

Diploma Thesis

How to Teach Semantics to a Search Engine

Björn Jarisch

Zurich, February 24, 2006

Supervisor: Cristian Duda
Supervisor: Prof. Dr. Donald Kossmann



Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Abstract

Google and others have revolutionized the way we search for information in the Internet, Intranet, and on our desktop. This work addresses the criticism that while today's search engines can index the content of documents, they do not understand their semantics or take them into account during the evaluation of queries. It lists examples in which these search engines return wrong results and shows how to extend their capabilities in order to improve those results.

The two main contributions of this work towards that goal are the definition of a rule language to define semantics in a way that can be understood by search engines, and the design of a scheme to normalize documents created through the application of such rules in order to efficiently index and query them. It describes implementations of an indexer and a query processor based on these contributions and shows experiments that indicate that our approach is competitive with traditional information retrieval while increasing the quality of query answers.

Acknowledgements

First, I would like to thank Prof. Dr. Donald Kossmann for taking a chance on me and giving me the opportunity to write my diploma thesis, as well as one of my semester theses, in his research group.

I would also very much like to thank Cristian Duda, who supervised my thesis and wrote large parts of the code used for the experiments in Chapter 7, Marcos Vaz Salles and Dr. Jens-Peter Dittrich. Together with Prof. Dr. Kossmann, they created the groundworks this entire work is based on, and most of the additional ideas presented here are the result of meetings between the three of us. They also helped to make this work immeasurably better by providing me with suggestions, corrections and constructive criticism, and I am very grateful to both of them. Dr. Jens-Peter Dittrich also developed the initial version of the query processing code shown in Chapter 6, which I subsequently appropriated. Much of the content of this thesis is based on an unpublished manuscript [DDJ+05], written by the five of us, and any material quoted or adopted from this manuscript is not specifically marked as such.

I want to thank Irina Carabus and Peter Fischer for helping to improve the aforementioned manuscript, and Heidy Schümperlin for proofreading this work. All remaining mistakes are mine alone.

In closing, I wish to say that I could not have asked for a better environment to work in or better people to work with, and I am very proud of our achievements over the last six months.

Contents

Chapter 1	Introduction	1
1.1	Problem Description	1
1.2	First Example: Form Letter.....	1
1.3	Second Example: Versioning	2
1.4	Proposed Solution	2
1.5	Structure of Thesis	4
Chapter 2	Rules	5
2.1	Overview	5
2.2	Excluded Rule	6
2.3	Annotation Rule.....	6
2.4	Alternative Rule	7
2.5	Version Rule.....	8
2.6	Placeholder Rule.....	9
2.7	Field Rule	10
2.8	Versionref Rule	11
2.9	Applying Rules.....	12
2.10	Joining Rules	14
2.11	Rule Language Schema.....	15
Chapter 3	Normalized Documents	18
3.1	Overview	18
3.2	Variables and Conditions.....	19
3.3	File Format	19
Chapter 4	Indexing Normalized Documents	21
4.1	Overview	21
4.2	Intervals and Tuples	22
4.3	Positions	23
4.4	File Format	25
Chapter 5	Indexing Implementation	28
5.1	Overview	28
5.2	Reader Class	29
5.3	InstanceWriter Class.....	30
5.4	IndexWriter Class	30
Chapter 6	Query Processing Implementation.....	32
6.1	Overview	32
6.2	Processor Class	33
6.3	TupleReader Class.....	36
6.4	TupleMerger Class	36
6.5	TupleSkipper Class	38

Chapter 7	Experiments.....	41
7.1	Overview	41
7.2	Latex.....	42
7.3	Twiki.....	45
7.4	Email.....	49
Chapter 8	Related Work.....	52
8.1	Semantic Web	52
8.2	Colorful XML	52
8.3	PIX.....	52
8.4	Others.....	52
Chapter 9	Conclusion and Future Work.....	54
9.1	Conclusion.....	54
9.2	Future work	54

List of Figures

Form Letter Example	1
Versioning Example.....	2
Logical Workflow of Solution	3
Rule Classes	5
Excluded Rule Example.....	6
Annotation Rule Example	7
Alternative Rule Example	8
Version Rule Example	9
Timeline for Version Rule Example.....	9
Placeholder Rule Example	10
Field Rule Example	11
Versionref Rule Example.....	12
Tree Traversal Example	13
Join Example.....	15
Rule Language Schema	17
Normalization Example.....	18
Normalized Document Schema	19
Normalized Document Examples.....	20
Index Example	21
Merging Conditions Example.....	22
Tuple Example	22
Merging Tuples Example	22
Normalized Document Example	23
Index with Positions for Normalized Document Example	24
Hierarchical Positions Example.....	25
Skip Words in Normalized Document Example	25
Index Format.....	26
Tuples.bin Format without Positions	27
Indexing Classes and Workflow	28
IWriter Interface Members	29
Reader Members	29
InstanceWriter Members.....	30
IndexWriter Members	30
Query Classes and Workflow	32
Processor Members	33
ITupleEnumerator Members	33
ITupleEnumerator Classes	34
Parse Function	34
Left-Deep Tree Example.....	35
TupleReader Members.....	36
TupleMerger Members	36
TupleMerger Functions	37
MergePositions Function.....	38
TupleSkipper Members	39

TupleSkipper Functions	39
Latex as XML Example	42
Rules for Latex Experiments	42
Index Size and Indexing Time	42
Size of Original vs. Normalized Files	42
Queries and Number of Results	43
Running Times for All Results, with Positions	43
Running Times for one Result per Document, with Positions	44
Running Times for All Results, without Positions	44
Running Times for one Result per Document, without Positions.....	45
Twiki Example	45
Twiki as XML Example	45
Rules for Twiki Experiments	46
Index Size and Indexing Time	46
Size of Original vs. Normalized Files	46
Queries and Number of Results	47
Running Times for All Results, with Positions	47
Running Times for one Result per Document, with Positions	48
Running Times for All Results, without Positions	48
Running Times for one Result per Document, without Positions.....	48
Email as XML Example.....	49
Rules for Email Experiments	49
Index Size and Indexing Time	49
Size of Original vs. Normalized Files	50
Queries and Number of Results	50
Running Times for All Results, with Positions	50
Running Times for one Result per Document, with Positions	51
Running Times for All Results, without Positions	51
Running Times for one Result per Document, without Positions.....	51

Chapter 1

Introduction

1.1 Problem Description

Almost everybody uses Google, MSN Search, or a related search engine in order to find documents on the Internet. Recently, their derivatives have also become popular in order to search for documents on the desktop of a PC: Google Desktop, MSN Desktop Search, Apple's Spotlight, and the extended search capabilities of the forthcoming Microsoft Windows Vista operating system are only the most prominent players in this arena.

Part of their success of these engines can be attributed to the fact that search and more specifically *information retrieval* technology is mature. Indeed, the basic model of information retrieval has not changed in thirty years: Queries are run against an inverted index (i.e. a list of words and associated document ids) which is created by decomposing a set of documents into keywords [BR99]. In some cases, the structure of documents is also taken into consideration [ALP04].

The drawback to these approaches is that keywords and structure are *syntactic*. Most documents, however, are generated by applications and as such contain not only data but also the *semantics* of that data. These semantics are not understood by today's search engines and are therefore not taken into account during the evaluation of queries. As a result, the search engines potentially return documents that are not relevant or miss documents even though they are relevant. The following two examples will illustrate this problem.

1.2 First Example: Form Letter

Figure 1 shows a form letter and the table used as its data source. Such documents can be found frequently in office environments and are usually stored – and therefore indexed – as two separate files in the file system.

```
Dear <ref ref-name="salutation" /> <ref ref-name="name" />,
the weekly meeting is set for today, 2PM.

<column name="salutation">
  <cell>Mr.</cell>
  <cell>Mr.</cell>
  <cell>Ms.</cell>
</column>
<column name="name">
  <cell>Rick Blaine</cell>
  <cell>Stuart Alan Jones</cell>
  <cell>Sarah Connor</cell>
</column>
```

Figure 1: Form Letter Example

The user, however, sees not two files but a letter he sent to three recipients, and he expects search results to reflect his view. Today’s search engines fail in this regard. For example, a Boolean search for the keywords *meeting* and *blaine* fails because neither the form letter nor the data source contain both words. The user, however, considers the letter relevant because he did in fact invite Mr. Blaine to a meeting. A search for *jones* and *blaine*, on the other hand, returns the data source as a result, even though the user never sent a letter that contains both names.

1.3 Second Example: Versioning

Another example involves only one document, which is shown in Figure 2. The document contains information recording the progress from the initial version to the current version. To the user, the document consists of several different versions. The initial version, for example, contained the phrase “meeting next Friday”, which, through a series of edits, turned into “conference next Thursday (not Friday!)”.

```
There is going to be a
<deletion change-id="c001">meeting</deletion>
<insertion change-id="c001">conference</insertion>
next
<deletion change-id="c002">Friday</deletion>
<insertion change-id="c002">Thursday</insertion>
<insertion change-id="c003"> (not Friday!)</insertion>.
```

Figure 2: Versioning Example

A phrase search for “*conference next thursday*”, however, will return no results because the search engine cannot find these three words in sequence anywhere in the document. A keyword search for *meeting* and *thursday*, on the other hand, will return the document as a result, even though these two words do not appear together in any version of the document.

1.4 Proposed Solution

As we mentioned, today’s search engines return incorrect results because they do not understand the semantics of data. In the example of Section 1.2, they do not understand that the form letter *references* the data source. They also fail to understand that the three resulting letters, as well as the different revisions of the document in Section 1.3, are considered independent *variants* or *versions*.

These references, variants and versions are examples of having one or multiple views, or *instances*, of a document which can differ from the document itself. In order to get correct results, the user could use the application to generate these instances (e.g., the actual letters of the first example, or the different versions of the document in the second example) and let them be indexed by a search engine in place of the original document. However, this process might be tedious and, depending on the application, difficult or even impossible to accomplish.

Instead, we propose the definition of certain *rules* to express semantics in documents in a format that search engines can understand and can use to generate instances for those documents automatically. These rules can be defined once for an entire class of documents (e.g. all OpenOffice documents or all Twiki web pages), or separately for each document.

If one considers the possibility of a bulk letter with several thousand recipients, it becomes clear that simply generating and indexing the various instances of every document is not a realistic option. For one thing, the size and creation time of an index over such data

would likely be enormous. In addition, the user would not be served very well if he searched for the word *meeting* and as a result got thousands of nearly identical letters.

As a solution, we show how to create a *normalized document* that represents an entire set of instances in a compact format, how to index such a document, and how to evaluate queries using such an index.

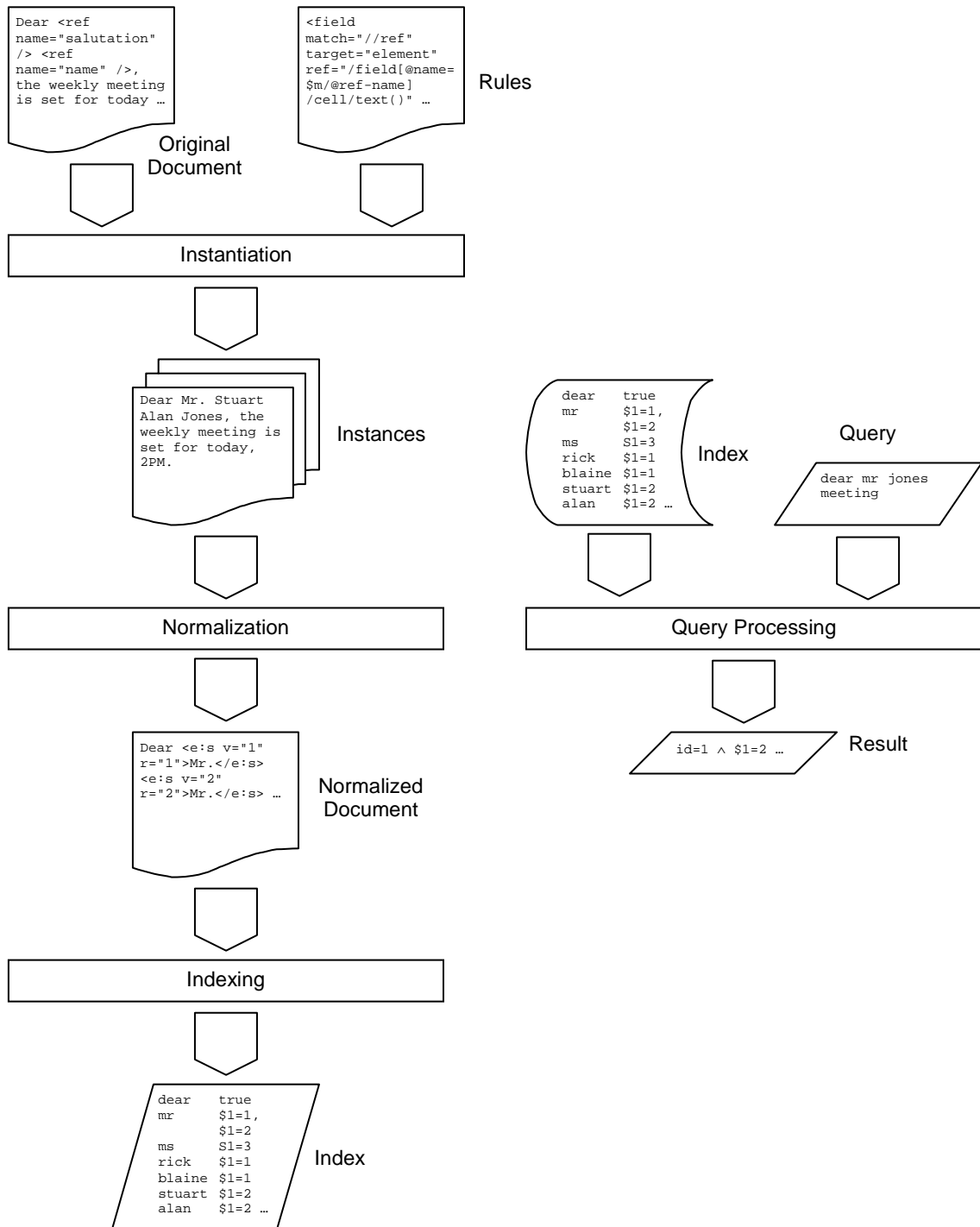


Figure 3: Logical Workflow of Solution

Figure 3 shows the logical workflow for the various parts of our solution, with the indexing steps on the left and the query processing step on the right: *Instantiation* is the creation of instances from a document using rules which explain its semantics. *Normalization* is the creation of a normalized document representing a set of instances. *Indexing* is the creation of an index based on set of normalized documents. *Query Processing* is the evaluation of a query based on an index. This workflow is only for illustration – in our implementation of the solution, instantiation and normalization are actually combined into a single step for performance reasons.

