</td>
</tr>
<tr>
<td>157</td>
<td>0.0029</td>
<td>0.0023</td>
</tr>
<tr>
<td>Average:</td>
<td>0.0442</td>
<td>0.0401</td>
</tr>
</tbody>
</table>

Table C.3: Individual topic results – INEX 2004: average precision per topic using generalized quantization. Shown are the ‘most correct’ ⚫ run and the ‘most optimized’ run ⚫ using the LMS retrieval model. We consider the figures shown in **bold** to be large differences; the figures shown in *italics* show only minor differences.
Table C.4: Individual topic results – INEX 2005: average ‘effort-precision’ per topic using generalized quantization. Shown are the ‘most correct’ run and the ‘most optimized’ run using the LMS retrieval model. We consider the figures shown in bold to be large differences; the figures shown in italics show only minor differences.

<table>
<thead>
<tr>
<th>Topic</th>
<th>‘most correct’ LMS,cC,oC,aC,uC,dC</th>
<th>‘most optimized’ LMS,cO,oI,aC,uO,dO</th>
</tr>
</thead>
<tbody>
<tr>
<td>253</td>
<td>0.0005</td>
<td>0.0005</td>
</tr>
<tr>
<td>256</td>
<td>0.1619</td>
<td>0.1631</td>
</tr>
<tr>
<td>257</td>
<td>0.0105</td>
<td>0.0105</td>
</tr>
<tr>
<td>260</td>
<td>0.0044</td>
<td>0.0044</td>
</tr>
<tr>
<td>261</td>
<td>0.0982</td>
<td>0.0977</td>
</tr>
<tr>
<td>264</td>
<td>0.0036</td>
<td>0.0036</td>
</tr>
<tr>
<td>265</td>
<td>0.0118</td>
<td>0.0114</td>
</tr>
<tr>
<td>270</td>
<td>0.0122</td>
<td>0.0122</td>
</tr>
<tr>
<td>275</td>
<td>0.1431</td>
<td>0.1432</td>
</tr>
<tr>
<td>284</td>
<td>0.0268</td>
<td>0.0268</td>
</tr>
<tr>
<td>Average</td>
<td>0.0473</td>
<td>0.0473</td>
</tr>
</tbody>
</table>
### Table C.5: INEX 2004 topics used for retrieval effectiveness evaluation in this thesis

<table>
<thead>
<tr>
<th>Topic</th>
<th>NEXI query</th>
</tr>
</thead>
<tbody>
<tr>
<td>127</td>
<td>//sec//(pfgc)[about (. , Godel Lukasiewicz and other fuzzy implication definitions)]</td>
</tr>
<tr>
<td>128</td>
<td>//article[about(., intelligent transport systems)]//sec[about(., on-board route planning navigation system for automobiles)]</td>
</tr>
<tr>
<td>129</td>
<td>//article[about(., atl, new book review bookshelf)]//sec[about(., database &quot;data warehouse&quot;) ]</td>
</tr>
<tr>
<td>130</td>
<td>//article[about(., p, object database)]//p[about(., version management)]</td>
</tr>
<tr>
<td>131</td>
<td>//article[about(., au,&quot;Jiawei Han&quot;)]/abs[about(., &quot;data mining&quot;) ]</td>
</tr>
<tr>
<td>132</td>
<td>//article[about(., abs, classification)]//sec[about(., experiment compare)]</td>
</tr>
<tr>
<td>133</td>
<td>//article[about(., /fn//tg//atl, Query) and about(./st, optimization)]</td>
</tr>
<tr>
<td>134</td>
<td>//article[(about(., &quot;phrase search&quot;) OR about(., &quot;proximity search&quot;) OR about(., &quot;string matching&quot;)) AND (about(.,tries) OR about(.,&quot;suffix trees&quot;) OR about(.,&quot;PAT arrays&quot;))]/sec[about(.,algorithm)]</td>
</tr>
<tr>
<td>135</td>
<td>//article[about(., atl, summaries)]//sec[about(., &quot;Internet security&quot;) or about(.,&quot;network security&quot;) ]</td>
</tr>
<tr>
<td>136</td>
<td>//bib[about(. , text categorisation) and about(., &quot;Support Vector Machines&quot; SVM)]</td>
</tr>
<tr>
<td>137</td>
<td>//article[about(., abs, &quot;digital library&quot;) or about(./hp1, &quot;digital library&quot;) ]</td>
</tr>
<tr>
<td>139</td>
<td>//article[(about(., /bb//au//snm, Bertino) or about(./bb//au//snm , Jajodia)) and about(./bb//atl, security model) and about(./bb//atl, - indexing model - object oriented model)]</td>
</tr>
<tr>
<td>140</td>
<td>//article[about(., xml)]//bdy//sec[about(.,&quot;information integration&quot; +web) or about(.,&quot;information exchange&quot; +web)]</td>
</tr>
<tr>
<td>141</td>
<td>//article[about(., java)]//sec[about(.,implementing threads)]</td>
</tr>
<tr>
<td>142</td>
<td>//abs[about(. ,database access using perl)]</td>
</tr>
<tr>
<td>143</td>
<td>//sec[about(.,stemming +information)]</td>
</tr>
<tr>
<td>144</td>
<td>//article[about(.,abs,Bioinformatics 'software systems')] and about(./p,Data Warehouse Multi-format Multi-type integration)]</td>
</tr>
<tr>
<td>145</td>
<td>//article[about(.,information retrieval)]//p[about(.,relevance feedback)]</td>
</tr>
<tr>
<td>149</td>
<td>//article[about(., /abs//kw//, &quot;genetic algorithm&quot;)]/bdy//sec[about(.,simulated annealing)]</td>
</tr>
<tr>
<td>150</td>
<td>//article[about(., animation)]//bdy//sec[about(., st, inverse kinematics)]</td>
</tr>
<tr>
<td>151</td>
<td>//article[about(., &quot;web search engine&quot;)//sec[about(., &quot;vector space model&quot;) ]</td>
</tr>
<tr>
<td>152</td>
<td>//article//p[about(.,&quot;linux &quot;word processing&quot;&quot;&quot;word processor&quot;&quot;&quot;office programs&quot;) ]</td>
</tr>
<tr>
<td>153</td>
<td>//article//bm//vt[about(.,&quot;phD student&quot;) OR about(.,&quot;phD candidate&quot;) ]</td>
</tr>
<tr>
<td>154</td>
<td>//article[about(./p,&quot;self organising feature map&quot;) and about(./fm//yr,2000)]//fig[about(./fgc,&quot;self organising map&quot;) ]</td>
</tr>
<tr>
<td>155</td>
<td>//article[about(., abs, &quot;spatial join&quot;)/bdy//sec[about(.,&quot;performance evaluation&quot;) ]</td>
</tr>
<tr>
<td>156</td>
<td>//article[about(., abs,-query &quot;query optimization&quot; -linear) and about(./bdy,newton +gradient hessian technique)]//bdy/*[about(.,+optimization -experiments) and (about(.,maximization) or about(.,minimization))]</td>
</tr>
<tr>
<td>Topic</td>
<td>NEXI query</td>
</tr>
<tr>
<td>-------</td>
<td>------------</td>
</tr>
<tr>
<td>253</td>
<td>//article[about(.//abs,evaluation &quot;usability experiment&quot; &quot;digital libraries&quot;)]/sec[about(., evaluation methodology measures &quot;usability testing&quot;)]</td>
</tr>
</tbody>
</table>