It should be noted that while our solution assumes that all documents are in XML format, this is done for reasons of simplicity – both of explanation and of design – and does not change the validity of the presented concepts. Furthermore, it is expected that many important document formats – such as the ones used by the Microsoft Office suite of products – will migrate to XML in next few years anyway.

1.5 Structure of Thesis

Chapter 2 introduces a simple yet powerful rule language that can be used to express semantics in documents. It will explain the different classes of rules and show examples of their use. Chapter 3 explains the concept of normalized documents, their advantages and their format. Chapter 4 shows how to create a compact inverted index based on normalized documents and describes our format for such an index. The implementations of an indexer and a query processor for such an index are presented in detail in Chapter 5 and Chapter 6, respectively. Chapter 7 shows the results of experiments we performed on different sets of application data that we converted to XML. Chapter 8 discusses related work and its connection to our work, and Chapter 9 contains the conclusion and directions for future work.

Chapter 2

Rules

2.1 Overview

As noted in Section 1.4, the purpose of rules is to express semantics in documents in a format that search engines can understand, and we assume that those documents are in XML format. A rule might express, for example, that `ref` elements in certain documents indicate a relation between the document and a referred data source, or that `insert` and `delete` elements indicate changes between different versions of a document. In doing so, rules describe the *transformation* that is supposed to be applied to those documents. Unlike transformation languages like XSLT, however, rules do not describe the transformation from one original document to *one* transformed document. Rather, they describe the transformation from the original document to *one or more* instances, such as the versions mentioned above.

In database terms, instances are *views* on the base data, and the rules are the *view definitions*. Indeed, all the work on XML views, XML data integration, and indexing of XML views is related (albeit mostly orthogonally) to our proposed solution. Rules, however, are different because they express the semantics of document content at a higher level of abstraction than traditional database operations like joins, sorting, and aggregation.

In our examples, there are simple patterns that occur repeatedly: The data can be distributed within a document or across documents and needs to be reassembled – like the form letter in Section 1.2 –, and the data can be versioned or annotated and different versions need to be extracted – like the versioned document in Section 1.3. Our rule language is specifically geared towards describing such patterns. It makes heavy use of XQuery and adopts the basic ideas of XSLT, but it is a drastic simplification that serves our purposes and can be optimized very well.

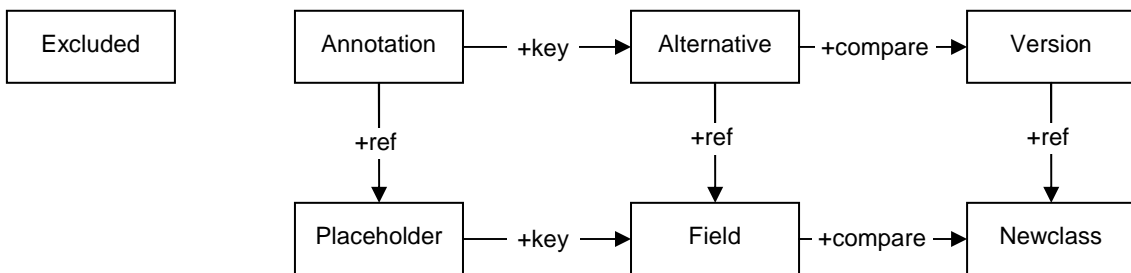


Figure 4: Rule Classes

As shown in Figure 4, there are seven classes of rules in our language, and each rule belongs to a certain class. All rules define which nodes in the document they apply to – those nodes are said to be *matching* the rule – and, depending on their class, have certain other attributes.

Sections 2.1 to 2.8 describe the seven classes and give an example for each class. In these examples, rules are only applied to element nodes, which are selected using XPath expressions. This restriction is only present in our implementation of the language – in general, rules can be applied to any kind of node defined in the XQuery 1.0 data model [FMM+05], and any XQuery expression can be used to select those nodes.

More than one rule can be applied to a document at the same time, and the classes and attributes of all applied rules define how many instances the original document is transformed into, and how the matching nodes are represented in each of those instances. This is explained in more detail in Section 2.9. It is also possible to correlate multiple rules using a join construct, which is explained in Section 2.10.

Finally, Section 2.11 shows the complete XSD schema for our rule language.

2.2 Excluded Rule

The simplest class of rules is *Excluded*. An Excluded rule indicates which nodes in a document should not be included in its instance. Examples are formatting instructions which may not be relevant for a search. A similar rule to exclude certain fragments from an XML document for phrase search has been proposed in the PIX project [AFS+03].

```
<font-declaration name="Lucida Grande" />
Remember next Friday's meeting.
```

(a) Original Document

```
<excluded match="//font-declaration" />
```

(b) Rule

```
Remember next Friday's meeting.
```

(c) Instance

Figure 5: Excluded Rule Example

Figure 5 shows a document containing a `font-declaration` element, a rule specifying that such elements should be excluded because they are not relevant for information retrieval, and the resulting instance without the declaration.

Excluded rules, like all rules, have a *match* attribute which contains an XQuery expression specifying the nodes in a document to which the rule applies. All rules also have a *description* attribute which can contain a description of the rule, but which is not relevant for processing.

2.3 Annotation Rule

The second class is *Annotation*. Examples for annotations are footnotes or comments in a text, which might be relevant for a search but still should not interrupt the flow of the surrounding content. Every Annotation rule defines two instances: One that is identical to the original document and one where all nodes matching the rule are excluded.

Rick Blaine <footnote id="ftn0">(the chair of the meeting)</footnote> has the flu.

(a) Original Document

```
<annotation match="//footnote" />
```

(b) Rule

Rick Blaine <footnote id="ftn0">(the chair of the meeting)</footnote> has the flu.

(c) First Instance

Rick Blaine has the flu.

(d) Second Instance

Figure 6: Annotation Rule Example

Figure 6 shows a document containing `footnote` elements, a rule defining such elements as annotations, and the two resulting instances.

Like Excluded rules, Annotation rules only have a *match* and a *description* attribute.

2.4 Alternative Rule

Alternative rules are a special form of annotations. They define not two but many instances and include each matching node in a document in only one of these instances. For example, a document could contain markup specifying which parts can be read by whom (e.g., management or public). Likewise, an electronic health record could contain markup that indicates what kind of information is available to which doctor.

```
Synopsis of Thursday's conference:
<conditional-text condition="public">The conference went very well.
</conditional-text>
<conditional-text condition="company">The conference was OK.
</conditional-text>
<conditional-text condition="management">The conference was OK.
</conditional-text>
<conditional-text condition="management">Note: We have to find a cheaper
venue! $50,000 is way too expensive.</conditional-text>
```

(a) Original Document

```
<alternative match="//conditional-text" key="$m/@value" type="string" />
```

(b) Rule

```
Synopsis of Thursday's conference:
<conditional-text condition="public">The conference went very well.
</conditional-text>
```

(c) Public Instance

```
Synopsis of Thursday's conference:
<conditional-text condition="company">The conference was OK.
</conditional-text>
```

(d) Company Instance

```
Synopsis of Thursday's conference:
<conditional-text condition="management">The conference was OK.
</conditional-text>
<conditional-text condition="management">Note: We have to find a cheaper
venue! $50,000 is way too expensive.</conditional-text>
```

(e) Management Instance

Figure 7: Alternative Rule Example

Figure 7 shows a document containing `conditional-text` elements, a rule defining such elements to be alternatives based on the value of the `condition` attribute, and the three resulting instances, with different degrees of information disclosure.

In addition to the *match* and *description* attributes, Alternative rules have a *key* attribute which contains an XQuery expression and a *type* attribute which specifies the data type of the expression result.

Alternative rules are evaluated in the following way: First, a variable called $\$m$ is bound to each matching node in turn and the key expression, which can refer to this variable, is evaluated. The result of this evaluation is called the *key value* for that node, and it is added to the rule's *key domain* if it is not already present. This is visible in the example, where the expression $\$m/value$ is evaluated for each node matching `//conditional-text`. Then, an instance is generated for each value in the key domain by excluding from the original document all matching nodes with a different key value. As a result, the number of instances defined by an Alternative rule is equal to the number of unique key values, which can be lower than the number of matching nodes.

Alternative rules also have an *optional* attribute. If it is set to *true*, an additional *empty* instance is generated, meaning an instance that does not contain any of the matching nodes.

2.5 Version Rule

Version rules are similar to Alternative rules. The difference is that each matching node in a document is included in a set of instances instead of only one instance, because the key equality condition of the Alternative rule is replaced by a comparison condition like *greater than* or *lower than*.

```
There is going to be a meeting next Friday.
<insertion change-id="c001">Time is not yet determined.</insertion>
<insertion change-id="c002">Keep the afternoon open.</insertion>
```

(a) Original Document

```
<version match="//insertion"
key="$m/@change-id" type="string" compare="greaterorequal" />
```

(b) Rule

There is going to be a meeting next Friday.

(c) First Instance

There is going to be a meeting next Friday.
<insertion change-id="c001">Time is not yet determined.</insertion>

(d) Second Instance

There is going to be a meeting next Friday.
<insertion change-id="c001">Time is not yet determined.</insertion>
<insertion change-id="c002">Keep the afternoon open.</insertion>

(e) Third Instance

Figure 8: Version Rule Example

Figure 8 shows a document containing `insertion` elements, a rule saying these elements should only be present in an instance if the value of their `change-id` attribute is greater or equal to the *key* value for the instance, and the three resulting instances.

Like Alternative rules, Version rules have a *match*, *description*, *key* and *type* attributes. In addition, they have a *compare* attribute which specifies how the key value for each node should be compared to each key in the key domain. In the example, the compare attribute is set to *greater or equal*, which means each matching node is present in an instance if its key value is greater or equal to the key value associated with that instance.

A Version rule creates one more instance in addition to those defined by its key domain: If its operator is *less*, the rule will not define an instance that includes all matching nodes unless we introduce an additional, *initial* instance with a key value *lower* than any value in the rule's key domain. Conversely, if the operator is *greater*, the rule will not define an instance that includes all nodes unless we add a *final* instance with a key value *greater* than any in the domain. If the rule's operator is *lessequal*, we instead add an instance with a lower key which *excludes* all matching nodes, and for *greaterorequal*, we add an instance with a greater key for the same reason. The timeline for the three instances of the example above, including the added initial instance with the lowest key value, is shown in Figure 9.

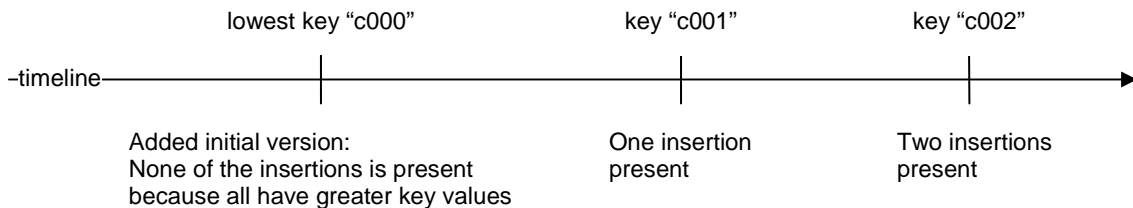


Figure 9: Timeline for Version Rule Example

Because the introduction of an extra instance guarantees that, depending on the operator, either the first or the last instance defined by the rule will be empty, the Version rule has no need for an *optional* attribute.

2.6 Placeholder Rule

Placeholder rules specify that data stored in a different location of the document or in a different file altogether should be inlined. Examples are bookmarks or `#include` directives in HTML documents.

```
<bookmark name="c2">Chapter 2</bookmark />
...
In <bookmark-ref refname="c2" />, we mentioned the meeting.
```

(a) Original Document

```
<placeholder match="//bookmark-ref" target="element"
  ref="//bookmark[@name=$m/@refname]/text()" />
```

(b) Rule

```
<bookmark name="c2">Chapter 2</bookmark />
...
In Chapter 2, we mentioned the meeting.
```

(c) Instance

Figure 10: Placeholder Rule Example

Figure 10 shows a document containing `bookmark-ref` elements referencing bookmarks within the document, a rule defining them as placeholders, and the resulting instance.

In addition to the *match* and *description* attributes, Placeholder rules have *ref*, *target* and *optional* attributes. The *ref* attribute contains an XQuery expression which specifies the data that should be copied. As with Alternative rules, it can refer to a $\$m$ variable bound in turn to each matching node. The *target* attribute can optionally be set to *element*. This specifies that the referenced data, instead of replacing the children of a matching node, should replace the node itself.

By default, a Placeholder rule defines exactly one instance. Even if the *ref* expression evaluates to multiple items, they are all included in the same instance. The optional attribute can optionally be set to *true* to specify that a second, empty instance be generated.

2.7 Field Rule

Field rules combine the behavior of the Alternative and Placeholder rules by replacing matching nodes in a document with other nodes, and then including each of the copied nodes in only one of multiple instances. They are useful to inline data stored in a different location if that data should also be split into multiple instances.

```
Dear <ref ref-name="salutation" /> <ref ref-name="name" />,
the weekly meeting is set for today, 2PM.
```

```
<column name="salutation">
  <cell>Mr.</cell>
  <cell>Mr.</cell>
  <cell>Ms.</cell>
</column>
<column name="name">
  <cell>Rick Blaine</cell>
  <cell>Stuart Alan Jones</cell>
  <cell>Sarah Connor</cell>
</column>
```

(a) Original Documents

```
<field match="//ref" target="element"
      ref="document('data.xml')/column[@name=$m/@ref-name]/cell/text()"
      key="position()" type="number" />
```

(b) Rule

Dear Mr. Rick Blaine, the weekly meeting is set for today, 2PM.

(c) First Instance

Dear Mr. Stuart Alan Jones, the weekly meeting is set for today, 2PM.

(d) Second Instance

Dear Ms. Sarah Connor, the weekly meeting is set for today, 2PM.

(e) Third Instance

Figure 11: Field Rule Example

Figure 11 shows the original documents from Section 1.2, a suitable rule, and the three resulting instances, each of which shows behavior similar to that of a Placeholder rule when applied to an instance defined by an Alternative rule.

Field rules have all the attributes of the Alternative and Placeholder rules with the same behavior. However, the key domain is determined in a different way: The `ref` expression is evaluated for each matching node by binding the variable `$m` to the node. Then, each node in the expression result is bound in turn to a variable `$r` and the key expression, which can reference both the `$m` and `$r` variables, is evaluated for each node. This is not visible in the example, where the `position()` function implicitly refers to each referenced node without the need for a variable.

2.8 Versionref Rule

Versionref rules combine the behavior of Version and Placeholder rules in exactly the same way that Field rules combine the behavior of Alternative and Placeholder rules. They are useful in cases where versioned data should be inlined and then included in instances depending on a comparison condition.

```
<deletion id="c003">Attendance is mandatory.</deletion>
...
There is going to be a conference next Thursday.
<change change-id="c003" />
```

(a) Original Document

```
<versionref match="//change" target="element"
           ref="//deletion[@id=$m/change-id]"
           key="$r/@id" type="string" compare="less" />
```

(b) Rule

```

<deletion id="c003">Attendance is mandatory.</deletion>
...
There is going to be a conference next Thursday.
<deletion id="c003">Attendance is mandatory.</deletion>

```

(c) First Instance

```

<deletion id="c003">Attendance is mandatory.</deletion>
...
There is going to be a conference next Thursday.

```

(d) Second Instance

Figure 12: Versionref Rule Example

Figure 12 shows a document containing `change` elements, a rule defining these elements to be references to `deletion` elements and those elements to be versioned content, and the two resulting instances.

Versionref rules have the same attributes as Version rules. In addition, they have a *ref* attribute with the same behavior as for the Field rule. Like Version rules, they define an additional instance to represent either the initial or the final version of the document.

2.9 Applying Rules

To actually create the instances defined by one or more rules, we decided to follow the same tree traversal approach that is used for XSL transformations: Instead of applying the rules to the original document in a certain order, we traverse the document tree depth-first. Each node we visit is processed according to the rule that the node matches, if any. Depending on the rule, we append the node, or other data, to the relevant instances. If the rule has a *ref* expression, we jump to the referenced nodes, traverse and process them, and then jump back to the original node and continue traversing.

With this approach, all expressions are evaluated against the original document, which is never modified, and all instances are only written to. This means that the application of one rule does not change the result of the match, *ref* or *key* expressions of any other rules, and we do not have to worry about the relative order of rule application.

```

<head>
  <row>Mr. Rick Blaine</row>
  <row>Ms. Sarah Connor</row>
</head>
<body>
  Dear <ref />,
  the weekly meeting is set for today, 2PM.
</body>

```

(a) Original Document

```

<excluded match="/head" />
<field match="/body/ref" target="element"
  ref="/head/row/text()" key="position()" type="number" />

```

(b) Rules

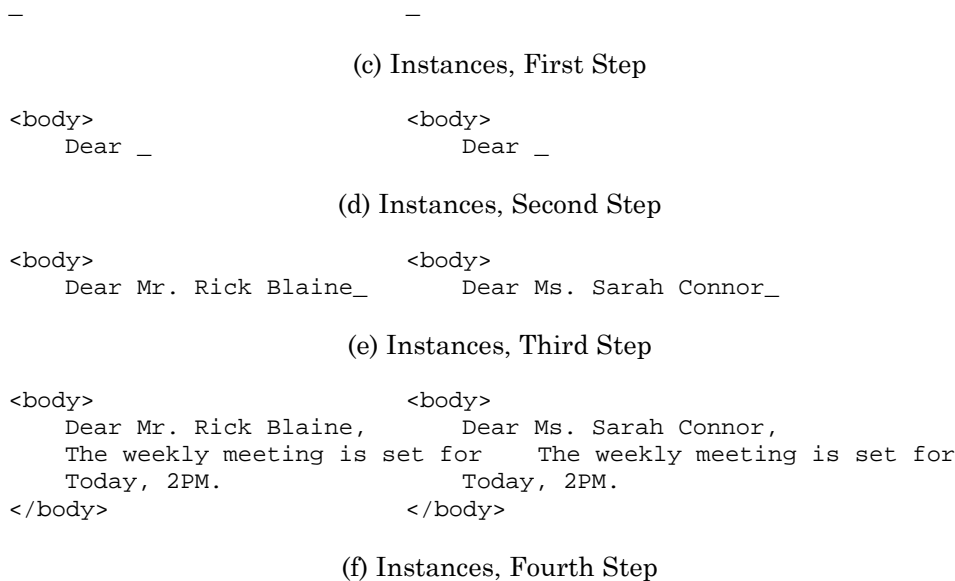


Figure 13: Tree Traversal Example

Figure 13 shows an example of such a tree traversal. The original document is divided into a `head` and `body` element, and the rules define that the entire `head` element should be excluded while the `ref` element in the `body` should be replaced with nodes from inside the `head`. In the first step, we traverse the document and get to the `head` element. Since it is matched by an Excluded rule, we do not write anything to the instances and jump to the next node. In the second step, we have stepped into the `body` and arrived at the `ref` element, all the while writing any node in the document that does not match any rule straight to both instances. In the third step, we have recognized that the `ref` matches a Field rule, jump to the referenced nodes, and write them to the correct instance as determined by their key values before jumping back. In the fourth step, we have reached the end of the original document and finish writing to the instances.

As the example shows, tree traversal avoids problems with rule ordering while, for example, allowing the Field rule to reference nodes that are then excluded from all generated instances. While we did evaluate the application of rules in a defined order, we found tree traversal to be more capable and robust.

Our tree traversal approach is not compatible with the notion of *composability* – applying more than one rule to the same node at once. However, composability does not add anything to the specific capabilities of our rule language, and we decided not to support it.

As Section 2.1 shows, most of our rules are built by adding an additional feature to an existing rule. These features are called *behaviors*, and there are four of them: *optional* defines whether to create an additional instance in which all matching nodes are excluded, *ref* defines whether to inline certain other data in place of the matching nodes, *key* defines whether to include the matching or referenced nodes in one instance or in one of many, and *compare* defines whether to use a comparison condition for the key values instead of an equality condition. With the exception of *compare*, which requires *key*, these behaviors are orthogonal to each other, and each possible combination is represented by a class of our rule language (with the exception of the useless trivial case, where no behaviors are defined and the single instance is identical to the original document). In addition, we have defined the Excluded class, which is the only class that cannot be built out of behaviors.

Because our rule classes already represent all possible compositions of different behaviors, we can disallow composability without negatively affecting the usability of the language, and we treat the match of any node to more than one rule as an error.

2.10 Joining Rules

We have seen that it is possible to apply more than one rule to a document at the same time. Usually, the total number of instances is then equal to the product of the number of instances defined by each rule on its own. If, for example, the form letter example from Section 2.7 contained some footnotes, we could define a matching Annotation rule in addition to the existing Field rule. The result would be six different instances – the first letter without footnotes, the first letter with footnotes, the second letter without footnotes, and so on.

However, for some rule combinations this is not the desired behavior. The *Join* construct allows multiple rules to be correlated by unifying their key domains. In other words, the key domain for each rule in a Join is set to the *union* of the key domains of all joined rules, with the condition that the data type of the key expression result is the same for all the rules. In database terms, a Join performs a *full outer join* operation on the joined rules' key domains.

```
There is going to be a
<deletion change-id="c001">meeting</deletion>
<insertion change-id="c001">conference</insertion>
next
<deletion change-id="c002">Friday</deletion>
<insertion change-id="c002">Thursday</insertion>
<insertion change-id="c003"> (not Friday!)</insertion>.
```

(a) Original Document

```
<join>
  <version match="//insertion" key="$m/@change-id"
    type="string" compare="greaterorequal" />
  <version match="//deletion" key="$m/@change-id"
    type="string" compare="less" />
</join>
```

(b) Joined Rules

```
There is going to be a
<deletion change-id="c001">meeting</deletion>
next
<deletion change-id="c002">Friday</deletion>
```

(c) First Instance

```
There is going to be a
<insertion change-id="c001">conference</insertion>
next
<deletion change-id="c002">Friday</deletion>
```

(d) Second Instance

```

There is going to be a
<insertion change-id="c001">conference</insertion>
next
<insertion change-id="c002">Thursday</insertion>.

```

(e) Third Instance

```

There is going to be a
<insertion change-id="c001">conference</insertion>
next
<insertion change-id="c002">Thursday</insertion>
<insertion change-id="c003"> (not Friday!)</insertion>.

```

(f) Fourth Instance

Figure 14: Join Example

As an example, Figure 14 shows the versioned document example from Section 1.3, two joined rules matching the `insertion` and `deletion` elements, and the four resulting instances representing the four versions of the original document. If the two rules were not joined, the result would have been twelve instances because the first rule on its own defines four instances, and the second rule defines three.

2.11 Rule Language Schema

The complete XSD schema our rule language is shown in Figure 15. Our implementation uses this schema to validate loaded rule configuration files before applying the rules to a document.

```

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="rules">
    <xs:complexType>
      <xs:choice minOccurs="0" maxOccurs="unbounded">
        <xs:element name="excluded" type="excludedType" />
        <xs:element name="annotation" type="annotationType" />
        <xs:element name="alternative" type="alternativeType" />
        <xs:element name="version" type="versionType" />
        <xs:element name="placeholder" type="placeholderType" />
        <xs:element name="field" type="fieldType" />
        <xs:element name="versionref" type="versionrefType" />
        <xs:element name="join" type="joinType" />
      </xs:choice>
    </xs:complexType>
    <xs:unique name="match">
      <xs:selector xpath="//*[@match]" />
      <xs:field xpath="@match" />
    </xs:unique>
  </xs:element>

  <xs:complexType name="excludedType">
    <xs:attribute name="match" type="xs:string" use="required" />
    <xs:attribute name="description" type="xs:string" />
  </xs:complexType>

```

```

<xs:complexType name="annotationType">
  <xs:attribute name="match" type="xs:string" use="required" />
  <xs:attribute name="description" type="xs:string" />
</xs:complexType>

<xs:complexType name="alternativeType">
  <xs:attribute name="match" type="xs:string" use="required" />
  <xs:attribute name="key" type="xs:string" default="position()" />
  <xs:attribute name="type" type="typeType" use="required" />
  <xs:attribute name="optional" type="xs:boolean" default="false" />
  <xs:attribute name="description" type="xs:string" />
</xs:complexType>

<xs:complexType name="versionType">
  <xs:attribute name="match" type="xs:string" use="required" />
  <xs:attribute name="key" type="xs:string" default="position()" />
  <xs:attribute name="type" type="typeType" use="required" />
  <xs:attribute name="compare" type="compareType" use="required" />
  <xs:attribute name="description" type="xs:string" />
</xs:complexType>

<xs:complexType name="placeholderType">
  <xs:attribute name="match" type="xs:string" use="required" />
  <xs:attribute name="ref" type="xs:string" use="required" />
  <xs:attribute name="target" type="targetType" default="content" />
  <xs:attribute name="optional" type="xs:boolean" default="false" />
  <xs:attribute name="description" type="xs:string" />
</xs:complexType>

<xs:complexType name="fieldType">
  <xs:attribute name="match" type="xs:string" use="required" />
  <xs:attribute name="ref" type="xs:string" use="required" />
  <xs:attribute name="target" type="targetType" default="content" />
  <xs:attribute name="key" type="xs:string" default="position()" />
  <xs:attribute name="type" type="typeType" use="required" />
  <xs:attribute name="optional" type="xs:boolean" default="false" />
  <xs:attribute name="description" type="xs:string" />
</xs:complexType>

<xs:complexType name="versionrefType">
  <xs:attribute name="match" type="xs:string" use="required" />
  <xs:attribute name="ref" type="xs:string" use="required" />
  <xs:attribute name="target" type="targetType" default="content" />
  <xs:attribute name="key" type="xs:string" default="position()" />
  <xs:attribute name="type" type="typeType" use="required" />
  <xs:attribute name="compare" type="compareType" use="required" />
  <xs:attribute name="description" type="xs:string" />
</xs:complexType>

<xs:complexType name="joinType">
  <xs:choice minOccurs="2" maxOccurs="unbounded">
    <xs:element name="alternative" type="alternativeType" />
    <xs:element name="version" type="versionType" />
    <xs:element name="field" type="fieldType" />
    <xs:element name="versionref" type="versionrefType" />
  </xs:choice>
  <xs:attribute name="type" type="typeType" use="required" />
  <xs:attribute name="description" type="xs:string" />
</xs:complexType>

```

```

<xs:simpleType name="targetType">
  <xs:restriction base="xs:string">
    <xs:enumeration value="element" />
    <xs:enumeration value="content" />
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name="typeType">
  <xs:restriction base="xs:string">
    <xs:enumeration value="boolean" />
    <xs:enumeration value="number" />
    <xs:enumeration value="string" />
    <xs:enumeration value="datetime" />
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name="compareType">
  <xs:restriction base="xs:string">
    <xs:enumeration value="less" />
    <xs:enumeration value="lessorequal" />
    <xs:enumeration value="equal" />
    <xs:enumeration value="greaterorequal" />
    <xs:enumeration value="greater" />
  </xs:restriction>
</xs:simpleType>

</xs:schema>

```

Figure 15: Rule Language Schema

Chapter 3

Normalized Documents

3.1 Overview

As mentioned in Section 1.4, a normalized document represents the instances of a document in a compact form by factoring out common content and reducing redundancy – a process known as *normalization*. Because the document retains all information about the instances it is based on, it can be used to create an index which is much more compact than one based on the instances themselves.

The idea underlying our approach to normalization is to *tag* content that is not common across all instances with a *condition* that specifies in which instances the content appears. This way, without losing any information, even content that is common to more than one instance has to be stored in the normalized document only once.

Dear Mr. Rick Blaine, the weekly meeting is set for today, 2PM.

(a) First Instance

Dear Mr. Stuart Alan Jones, the weekly meeting is set for today, 2PM.

(b) Second Instance

Dear Ms. Sarah Connor, the weekly meeting is set for today, 2PM.

(c) Third Instance

```
Dear
<condition value="instance 1 or 2">Mr.</condition>
<condition value="instance 3">Ms.</condition>,
<condition value="instance 1">Rick Blaine</condition>
<condition value="instance 2">Stuart Alan Jones</condition>
<condition value="instance 3">Sarah Connor</condition>,
the weekly meeting is set for today, 2PM.
```

(e) Simplified Normalized Document

Figure 16: Normalization Example

Figure 16 shows the example letters from Section 2.7 and a corresponding normalized document tagged in a very simple format. Information common to more than one instance is factored out, leading to a very compact normalized document except for the overly verbose tags.

A compact index can now be created by storing each piece of content in the normalized document together with any related tag – the original document, or the rules that were applied to it, are no longer required. Additionally, any instance that is represented by a nor-

malized document can be generated by simply excluding from the normalized document any content tagged with a condition the instance would not satisfy.

3.2 Variables and Conditions

In order to express conditions in a more structured format than the one shown in Section 3.1, we assign a numerical value to every rule, and to every instance defined by a rule: Every rule (or group of joined rules) that is applied to a document is assigned a different *variable*. Excluded rules and Placeholder rules with optional set to *false*, which define only one instance, are exempt. The variable is an integer with a range equal to the number of instances defined by the rule, and each value in the range represents one of the instances. Instances representing an empty instance or an initial version use the value zero, and all other instances are enumerated starting at one.

This allows us to refer to instances using integer values instead of key values of some other possible data type. The Version rule from Section 2.5, for example, has a variable with the range {0,1,2}, representing the key domain {"c000","c001","c002"}. This way, any condition imposed by a rule in our rule language can be expressed by a variable, a comparison operator and an integer value in the variable's range.

3.3 File Format

In order to use these conditions to efficiently tag content in a normalized document, we are using an element *s* – short for *switch* – in the namespace `urn:eth:semantic-search`. The element has two attributes *v* and *r*, which stand for *variable* and *range*, respectively. Its XSD schema is given in Figure 17.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  targetNamespace="urn:eth:semantic-search"
  elementFormDefault="qualified">
  <xs:element name="s">
    <xs:complexType mixed="true">
      <xs:choice minOccurs="0" maxOccurs="unbounded">
        <xs:any namespace="##any" processContents="skip" />
      </xs:choice>
      <xs:attribute name="v" type="xs:integer" use="required" />
      <xs:attribute name="r" type="xs:string" use="required" />
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Figure 17: Normalized Document Schema

We are using the short names *s*, *v* and *r* instead of *switch*, *value* and *range* to keep the normalized document as compact as possible. Normalized documents, like any XML document, can of course be compressed using any of a number of algorithms, but we did not implement this in our solution.

A switch's variable specifies the variable of an applied rule (or joined rules). Its range specifies the range for that variable, using either a single value in the variable's range or combining it with one of four strings representing comparison operators: *lt*, *le*, *ge* and *gt* stand for the operators *lower than*, *lower or equal*, *greater or equal* and *greater than*, respectively. The comparison *greater than 3*, for example, would be stored in the range as *gt3*. The combination of variable and range thus describes the condition that must be satisfied for the content inside the switch to be included in an instance. As an example of this format,

Figure 18 shows the normalized documents for the Field and Version examples from Sections 2.7 and 2.10.

```
Dear
<e:s v="1" r="1">Mr.</e:s>
<e:s v="1" r="2">Mr.</e:s>
<e:s v="1" r="3">Ms.</e:s>

<e:s v="1" r="1">Rick Blaine</e:s>
<e:s v="1" r="2">Stuart Alan Jones</e:s>
<e:s v="1" r="3">Sarah Connor</e:s>,
the weekly meeting is set for today, 2PM.
```

(a) Field

```
There is going to be a
<e:s v="1" r="lt1"><deletion change-id="c001">meeting</deletion></e:s>
<e:s v="1" r="ge1"><insertion change-id="c001">conference</insertion>
</e:s>
next
<e:s v="1" r="lt2"><deletion change-id="c002">Friday</deletion></e:s>
<e:s v="1" r="ge2"><insertion change-id="c002">Thursday</insertion></e:s>
<e:s v="1" r="ge3"><insertion change-id="c003"> (not Friday!)</insertion>
</e:s>.
```

(b) Join

Figure 18: Normalized Document Examples

It is of course possible that several such switches are nested inside each other, especially if multiple rules are applied to a document. In that case, the content inside the innermost switch is only present in an instance if the conditions of all nested switches are satisfied.

Chapter 4

Indexing Normalized Documents

4.1 Overview

Ordinarily, indexes for keyword search are implemented using an inverted file index, where each word is stored along with the sorted ids of the documents in which the word occurs. A query against such an index is run by merging the lists of document ids for all keywords in the query, and the result is a list of ids of documents containing the keywords.

For our solution, however, we would like to index normalized documents in such a way that the resulting, compact index still contains all information about the instances the normalized documents represent, in order to avoid having to index the instances themselves. For this, we propose the use of an enhanced inverted index that includes the *conditions* under which a word appears in a normalized document along with the other information. A query on such an index merges these conditions in addition to the document id lists, and the result is a list of document ids and conditions that define the actual document instances in which the words appear.

Such an index can be created from normalized documents by traversing each document and adding each visited word to the index together with the condition of the switch in which it appears. If the word appears in all instances and is therefore not inside a switch, it is added to the index with the trivial condition `true`. If, on the other hand, the word appears inside several nested switches as mentioned at the end of Section 3.2, it is added to the index together with the *conjunction* of the conditions of these switches. Finally, if the word appears multiple times inside the same document, it is added to the index once for each different condition under which it appears.

<u>word</u>	<u>id</u>	<u>conditions</u>
dear	1	true
mr	1	$\$1=1,$ $\$1=2$
ms	1	$S1=3$
rick	1	$\$1=1$
blaine	1	$\$1=1$
...		

Figure 19: Index Example

Figure 19 shows part of the index for the Field example from Section 3.3. The word *dear* is stored with the condition `true` because it appears in all instances while *mr* is stored with the conditions $\$1=1$ and $\$1=2$, using $\$1$ to refer to variable 1, because it appears once under each of these conditions. A search for the words *mr* and *blaine* against this index merges the conditions $\$1=1$ and $\$1=2$ with the condition $\$1=1$, and the resulting condition $\$1=1$ correctly identifies the first of the three letters as the only one to contain both words. A search

for *ms* and *blaine*, on the other hand, merges the conditions $\$1=3$ and $\$1=1$, with the result being *false* because no instance contains both of these words.

first condition	second condition	result
$\$1=1$	$\$1>=1 \wedge \$1<=3$	$\$1=1$
$\$1=2 \wedge \$2>3$	$\$1>=1 \wedge \$1<=3$	$\$1=2 \wedge \$2=3$
$\$1=2 \wedge \$2>3$	$\$2=3$	<i>false</i>
$\$1=4$	$\$2=3$	$\$1=4 \wedge \$2=3$

Figure 20: Merging Conditions Example

Figure 20 shows some more examples of merging conditions.

4.2 Intervals and Tuples

In the actual index format used in our implementation, conditions are not stored as strings like $\$1=2 \wedge \$2>3$. Instead, each condition is stored as a *tuple* and each tuple as a list of *intervals*, with one interval for each variable used in the condition. Each interval has open lower and upper bounds, called *start* and *end*, that define the range of values that will satisfy the condition. This representation is possible because, as mentioned in Section 3.2, all conditions consist of combinations of variables, operators and values, and conditions like $\$1>\2 are not possible.

As an example, the condition $\$1>=1$ is represented by a tuple with the interval $\$1 \in [1, 1]$, and the condition $\$1=2 \wedge \$2>3$ by a tuple with the intervals $\$1 \in [2, 2]$ and $\$2 \in [4, \infty]$. However, if the range for variable 2 is $\{0, 1, 2, 3, 4, 5\}$, that last interval can also be written as $\$2 \in [4, 5]$, which is the format we use because low numbers can be compressed more efficiently. The condition *true* is represented by an *empty* tuple containing zero intervals. Figure 21 shows some more examples of conditions and their corresponding tuples.

condition	tuple
$\$1<=1 \wedge \$2<3$	$\$1 \in [0, 1], \$2 \in [0, 2]$
$\$1=2 \wedge \$2>3$	$\$1 \in [2, 2], \$2 \in [4, 5]$
$\$1<5 \wedge \$3>=3 \wedge \$3<10$	$\$1 \in [0, 4], \$3 \in [3, 9]$

Figure 21: Tuple Example

Merging two tuples is defined as the creation of the tuple that represents the merging of the conditions that are represented by these tuples. Two tuples can therefore be merged unless they contain intervals for the same variable that do not intersect, and the result of the merge is a tuple that contains the intersected intervals in addition to any other intervals contained in one of the two tuples. Figure 22 shows the equivalent to Figure 20, this time using tuples.

first tuple	second tuple	result
$\$1 \in [1, 1]$	$\$1 \in [1, 3]$	$\$1 \in [1, 1]$
$\$1 \in [2, 2], \$2 \in [3, 5]$	$\$1 \in [1, 3]$	$\$1 \in [1, 1], \$2 \in [3, 3]$
$\$1 \in [2, 2], \$2 \in [3, 5]$	$\$2 \in [3, 3]$	<i>false</i>
$\$1 \in [4, 4]$	$\$2 \in [3, 3]$	$\$1 \in [4, 4], \$2 \in [3, 3]$

Figure 22: Merging Tuples Example

Before they are stored in an index, the tuples for each document and word are sorted by a *sort interval*, defined as the interval for the variable with the greatest range for that document. Tuples are sorted by ascending start value of that interval, with tuples without an interval for that *sort variable* being treated as having a sort interval from zero to plus infinity. Sorting the tuples this way allows query processors to use very efficient sweep line algorithms, such as the one described in Section 6.4, when merging two tuple lists, and using the variable with the greatest range aids performance by maximizing the selectivity of the merge.

4.3 Positions

To enable phrase search, an inverted index usually contains, in addition to words and document ids, a list of positions at which each word occurs inside a document, and the query processor uses this information to restrict the merging of document id lists to those documents where the words appear in sequence. These positions are normally generated by counting each word from the top of the document. PIX, for example, uses this approach. However, generating positions in this way it is not suitable for normalized documents, for reasons that we explain below.

```

Dear
<e:s v="1" r="1">Mr.</e:s>
<e:s v="1" r="2">Mr.</e:s>
<e:s v="1" r="3">Ms.</e:s>

<e:s v="1" r="1">Rick Blaine</e:s>
<e:s v="1" r="2">Stuart Alan Jones</e:s>
<e:s v="1" r="3">Sarah Connor</e:s>,
the weekly meeting is set for today<e:s v="2" r="1"> (Friday)</e:s>, 2PM.

```

Figure 23: Normalized Document Example

Using the normal approach for the normalized document in Figure 23, the word *Dear* has position 0. *Mr* and *Ms* have positions 1, 2 and 3, respectively, but depending on the value of variable 1, either one of them can follow *Dear*. Because this fact cannot be deduced from their positions, additional information would have to be stored in the index for phrase search to work. Simply storing a list of all possible succeeding words for each word is not feasible, considering the normalized document could be a form letter with thousands of recipients.

Our approach to this problem is to assign *identical* positions to all words that, depending on the condition, can succeed a word at the same position, as long as they do not appear together in any instance. Such words are said to be *parallel* to each other because the conditions that apply to them are disjoint and do not intersect. In the example, *Mr* and *Ms* can both be assigned position 1 because they follow *Dear* at position 0 and never appear in the same instance, and *Rick*, *Stuart* and *Sarah* can be at position 2 because they all follow a word at position 1.

The word *the* in the example poses another problem. Depending on the condition, it can follow either *Blaine*, *Jones* or *Connor*. These words now have the positions 3, 4 and 3, respectively, but there is no position that can be assigned to *the* that is the successor to all these positions at the same time.

Our solution is to use a form of *hierarchical* positioning which is designed so that different positions can have the same successor: Each word and switch at the top level of the document – meaning not inside any other switch – is assigned a sequential position with a *successor level* of 0. The successor level indicates the level of the succeeding position, and a

level of 0 means that the succeeding position is at the top level of the document. As with words, switches that succeed a word or switch at the same position are assigned identical positions as long as they are parallel to each other. Switches that contain no words are not assigned a position. In the example, the word *Dear* would still have position 0, the first three switches would all have position 1, the second three switches would have position 2, and the word *the* would have position 3.

Positions for words and switches inside a switch are assigned recursively by taking the position of the containing switch and *appending* the position of the word or switch, as counted from the start of the containing switch. Because words at the very first position inside a switch can be considered to be at the same position as the switch itself, any trailing zeroes in their positions are eliminated. In the example, the words *Stuart*, *Alan* and *Jones* would therefore be assigned the positions 2, 2.1 and 2.2, respectively, because they are nested in a switch with position 2.

The successor level for nested words and switches is determined as follows: Unless the word or switch is at the very last position inside the containing switch, its successor level is equal to the number of switches it is nested in – indicating that the successor is found at the same level of nesting. In the example, *Stuart* and *Alan* would both have a successor level of 1 because they are nested inside one switch and not at the last position. If the word or switch is at the last position, however, the successor level is defined as the level at which the next word or switch can be found. In the example, *Jones* is the very last word in the containing switch, and because the next word is *the* at the top level, *Jones* is assigned a successor level of 0.

word	id	tuple	position	successorlevel
dear	1	true	0	0
mr	1	$\$1 \in [1, 2]$	1	0
ms	1	$\$1 \in [3, 3]$	1	0
rick	1	$\$1 \in [1, 1]$	2	1
blaine	1	$\$1 \in [1, 1]$	2.1	0
stuart	1	$\$1 \in [2, 2]$	2	1
alan	1	$\$1 \in [2, 2]$	2.1	1
jones	1	$\$1 \in [2, 2]$	2.2	0
sarah	1	$\$1 \in [3, 3]$	2	1
connor	1	$\$1 \in [3, 3]$	2.1	0
the	1	true	3	0
weekly	1	true	4	0
meeting	1	true	5	0
is	1	true	6	0
set	1	true	7	0
for	1	true	8	0
today	1	true	9	0
friday	1	$\$2 \in [1, 1]$	10	0
2pm	1	true	11	0

Figure 24: Index with Positions for Normalized Document Example

Figure 24 shows the result of applying the described method of hierarchical positioning to the example. Because the successor to any hierarchical position is determined by truncating that position up to its successor level and incrementing its value at that level by 1, the words *Blaine*, *Jones* and *Connor* now all have the same successor *the* at position 3, even though they are at different positions. Conversely, *Alan* and *Connor* are both at position 2.1 but have different successors. Figure 25 shows some more positions, their successor levels and the resulting successors.

<u>position</u>	<u>successorlevel</u>	<u>successor</u>
5	2	5.0.1
5	3	5.0.0.1
5.4	1	5.5
5.4.7	1	5.5
5.4.0.1	1	5.5

Figure 25: Hierarchical Positions Example

One last problem arises when in certain instances, at certain positions, there is no word at all. This is the case in the example if the condition $\$2=1$ is not met – the word *today* is at position 9, and *2pm* is at position 11, but there is no word at position 10.

We solve this by assigning any position to the *empty string* – called a *skip word*, and henceforth denoted by (*skip*) – for any condition where there is no other word at that position. In the example, this would be the case for position 10 and the condition $\$2=0$. The tuple for a skip word at a certain position is chosen so that, in the relevant variable’s range, it represents the *complement* to the tuples of any switches at that position. The tuple for the skip word in the example is $\$2\in[0,0]$ because the tuple for the switch is $\$2\in[1,1]$ and in variable 2’s range of $\{0,1\}$, 0 is the complement to 1. As another example, if the range for variable 1 was $\{1,2,3,4,5\}$, there would be a skip word at positions 1 and 2 with the tuple $\$1\in[4,5]$ because $\{4,5\}$ would be the complement to $\{1,2,3\}$.

<u>word</u>	<u>id</u>	<u>tuple</u>	<u>position</u>	<u>successorlevel</u>
(skip)	1	$\$1\in[0,0]$	10	0

Figure 26: Skip Words in Normalized Document Example

Skip words and their positions and conditions can be stored in the index like ordinary words, as is shown in Figure 26. Adding skip words to the index like this allows phrase search to work even if the words in the phrase are not assigned consecutive positions. All that is necessary is that a phrase like “*today 2pm*” is also searched for as “*today (skip) 2pm*”, “*today (skip) (skip) 2pm*” and so on – in other words, any space between two words in the phrase is considered to contain an arbitrary number of skip words, including zero. In the example, a search for “*today 2pm*”, when treated as “*today (skip) 2pm*”, successfully returns a result at the positions 9 through 11 for the condition $\$2=0$.

If we add hierarchical positions and successor levels for each word and skip word to our index, we can support phrase search by merging not only the list of document ids and tuples for each word in a query, but also the lists of positions for each tuple. Merging these lists for two words in a phrase is done by checking, for each pair of positions, if the position for the word on the right is the successor to the position for the word on the left. In that case, the position on the right is included in the result of the merge and, if necessary, can be merged with the positions for another word even further to the right.

Positions for each word and tuple have to be sorted before they are added to the index to allow them to be merged using a modified sort-merge algorithm, such as the one described in Section 6.4. Our indexer implementation, described in Chapter 5, ensures this by already assigning the positions in properly sorted order.

4.4 File Format

Figure 27 shows the binary format we are using to store the index in Extended Backus-Naur form (EBNF). The index is stored in three separate files: `documents.bin`, `words.bin` and `tuples.bin`. All three files consist only of strings, 32-bit integers and 64-bit longs. The

files can be compressed by writing the integers and longs using 7-bit encoding [Mic05]. This reduces the size of `tuples.bin` significantly while still allowing for random access, which is important for our implementation of the query processor.

```
documents.bin := (id pathlength path sortvariable)+
id            := integer
pathlength   := integer
path         := string
sortvariable := integer
```

(a) Documents.bin Format

```
words.bin      := (wordlength word offset)+
wordlength     := integer
word           := string
offset         := long
```

(b) Words.bin Format

```
tuples.bin     := ((id tuples)+ 0)+
id             := integer
tuples        := tuplecount wordcount (tuple positions)+
tuplecount     := integer
wordcount     := integer
tuple         := intervalcount interval*
intervalcount := integer
interval      := variable start (end | 0)
variable      := integer
start        := integer
end          := integer
positions    := positioncount position+
positioncount := integer
position     := (levelcount level+ successorlevel) | (1 level)
level       := integer
successorlevel := integer
```

(c) Tuples.bin Format

Figure 27: Index Format

The file `documents.bin` contains information about the indexed documents, with a numerical document `id`, the complete `path` to the document and its `pathlength`, and the `sortvariable` for each document.

Information about the words in all indexed documents is stored in `words.bin`. For each word, it contains its `wordlength` in bytes, the word itself, and the `offset` in `tuples.bin` at which the tuples for that word are stored.

The file `tuples.bin` is the largest of the tree files because it contains information about which documents every word appears in, under what conditions, and in which positions. It contains, for each word, a sorted list of `id` of each document in which the word appears, followed by the `tuples` representing the conditions under which the word appears in that document. The list is terminated by a sentinel `id` with a value of 0, which allows us to determine when we have read all information for a certain word from `tuples.bin`. The tuples for a certain word and document `id` are stored as the `tuplecount` of the number of tuples for that word and document, the `wordcount` indicating how many times the word appears in the various instances of the document, followed by a sorted list of `tuple` and associated positions for each tuple. Each tuple is stored as the `intervalcount` followed by a list of

interval, and each interval is stored as its `variable`, `start` value and `end` value. If the `end` value is equal to the `start` value, it is instead stored as 0 to improve compression. The `positions` are stored as the `positioncount` followed by a sorted list of `position`, and each `position` is stored as the `levelcount`, a list of `level` and the `successorlevel`. The `levelcount` is defined as the depth of the position plus one, but for words at the top level of the document – where the depth is one and the successor level is zero – the successor level is omitted and the `levelcount` stored as 1 to save space.

If support for phrase search is not required, `tuples.bin` can also be stored in the more compact format listed in Figure 28. Because the tuples no longer contain associated lists of positions, any word which appears in a document with the condition `true` is stored in the index only with the tuple representing that condition, since any other conditions are more restrictive and therefore redundant.

```

tuples.bin      := ((id tuples)+ 0)+
id              := integer
tuples         := tuplecount wordcount tuple+
tuplecount     := integer
wordcount      := integer
tuple          := intervalcount interval*
intervalcount  := integer
interval       := variable start (end | 0)
variable       := integer
start          := integer
end            := integer

```

Figure 28: `Tuples.bin` Format without Positions

Our implementations of an indexer and query processor based on the format described here support both indexes with and without positions. They are described in Chapter 5 and Chapter 6. Experiments showing the expected tradeoff of support for phrase search versus index size and indexing speed are shown in Chapter 7.

Chapter 5

Indexing Implementation

5.1 Overview

Using Visual Basic.NET and the Microsoft .NET Framework 2.0, we implemented an indexer with the capability to generate normalized documents and indexes in the formats described in Sections 3.3 and 4.4. Its main classes and the resulting workflow are diagrammed in Figure 29.

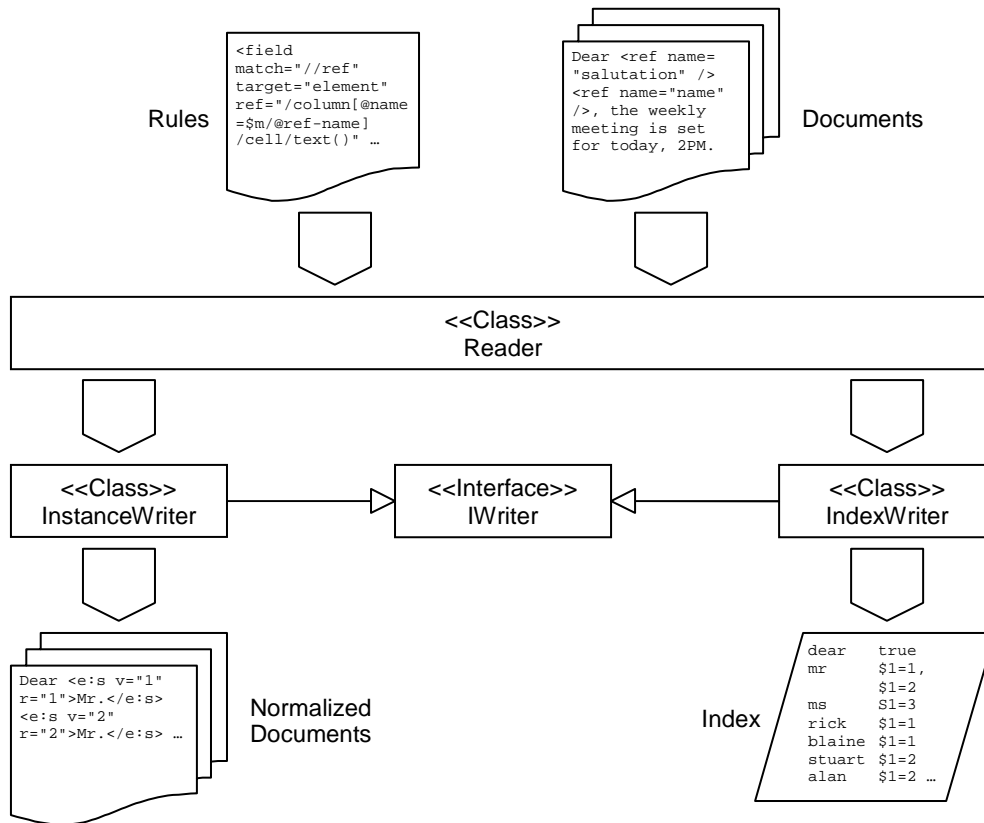


Figure 29: Indexing Classes and Workflow

As shown in the diagram, the indexer is split into two main parts: A Reader class that reads documents and rules, and the InstanceWriter and IndexWriter classes – both of which implement the IWriter interface – that write normalized documents and indexes.

The Reader class implements the instantiation and normalization steps of the logical workflow we described in Section 1.4. It does this without ever materializing a normalized

document. Instead, it sends a stream of *messages* representing the documents to a class implementing the `IWriter` interface by calling the methods defined in that interface – similar to a SAX-Parser [Meg04] but at a much higher level of abstraction. This results in higher performance than materializing the documents and then parsing them again for indexing or other purposes.

```
Sub StartDocument(fileName, variableRanges)
Sub EndDocument()
Sub StartIndent()
Sub EndIndent()
Sub StartSwitch(variable, value, compare)
Sub EndSwitch()
Sub Write(node)
```

Figure 30: IWriter Interface Members

Figure 30 shows the methods of the `IWriter` interface that are called by the `Reader`. The `Write`, `StartIndent` and `EndIndent` methods represent the content and hierarchy of a document, `StartSwitch` and `EndSwitch` represent the opening and closing tags of the switch elements described in Section 3.3, and `StartDocument` and `EndDocument` mark the beginning and end of each normalized document.

The messages are used by the `InstanceWriter` class to save the represented normalized documents as files, using the format described in Section 3.3. The `IndexWriter` class, on the other hand, uses them to generate an index in the format described in Section 4.4. As such, it implements the indexing step of the aforementioned workflow.

5.2 Reader Class

The `Reader` class implements the loading of rules from a configuration file, the loading of documents, and the sending of a message stream representing the normalized document for the currently loaded document and rules to a specified `IWriter`. Its members are shown in Figure 31.

```
Sub New()
Sub LoadRules(fileName)
Sub LoadFile(fileName)
Sub WriteTo(writer)
```

Figure 31: Reader Members

The `Reader` combines the tree traversal approach from Section 2.9 with the normalization idea from Section 3.1 in order to combine instantiation and normalization into a single step. It still traverses the document tree and processes each visited node according to any matching rule, interpreting that rule according to the descriptions in Chapter 2. However, instead of being appended to any instances, the result is wrapped in a switch element with a condition representing these instances, and this fragment of a normalized document is then sent out to an `IWriter` class as a stream of messages. If the instances *do* need to be materialized, they can simply be generated from the normalized document as mentioned towards the end of Section 3.1, but this functionality is not part of the `Reader` class.

It is possible to send the same stream of messages to several `IWriter` classes – for example to materialize a normalized document as well as index it at the same time – because information like matching nodes and key domains is only generated once and then cached until the loaded document or set of rules changes.

5.3 InstanceWriter Class

The `InstanceWriter` class implements the `IWriter` interface. It uses the messages it receives from a `Reader` to first construct the represented normalized document in memory and then save it as a file in a location specified by the user. Its constructor is shown in Figure 32.

```
Sub New(outputFolder)
```

Figure 32: InstanceWriter Constructor

The files produced by an `InstanceWriter` are not required for index generation but can be used for query result presentation as described in Section 6.1. It is possible, however, to create an index from a set of normalized documents by applying Alternative rules matching the switch elements in those documents, in case the original documents are no longer available.

5.4 IndexWriter Class

The `IndexWriter` class implements the `IWriter` interface and uses the messages it receives to create an index in the format specified in Section 4.4. Its additional members are shown in Figure 33.

```
Sub New(outputFolder, options)
Sub Save()
```

Figure 33: IndexWriter Members

When the `IndexWriter` is instantiated, the user can specify the location where the new index should be stored as well as options which include whether to remove *stop words* – common words in the English language such as “the”, “for” and “is” – from the index, whether to store positions, and whether to save the index in a compressed or uncompressed format. Our implementation does not support appending to an existing index, even though the format of the index would allow this, and this feature is listed in Section 9.2 as a possible avenue for future work.

To be able to construct the index, the `IndexWriter` must determine in which documents, under what conditions, and at which positions each word appears. To this end, the `IndexWriter` uses the `StartDocument` and `EndDocument` messages to construct a list of document names and ids. It uses the variable ranges specified in the `StartDocument` message to designate the variable with the greatest range as the sort variable for the current document. The `IndexWriter` also maintains a stack of currently relevant conditions by pushing a new condition on the stack for each `StartSwitch` message and popping the last condition off the stack with each `EndSwitch` message.

Every time a `Write` message containing a text node is received, the node’s value is converted to lowercase and split into an array of words, using a regular expression that matches XML entities and characters with a nonword Unicode category. Removing XML entities from the indexed content in this way is important to ensure that phrase search works properly. The phrase “This is Plug&Play”, for example, potentially stored in XML as `This is Plug&Play`, should be recognized as {“this”, “is”, “plug”, “play”}, not as {“this”, “nbsp”, “is”, “nbsp”, “plug”, “amp”, “play”}.

Each word in the array is then added to an in-memory list along with the current document id and a tuple representing the conjunction of all conditions currently on the stack.

The tuple is constructed according to Section 4.2, using the variable ranges specified in the `StartDocument` message. If positions are to be stored, an in-memory tree structure, representing the normalized document, is used to determine the hierarchical position of the word and its successor level as described in Section 4.3, and the position is also added to the list.

Once the `save` method is called by the user, all tuples in memory are sorted by the appropriate sort interval, and if any tuple represents the condition `true` and positions are not stored, all other tuples for that word and document are discarded. Then, all document names, ids, words, tuples and optionally positions in memory are saved as `documents.bin`, `words.bin` and `tuples.bin` in the specified location and format. Depending on the options, these files are either stored uncompressed, or they are compressed by using 7-bit encoding for integer and long values.

Chapter 6

Query Processing Implementation

6.1 Overview

In addition to an indexer, we created a query processor capable of keyword and phrase search against indexes in the format described in Section 4.4. Like the indexer, it was written in Visual Basic.NET and uses the Microsoft .NET Framework 2.0. Its main classes and workflow are diagrammed in Figure 34.

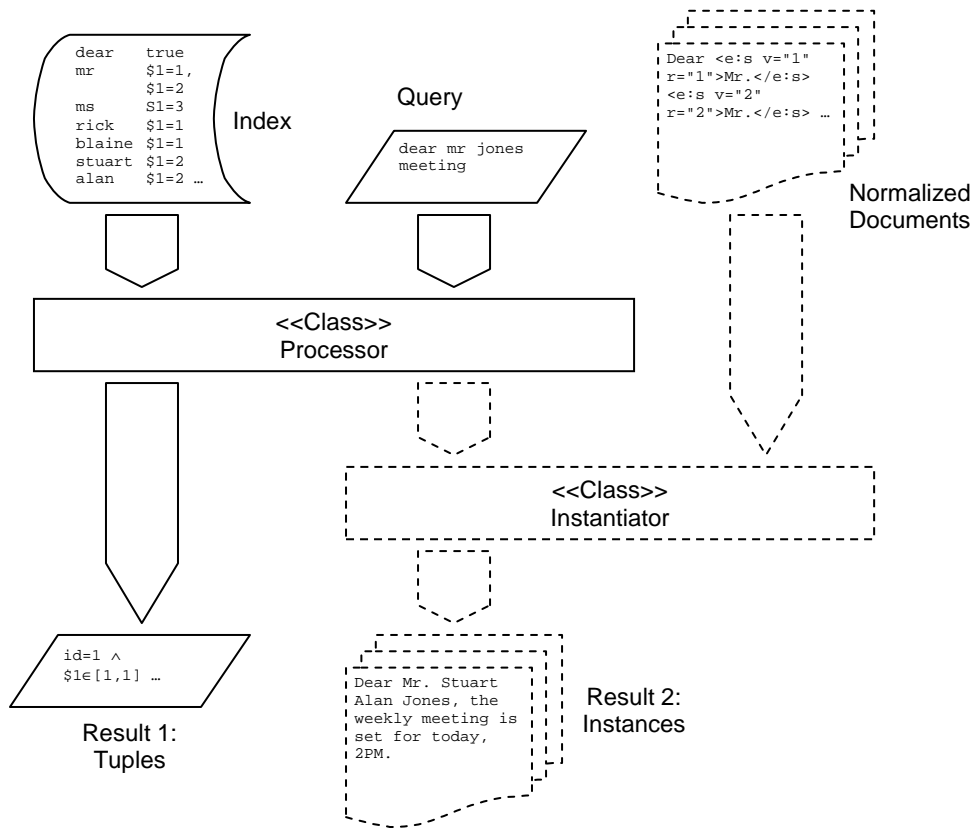


Figure 34: Query Classes and Workflow

The actual query processing step of the workflow from Section 1.4 is performed by the Processor class, which evaluates a query against an index and returns a list of resulting tuples. These tuples describe, in the format shown in Section 4.2, exactly which document instances match a given query. The number of returned tuples can be restricted, both per document and in total, in order to improve evaluation performance.

If presenting these tuples to the user as the result of a query would prove to be insufficient, they could also be used as inputs for another class, for example to materialize the instances themselves and present them to the user instead. While this functionality is not present in our implementation, the corresponding `Instantiator` class is still shown in the diagram for illustrative purposes.

Our query processor also does not implement some traditional information retrieval techniques such as stemming or scoring. However, since our solution is based on the concept of an inverted index, there is nothing to prevent them from being added later.

6.2 Processor Class

The `Processor` class implements the evaluation of queries against an index. As mentioned, it supports restricting the number of returned tuples, both per document and in total. Its members are shown in Figure 35.

```
Public Sub New(path, options, keepInMainMemory)
Public Function Query(queryExpression, totalMaximum,
                    documentMaximum) As List(Of Tuple)
```

Figure 35: Processor Members

When the `Processor` class is instantiated, it reads the content of the `documents.bin` and `words.bin` files, described in Section 4.4, into memory. Whether the content of `tuples.bin` – usually the largest file by far – should be kept in memory or only read from disk on demand can be specified, along with the location of the index, whether it includes positions and stop words, and whether it is compressed.

The `Processor` evaluates a query by using it to construct a left-deep tree (LDT) of iterators which read and merge tuples on demand [Gra96]. Phrase search is enabled by restricting any merging to tuples in sequential positions. Because the tuples returned by the iterator at the top of the LDT are the result of merging one tuple for each word in the query, they represent the results of the query itself.

```
Function MoveAhead(oldDocumentId) As Boolean
Function MoveNext() As Boolean
Property Current As Tuple
```

Figure 36: `ITupleEnumerator` Members

The `ITupleEnumerator` interface implemented by these iterators is shown in Figure 36. A class implementing this interface tries to find a new tuple every time the `MoveNext` function is called, and if one is found, makes it available through the `Current` property. While the way `MoveNext` is implemented can vary from class to class, any found tuples *must* be returned sorted by document id and sort interval as described in Section 4.2. This is necessary in order to be able to use a sweep line algorithm to merge the found tuples. The class also has to implement the `MoveAhead` function and, when it is called, try to find a tuple with a greater document id than the one specified. This allows the `Processor` to get a certain number of result tuples per document and then move ahead to the next document.

The three classes that implement the `ITupleEnumerator` interface are shown in Figure 37. The `TupleReader` class is used to read tuples for a specified word from an index, the `TupleMerger` class merges tuples returned by two other `ITupleEnumerator` classes, and the `TupleSkipper` class is a modification of the `TupleMerger` and used to implement the behav-

ior of skip words as described in Section 4.3. Sections 6.3 to 6.5 explain in more detail how each of these classes works.

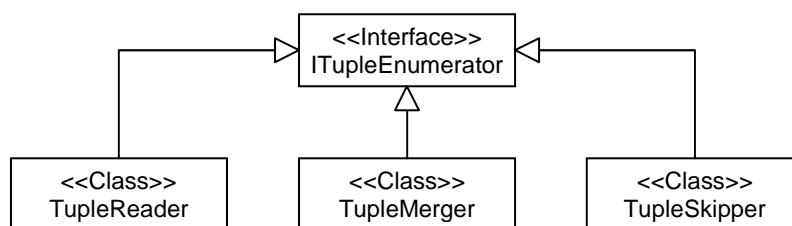


Figure 37: ITupleEnumerator Classes

The query format employed by the `Processor` uses a dash to connect words that are part of a phrase and a space to delimit phrases and single words. The query “*mr-jones meeting*”, for example, specifies a search for the phrase “*mr jones*” and the keyword *meeting*. Knowing this, a LDT is created from the query as follows:

First, a `TupleReader` is created for each word in the query. If the word is not part of a phrase, the `TupleReader` is instructed to ignore any positions that are read from the index.

Next, a `TupleMerger` is created for each phrase by combining the `TupleReaders` for the first two words into a new `TupleMerger` and recursively combining the result with the `TupleReader` for the next word into a new `TupleMerger`. If the used index does not contain any tuples for skip words, the combining is done by simply using the two `TupleReader` or `TupleMerger` iterators as inputs for a new `TupleMerger`. If such tuples do exist, however, the first iterator is used as the input for a new `TupleSkipper` instead, which is then used together with the other iterator as the input for the new `TupleMerger`. Inserting a `TupleSkipper` in this way allows tuples with non-sequential positions to be merged as long as there are skip words at intermediate positions.

Finally, these `TupleMergers` are combined with any remaining `TupleReaders` for single keywords by, again, recursively combining them into new `TupleMergers`, which are instructed to ignore positions when merging tuples.

```

Function Parse(queryExpression) As ITupleEnumerator
    root = Nothing
    phrases = queryExpression.Split(" ")
    For Each phrase In phrases
        sub = Nothing
        words = phrase.Split("-")
        If words.Length = 1 Then
            ' Single word, sub is a reader.
            sub = CreateTupleReader(words(0), False)
        ElseIf words.Length > 1 Then
            ' Phrase, sub is another left-deep tree of readers.
            For Each word In words
                reader = CreateTupleReader(word, True)
                sub = CreateTupleMerger(sub, reader, True)
            Next word
        End If
        root = CreateTupleMerger(root, sub, False)
    Next phrase
    Return root
End Function
  
```

Figure 38: Parse Function

Figure 38 shows the code for the described operation. The sub iterator represents the `TupleMerger` for a phrase or `TupleReader` for a keyword, and `root` represents the `TupleMerger` (or `TupleReader`, in case of a query with only a single word) for the entire query. The Boolean argument for `CreateTupleReader` and `CreateTupleMerger` specifies whether to use or discard positions when reading or merging tuples, and whether `CreateTupleMerger` should insert a `TupleSkipper` as described above. The resulting LDT for the example query is shown in Figure 39.

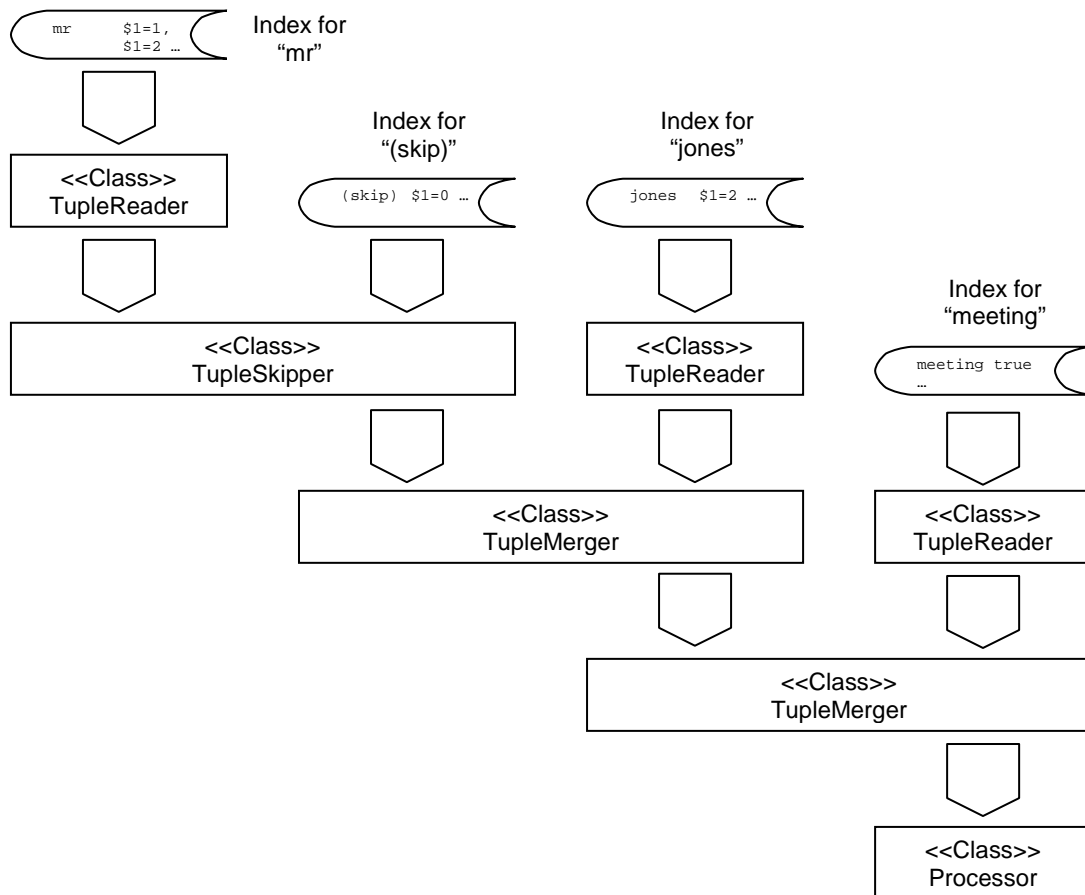


Figure 39: Left-Deep Tree Example

We know that the described method for creating left-deep trees is not optimal and could be improved by, for example, sorting the `TupleReaders` by increasing word count before merging them. Such optimizations of the query plan are a possible avenue for future work and are mentioned in Section 9.2. It should also be noted that if the options for the `Processor` specify that the index contains no stop words, any stop words are removed from the query while creating the LDT, therefore treating a query such as *“meeting is set for today 2pm”* like *“meeting set today 2pm”*.

After the LDT is generated, the `Processor` calls `MoveNext` on the root `ITupleEnumerator` to generate result tuples, adds them to a list, and optionally calls `MoveAhead` to move to the next document. Once the requested number of total results is reached or no more tuples can be found, the `Processor` returns the list as the result of the query.

6.3 TupleReader Class

The `TupleReader` class implements the `ITupleEnumerator` interface by reading tuples for a certain word from an index. Its constructor is shown in Figure 40.

```
Sub New(stream, options, usePositions, sortVariables)
```

Figure 40: `TupleReader` Constructor

If the `Processor` which instantiates the `TupleReader` has read the content of `tuples.bin` into memory, it supplies a `MemoryStream` on that data for the `TupleReader` to read from. Otherwise, it supplies a `FileStream` on the `tuples.bin` file itself. In either case, it first seeks to the correct offset in the stream according to the information read from `words.bin`. The `Processor` also specifies the index format, and whether to use or discard positions. Because the `TupleReader` needs to know the sort variables for all documents to construct the tuples, these are supplied as well.

Every time the `MoveNext` function is called, the `TupleReader` reads one tuple from the stream and makes it available through the `Current` property. Because the tuples in the index are sorted by document id and sort interval, any tuples returned by a `TupleReader` will likewise be sorted. If the index contains positions and the `Processor` specified their use, these are also read from the stream and added to the new tuple. Otherwise, they are read but discarded immediately. This is the case if the `TupleReader` is reading tuples for a keyword – as opposed to a word inside a phrase – because positions are irrelevant for keyword search, but must still be read to advance the offset in the stream.

If `MoveAhead` is called, the `TupleReader` will read and immediately discard every tuple until it reads a greater document id than the one specified in the call. If the `TupleReader` reads the sentinel id 0 during a call to `MoveNext` or `MoveAhead`, indicating the end of the index for the relevant word, it returns `False`, leaving the `Current` property undefined.

6.4 TupleMerger Class

The `TupleMerger` class implements the `ITupleEnumerator` interface and is used to merge tuples from the two other `ITupleEnumerators` specified as its inputs. Its constructor is shown in Figure 39.

```
Sub New(leftInput, rightInput, usePositions)
```

Figure 41: `TupleMerger` Constructor

Modern search engines use a *sort-merge algorithm* to merge lists of document ids for different keywords, taking advantage of the fact that these lists were previously sorted [BCC94]. Similarly, the `TupleMerger` class can use an adapted *sweep line algorithm* [DST+02] to merge tuples from its two input iterators as long as the tuples returned by each iterator are sorted by document id and ascending sort interval start.

The adapted algorithm, whose code is shown in Figure 42, works the following way: `Sweep` compares the two tuples at the beginning of each of its two input queues and dequeues the one with the smaller document id or sort interval start (as mentioned in Section 4.2, tuples with no sort interval are considered to have a sort interval from zero to plus infinity). It adds this `newTuple` to the *sweep area* for the queue it came from and then queries it against the sweep area for the other queue.

```

Function Sweep() As Queue(Of Tuple)
    ' While both queues have tuples left, compare them,
    ' query the lower tuple against the opposite sweep
    ' area, and add it to its own sweep area.
    results = New Queue(Of Tuple)
    Do Until leftQueue.Empty OrElse rightQueue.Empty
        If leftQueue.Peek() < rightInput.Peek() Then
            newTuple = leftQueue.Dequeue()
            leftSweepArea.Add(newTuple)
            results.Enqueue(Query(rightSweepArea, newTuple, True))
        Else
            newTuple = rightQueue.Dequeue()
            rightSweepArea.Add(newTuple)
            results.Enqueue(Query(leftSweepArea, newTuple, False))
        End If
        If Not results.Empty
            Return results
        End If
    Loop
    ...
    Return Nothing
End Function

```

(a) Sweep Function

```

Function Query(sweepArea, newTuple, swap) As Queue(Of Tuple)
    results = New Queue(Of Tuple)
    For Each tuple In sweepArea
        ' Check for each tuple if the document id or
        ' sort interval is lower than the new tuple's.
        If Sequential(tuple, newTuple) Then
            ' Remove tuples from the sweep area
            ' that are behind the new tuple.
            sweepArea.Remove(tuple)
        Else
            ' All other tuples intersect on the sort interval
            ' or they would not be in the sweep area yet.
            mergeTuple = Merge(tuple, newTuple, usePositions, swap)
            If mergeTuple IsNot Nothing Then
                results.Enqueue(mergeTuple)
            End If
        End If
    Next tuple
    Return results
End Function

```

(b) Query Function

Figure 42: TupleMerger Functions

Query first removes all tuples from the sweep area whose document id or sort interval end is lower than the `newTuple`'s document id or sort interval start, respectively. These tuples will not intersect any future `newTuple` that is coming from a queue sorted by document id and ascending sort interval start and can safely be discarded. Any tuples remaining in the sweep area are at least guaranteed to match the `newTuple`'s document id and intersect its sort interval (because their document id and sort interval start cannot be greater than `newTuple`'s or they would not have been dequeued first). Query tries to Merge each of these tuples with the `newTuple`, using the method described in Section 4.2. If, for phrase search,

usePositions was set to True at instantiation, it also tries to merge their position lists by using the MergePositions function shown in Figure 43, which uses a modified sort-merge algorithm. The swap parameter specifies whether to switch the two tuples' positions when calling MergePositions. If the Merge is successful, the result is added to the list of results. Sweep then returns this list if it is not empty, or it tries again with the next newTuple. The code that runs once one of the two input queues is empty is not shown for brevity, but works along the same lines.

```

Function MergePositions(leftPositions, rightPositions)
    As List(Of Position)
    l = 0
    r = 0
    positions = New List(Of Position)
    Do While l < countLeft AndAlso r < countRight
        If leftPositions(l) < leftPositions(r) Then
            ' Left position is smaller than right,
            ' so see if right position is successor.
            If Successor(leftPositions(l), rightPositions(r)) Then
                positions.Add(rightPositions(r))
            End If
            l += 1
        Else
            r += 1
        End If
    Loop
    Return positions
End Function

```

Figure 43: MergePositions Function

TupleMerger can now implement the MoveNext function by making its two input iterators available to Sweep as queues, calling Sweep to get a list of result tuples, making them available one at a time through the Current property, and calling Sweep again when reaching the end of the list. Because the tuples from both ITupleEnumerator inputs are sorted by document id and sort interval and are queried against the sweep area in that order, and because the results of such a query all have identical document ids and sort interval start values, all tuples returned by a TupleMerger will likewise be sorted, as is necessary to implement MoveNext. MoveNext returns False once calls to MoveNext for both its inputs return False, indicating that there are no more tuples to be merged.

MoveAhead is implemented by clearing the list of results, clearing both sweep areas, and calling MoveNext on both input iterators. It both of these calls return False, MoveNext returns False as well.

6.5 TupleSkipper Class

As mentioned at the end of Section 4.3, for phrase search on an index in our format to work correctly, a phrase must be treated as if any space between two words contained an arbitrary number of skip words, including zero. In other words, when searching for the phrase “for today 2pm”, it is also necessary to search for “for (skip) today 2pm”, “for (skip) today (skip) 2pm”, “for (skip) today (skip) (skip) 2pm”, and so on. This is similar to merging the tuples for any phrase fragment such as “for today” with tuples for skip words, except that instead of merging the tuples exactly once, a tuple for a phrase fragment can be merged with tuples for skip word zero, one, or many times, resulting in tuples representing “for to-

day”, “*for today (skip)*”, “*for today (skip) (skip)*” and so on. This functionality is implemented by the `TupleSkipper` class, whose constructor is shown in Figure 44.

```
Sub New(wordInput, skipInput)
```

Figure 44: `TupleSkipper` Constructor

The `TupleSkipper` class is a variation of the `TupleMerger` class, with a few differences: The call to `Query` in the `Sweep` function has been replaced by a call to a new `RecursiveQuery` function, each tuple for a word or phrase fragment from its `wordInput` is added once to the results without being merged, and the `Merge` function is always called with `usePositions` being `True`. Apart from these changes, `TupleSkipper` implements the `ITupleEnumerator` interface exactly as the `TupleMerger` class does. The code for the changed `Sweep` function and the new `RecursiveQuery` function is shown in Figure 45.

```
Function Sweep() As Queue(Of Tuple)
    results = New Queue(Of Tuple)
    Do Until wordQueue.Empty OrElse skipQueue.Empty
        If wordQueue.Peek() < skipInput.Peek() Then
            newTuple = wordQueue.Dequeue()
            wordSweepArea.Add(newTuple)
            ' Add word to results unmerged.
            results.Enqueue(newTuple)
            results.Enqueue(Query(skipSweepArea, newTuple, True))
        Else
            newTuple = skipQueue.Dequeue()
            skipSweepArea.Add(newTuple)
            results.Enqueue(Query(wordSweepArea, newTuple, False))
        End If
        If Not results.Empty
            Return results
        End If
    Loop
    ...
    Return Nothing
End Function
```

(a) `Sweep` Function

```
Function RecursiveQuery(sweepArea, newTuple, swap) As Queue(Of Tuple)
    results = New Queue(Of Tuple)
    recursiveResults = Query(sweepArea, newTuple, swap)
    For Each mergeTuple As Tuple In recursiveResults
        ' Add new tuple to results.
        results.Enqueue(mergeTuple)
        ' Add new tuple to the word sweep area.
        wordSweepArea.Add(mergeTuple)
        ' Query again against the skip sweep area.
        results.Enqueue(RecursiveQuery(rightSweepArea, mergeTuple, True))
    Next mergeTuple
    Return results
End Function
```

(b) `RecursiveQuery` Function

Figure 45: `TupleSkipper` Functions

If a `TupleSkipper` is now created, for example, for a `TupleReader` for the word *today* and then used together with the `TupleReader` for *2pm* to create a new `TupleMerger`, it will pass through any tuples it gets from the `TupleReader` for *today*. However, it will also merge these tuples repeatedly with tuples from another `TupleReader` for *(skip)*, add all merged tuples to its list of results – they are results for “*today (skip)*”, “*today (skip) (skip)*” and so on –, and add them to the sweep area for *today* so that they may be merged with other skip word tuples later on.

Chapter 7

Experiments

7.1 Overview

Using the indexer and query processor implementations described in Chapter 5 and Chapter 6, we conducted a number of experiments on documents we had converted into XML format. We used three different sets of documents – Latex files, Twiki web pages and Emails –, and the results are listed in Sections 7.2 to 7.4.

We created two indexes, based on a list of suitable rules, for each set: One in the default, compressed format described in Section 4.4, and one with no positions or common stop words – mentioned in Section 5.4 – in order to examine the tradeoff of support for phrase search versus index size and indexing speed. Additionally, we created two indexes using the same formats, but no rules at all. They represent the format and performance of a traditional inverted index and were used as a baseline. Most of our performance measurements, such as indexing time, index size and query time, will be given in absolute numbers as well as relative to this baseline.

Each indexing test consisted of creating the index using the `IndexWriter` from Section 5.4 and creating the associated normalized documents using the `InstanceWriter` from Section 5.3. The normalized documents were not actually needed to create the index, but we consider this the most likely usage scenario because it allows for the materializing of instances as described in Section 6.1. For each test, we measured the indexing time, index size and size of the normalized templates. Times were calculated as the average of two runs after one warm-up run.

In addition, we ran a number of keyword and phrase searches, designed to either compare the performance of the rules-based index against the baseline or show the advantages of the rules-based index in terms of result quality. The phrase queries, of course, were not run against those indexes without positions. We measured execution times twice: Once for returning all results, which allows determining the exact document instances matching the query, and once for returning only one result per document, which is more similar to the results returned when using traditional inverted indexes. Due to the very short execution time of most queries, times were calculated as the average of 5,000 runs after one warm-up. Each set of query tests was run twice, once using a *hot* cache where the entire index was loaded into memory before evaluation, and once using a *cold* cache where the content of `tuples.bin` was loaded from disk (or disk cache) on demand, as described in Section 6.2.

All experiments were run on IBM ThinkPads with Intel Pentium M 1.7GHz processors, 1GB of RAM and Windows XP SP2 as the operating system. The code was written in Visual Basic .NET, compiled with optimizations, and run as an executable on the Microsoft .NET Framework 2.0.

7.2 Latex

This set of documents was created by converting 1047 files – papers, manuscripts and other documents – from the binary LaTeX format to XML by using the Tralics translator [Gri03]. The size of the converted files totaled 19.01 MB, and an example of their format is given in Figure 46.

```
Table <ref target="uid49" /> lists the hit ratio (on tuples of
<hi rend="it">MARA</hi>) and the costs of the 1.2 million small
<hi rend="it">MARA</hi> queries
<note id="uid50" place="foot">...</note>
for the three different configurations.
...

```

Figure 46: Latex XML Example

Latex documents, especially papers and manuscripts, tend to contain a large number of inlined figures, pictures, tables, and notes such as the `note` element in the example above. We created an Annotation rule, shown in Figure 47, that defines such elements to be annotations in order to be able to search for phrases across these elements. One such search might be for the phrase “MARA queries for the three different configurations” in the example above.

```
<annotation match="//figure|//picture|//table|//note[@place='foot']" />
```

Figure 47: Latex Rule

The results of our indexing and query processing experiments are shown below.

Indexing

	Index Size in MB			Indexing Time in s		
	Baseline	Rules	Difference	Baseline	Rules	Difference
Positions	5.96	6.23	+5%	38.61	40.27	+4%
No positions	1.72	1.76	+2%	35.26	36.91	+5%

Figure 48: Index Size and Indexing Time

Figure 48 shows the index size and indexing time for the four indexes we created. Even though our rules create two instances for every single document, the numbers for our rules-based index are virtually the same as for the baseline index. Given the fact that information common to both instances is stored in the index only once, this is to be expected. The increase in size of the rules-based index is mainly due to the addition of skip word tuples at annotation positions as explained in Section 4.3. The large drop in index size when omitting positions and stop words is also expected, given that the list of positions for each tuple does not have to be stored. Overall, the index sizes are very reasonable when compared to the total size of the indexed documents.

	Size of Documents in MB		
	Baseline	Rules	Difference
	19.01	19.31	+2%

Figure 49: Size of Original vs. Normalized Files

The size of the normalized documents, shown in Figure 49, is virtually unchanged compared to the original documents, giving proof of the compactness of the format that was described in Section 3.3.

Query Processing

	Query	# of Matching Documents		# of Matching Instances
		Baseline	Rules	Rules
Q1	example paper	268	268	535
Q2	sigmod presentation	27	27	52
Q3	minimum selectivity	30	30	60
Q4	expression constructed over the alphabet	9	9	18
Q5	minimum-selectivity	8	8	16
Q6	expression-constructed-over-the-alphabet	0	1	1

Figure 50: Queries and Number of Results

Figure 50 shows the queries that were run against the indexes. As mentioned in Section 6.2, a dash is used to connect words that should be searched for as a phrase, so Q5 and Q6 will not be run against indexes that do not contain positions. In addition, Q4 will be run against those indexes as “*expression constructed over alphabet*”. Q1 through Q4 are queries with increasing selectivity and decreasing number of results. Q2 and Q3 return only one instance for some matching documents because at least part of the query was only found inside an annotation. Q5 shows the better selectivity of the phrase query over the corresponding keyword query Q3, but no change in the number of results for the rules-based index when compared to the baseline.

```
There is an expression
<note id="uid50" place="foot"> quickly</note>
constructed over the alphabet.
```

Figure 51: Possible Document for Q6

Q6, however, only returns a result for the rules-based index, because the relevant document looks similar to Figure 51 and only the exclusion of the `note` element in the empty instance makes a match possible. This is one case where the application of rules leads to better quality of results because more documents that are relevant can be found.

	Hot Cache in ms			Cold Cache in ms		
	Baseline	Rules	Difference	Baseline	Rules	Difference
Q1	0.85	0.96	+13%	1.65	1.79	+8%
Q2	0.14	0.14	±0%	0.90	0.89	-1%
Q3	0.14	0.15	+7%	0.92	0.90	-2%
Q4	7.48	7.67	+3%	10.70	10.84	+1%
Q5	0.20	1.47	+635%	0.98	2.76	+182%
Q6	0.28	3.35	+1096%	2.24	7.53	+236%

Figure 52: Running Times for All Results, with Positions

Figure 52 shows, for the keyword queries Q1 through Q3, the relation of the runtime of a query to the number of returned results. We will see why the runtime for Q4 is so high once

we look at the numbers for the indexes without position. The runtimes for the keyword queries are about the same for the runes-based and baseline indexes. This is understandable, considering most words appear outside any annotations and are therefore stored with the same tuple in both indexes.

The runtimes for Q5 are higher than for Q3 because, in addition to lists of document ids and tuples, it is also necessary to merge the position lists for the tuples. For Q6, however, they are lower than for Q4 because the phrase “*expression constructed*” has much higher selectivity than the keywords *expression* and *constructed*. The reason why the runtimes for the phrase queries are several times higher for the rules-based index when compared to the baseline is that, in order to find phrases across annotations, each tuple for a phrase fragment is attempted to be merged with any skip word tuples. This process is described in Section 6.5, and it pays off when Q6 returns a result only when using the rules-based index.

Using a cold cache introduces a delay proportional to the number of tuples that have to be loaded from disk. Except for that delay, the numbers for the cold and hot cache tests are the same. It is up to the user to decide whether this delay is an acceptable tradeoff for the reduced memory requirements.

	Hot Cache in ms			Cold Cache in ms		
	Baseline	Rules	Difference	Baseline	Rules	Difference
Q1	0.86	0.91	+6%	1.65	1.70	+3%
Q2	0.14	0.14	±0%	0.88	0.86	-2%
Q3	0.14	0.15	+7%	0.94	0.90	-4%
Q4	7.43	7.60	+2%	10.68	10.81	+1%
Q5	0.20	1.48	+640%	0.96	2.73	+184%
Q6	0.28	3.32	+1086%	2.26	7.46	+230%

Figure 53: Running Times for one Result per Document, with Positions

As we can see in Figure 53, returning only one result per document and then moving ahead, as explained in Section 6.2, does not perceptibly change query running times. This is not surprising – since we have at most one rule, one variable, and two instances per document, the index contains only one tuple for most words, and the `tupleReader`, described in Section 6.3, cannot move ahead at all.

	Hot Cache in ms			Cold Cache in ms		
	Baseline	Rules	Difference	Baseline	Rules	Difference
Q1	0.46	0.45	-2%	1.22	1.19	-2%
Q2	0.08	0.08	±0%	0.84	0.82	-2%
Q3	0.09	0.09	±0%	0.85	0.85	±0%
Q4	0.21	0.19	-10%	1.72	1.78	+3%

Figure 54: Running Times for All Results, without Positions

Figure 54 shows why removing stop words from the index can be a good idea. The running time for Q4, now interpreted as “*expression constructed over alphabet*”, drops by about a factor of forty for the hot cache because the massive list of tuples for the word *the*, which is no longer in the index, does not have to be loaded and merged. For the other queries, the hot cache times drop by about half because the position lists for each tuple do not have to be read and then discarded. Whether this speedup – which is proportionally much smaller for the cold cache – is worth not being able to run phrase queries is again up to the user.

	Hot Cache in ms			Cold Cache in ms		
	Baseline	Rules	Difference	Baseline	Rules	Difference
Q1	0.47	0.48	+2%	1.24	1.24	±0%
Q2	0.08	0.08	±0%	0.83	0.83	±0%
Q3	0.09	0.09	±0%	0.83	0.84	+1%
Q4	0.20	0.20	±0%	1.70	1.74	+2%

Figure 55: Running Times for one Result per Document, without Positions

Figure 55 shows that, as with the indexes with positions and for the same reasons, returning only one result does not have any perceptible influence on the query times.

Generally, the running times for the rules-based index are very competitive when compared to the baseline, even for queries with low selectivity and phrase queries. The only hit is taken when skip word tuples have to be loaded and merged for phrase search.

7.3 Twiki

The Twiki set of documents was created by converting a suitable TWiki [Twi06] website to XML, using our own converter. The resulting 410 documents have a total size of 4.23 MB.

Twiki uses a text-based file format that allows tracking revisions of documents. In each document, the latest version is shown on top, and each previous version includes information about how to create it by changing the newer version. As an example, the versioned document from Section 1.3 in Twiki format might look as shown in Figure 56, with the line @d2 1 indicating that the second line has to be deleted from the newer version in order to get the older version.

```

...
1.2
text
@There is going to be a meeting next Friday.
Time is not yet determined.
@

1.1
text
@d2 1
@

```

Figure 56: Twiki Example

If we convert the example using our XML converter, it looks as shown in Figure 57, with a `d` element indicating what content to delete to get version number 0.

```

<doc>
  <metadata>...</metadata>
  There is going to be a meeting next Friday.
  <d version="0">Time is not yet determined.</d>
</doc>

```

Figure 57: Twiki XML Example

Because the `a` and `d` elements that signify additions and deletions refer to later, not earlier, versions, the Version rules required to create the suitable instances have inverted operators compared to the versioning example in Section 2.5.

```
<excluded match="/doc/metadata" />
<excluded match="/doc//meta" />
<join type="number">
  <version match="/doc//a" key="$m/@version"
    type="number" compare="lessorequal" />
  <version match="doc//d" key="$m/@version"
    type="number" compare="greater" />
</join>
```

Figure 58: Twiki Rules

Figure 58 shows the defined rules, including two Excluded rules to remove metadata that is not relevant for searching.

Indexing

	Index Size in MB			Indexing Time in s		
	Baseline	Rules	Difference	Baseline	Rules	Difference
Positions	1.91	3.52	+84%	9.57	10.89	+14%
No positions	0.56	1.07	+91%	9.00	9.62	+7%

Figure 59: Index Size and Indexing Time

The larger number of instances per document when compared to the Latex documents from Section 7.2 shows in Figure 59. The rules-based indexes are about twice the size of the baseline indexes because each word has to be stored with tuples that describe in which versions of each document the word appears. Again, the indexes with positions are significantly larger than those without, due to the addition of position lists for each tuple.

As with the Latex documents, the indexing times are very competitive, with about two seconds required to create the index and normalized files for each megabyte of original documents.

Size of Documents in MB		
Baseline	Rules	Difference
4.74	4.53	-4%

Figure 60: Size of Original vs. Normalized Files

Figure 60 shows that the size of the normalized documents actually goes *down* when compared to the original files. The addition of switches to tag versioned content is more than offset by the exclusion of irrelevant metadata.

Query Processing

	Query	# of Matching Documents		# of Matching Instances	
		Baseline	Rules	Baseline	Rules
Q1	help author	40	8	64	64
Q2	virtual hands	5	5	39	39
Q3	example wiki	34	31	475	475
Q4	example wiki page	29	27	365	365
Q5	example wiki-page	2	1	4	4
Q6	site-is-goodstyle	0	1	10	10

Figure 61: Queries and Number of Results

The queries that were run against the indexes are shown in Figure 61. Q1 contains the word *author*, which is very frequent in Twiki metadata. Because such metadata is excluded from the rules-based index, it only returns the remaining, more relevant results. Q2 is a query with high selectivity, but identical results for both indexes. Q3 through Q5 are queries with increasing selectivity and show a lower number of results for the rules-based index, again because of the exclusion of metadata. Q6 shows how the rules-based index finds a result because it properly handles versioned content, allowing it to find an instance containing the phrase “*site is goodstyle*” even though these words do not appear in sequence in any original document.

	Hot Cache in ms			Cold Cache in ms		
	Baseline	Rules	Difference	Baseline	Rules	Difference
Q1	0.45	0.11	-76%	1.14	0.85	-25%
Q2	0.02	0.04	+100%	0.74	0.79	+7%
Q3	0.22	0.81	+268%	0.91	1.63	+79%
Q4	0.40	2.33	+483%	1.51	3.55	+135%
Q5	0.64	5.77	+802%	1.66	7.48	+351%
Q6	1.19	10.08	+747%	2.43	12.37	+409%

Figure 62: Running Times for All Results, with Positions

The lower running time for Q1 against the rules-based index directly correlates with the lower number of returned results because of the excluded metadata. The other queries, however, take much longer to run. This is because most words do not occur in all versions of document and are therefore stored using one or more tuples describing the matching versions instead of only once with the empty tuple, and these tuples all have to be merged. In addition, versioned content produces a large number of skip word tuples that have to be merged for phrase search, leading to the even larger increases in running time for Q5 and Q6. These longer running times, however, have to be weighed against the fact that the rules-based index is able to return a result for Q6, and the baseline index is not.

The cold cache numbers show the same increase in running times that we saw for the Latex experiments in Section 7.2.

	Hot Cache in ms			Cold Cache in ms		
	Baseline	Rules	Difference	Baseline	Rules	Difference
Q1	0.43	0.11	-74%	1.18	0.85	-28%
Q2	0.02	0.03	+50%	0.73	0.76	+4%
Q3	0.24	0.51	+113%	0.94	1.29	+37%
Q4	0.39	1.08	+177%	1.45	2.28	+57%
Q5	0.61	5.77	+846%	1.69	7.52	+345%
Q6	1.19	10.09	+748%	2.42	12.39	+412%

Figure 63: Running Times for one Result per Document, with Positions

Figure 63 shows how only returning one result per document pays off for Q3 and Q4, which match a large number of instances per document. Hot cache running times for these queries go down by about half, but stay the same for the other queries. If only the matching documents are needed, it is therefore a good idea to specify this in order to reduce the running time of certain queries.

	Hot Cache in ms			Cold Cache in ms		
	Baseline	Rules	Difference	Baseline	Rules	Difference
Q1	0.19	0.08	-58%	0.93	0.82	-12%
Q2	0.01	0.03	+200%	0.70	0.75	+7%
Q3	0.08	0.55	+588%	0.81	1.33	+64%
Q4	0.17	1.80	+959%	1.26	3.03	+140%

Figure 64: Running Times for All Results, without Positions

The numbers in Figure 64 show that, as with the Latex experiments in Section 7.2, using the indexes without positions lowers query running times, this time by twenty to fifty percent for the hot cache.

	Hot Cache in ms			Cold Cache in ms		
	Baseline	Rules	Difference	Baseline	Rules	Difference
Q1	0.19	0.07	-63%	0.92	0.80	-13%
Q2	0.01	0.02	+100%	0.68	0.72	+6%
Q3	0.08	0.24	+200%	0.84	1.01	+20%
Q4	0.17	0.60	+253%	1.29	1.76	+36%

Figure 65: Running Times for one Result per Document, without Positions

Matching the results for the indexes with positions, Figure 65 again shows a drop-off for Q3 and Q4 when only one result per document is returned.

The results of the experiments with the rules-based indexes are again competitive, and for Q1 they are even better than the baseline because of the exclusion of metadata. For the other queries, we get generally better results, but at the cost of higher running times, especially for phrase search. As we mentioned in Section 6.2, optimizing the query plans in order to avoid a large number of attempted merges with skip word tuples is one possible way to improve these results.

7.4 Email

This set consists of a single document with a size of 51.77MB and is intended to demonstrate the ability of our indexer and query processor to handle large documents. It was created by converting a store of 16,500 email messages in 10,200 conversation threads into XML. Using our own converter, the emails were grouped by conversation and stored in the format shown in Figure 66.

```
<map>
  <conversation>
    <message>
      ...
      <from>Rick Blaine</from>
      <to>Stuart Alan Jones</to>
      <subject>Conference on Thursday</subject>
      <body>Stuart, Don't forget the conference tomorrow!</body>
    </message>
    ...
  </conversation>
  ...
</map>
```

Figure 66: Email XML Example

Since the emails are grouped by conversation thread, we created an Alternative rule that defines an instance for each such thread. This rule, shown in Figure 67, enables searching for conversations that contain certain keywords or phrases.

```
<alternative match="/map/conversation" key="position()" type="number" />
```

Figure 67: Email Rule

Because the set consists of only one document, the results when using the baseline index are very informative – basically, they can be seen as yes/no answers on whether or not the entire set matches the specified query. For the same reason, returning only one result per document would not be very helpful. Instead, we return the first 20 results, similar to the behavior of typical web search engines.

Indexing

	Index Size in MB			Indexing Time in s		
	Baseline	Rules	Difference	Baseline	Rules	Difference
Positions	29.14	41.67	+43%	135.85	157.33	+16%
No positions	2.86	12.61	+341%	106.61	132.62	+24%

Figure 68: Index Size and Indexing Time

Figure 68 shows that, as expected, storing information about the threads in which each word appears increases the index size considerably. This is especially obvious for the indexes with no positions, where each word has to be stored with potentially thousands of tuples in the rules-based index, but with a single empty tuple in the baseline index. The smaller size of the baseline indexes is offset, of course, by the fact that they cannot be searched by conversation.

Size of Documents in MB		
Baseline	Rules	Difference
51.46	51.77	+1%

Figure 69: Size of Original vs. Normalized Files

The compactness of our normalized document format is evident again in Figure 69, showing only a one percent increase in size compared to the original document.

Query Processing

	Query	# of Matching Documents		# of Matching Instances
		Baseline	Rules	Rules
Q1	meeting kossmann	1	1	1042
Q2	important message	1	1	349
Q3	sigmod meeting	1	1	76
Q4	ecdI healthcare	1	1	1
Q5	reviewer-feedback	1	1	1
Q6	important-message	1	1	1

Figure 70: Queries and Number of Results

Figure 70 shows the queries that were run against the indexes. Q1 through Q4 are increasingly selective keyword queries, while Q5 and Q6 are phrase queries that both match only a single conversation. All indexes, of course, return only one document per query.

	Hot Cache in ms			Cold Cache in ms		
	Baseline	Rules	Difference	Baseline	Rules	Difference
Q1	4.90	12.48	+155%	6.44	14.56	+126%
Q2	0.98	3.21	+228%	2.00	4.67	+134%
Q3	0.72	1.69	+135%	1.66	3.04	+83%
Q4	0.02	0.03	+50%	0.84	0.88	+5%
Q5	0.88	1.43	+63%	2.00	2.77	+39%
Q6	2.57	4.31	+68%	4.06	5.79	+43%

Figure 71: Running Times for All Results, with Positions

Figure 71 shows modest increases in running time for the rules-based index when compared to the baseline, due to the greater number of tuples that have to be merged. Considering the size of the index, the running times are quite impressive, with a maximum time of 14.56ms against a 41.67MB index when using a cold cache.

For the keyword queries, higher selectivity once again correlates with lower running times. Also, the running times for the phrase query Q6 are higher than for the related keyword query Q2 because position lists have to be merged. This is very similar to what we saw during the Latex and Twiki experiments.

	Hot Cache in ms			Cold Cache in ms		
	Baseline	Rules	Difference	Baseline	Rules	Difference
Q1	4.91	7.78	+58%	6.45	9.58	+49%
Q2	0.98	1.87	+91%	2.04	3.14	+54%
Q3	0.71	1.43	+101%	1.67	2.67	+60%
Q4	0.01	0.02	+100%	0.83	0.90	+8%
Q5	0.89	1.44	+62%	2.01	2.68	+33%
Q6	2.54	4.30	+69%	3.95	5.72	+45%

Figure 72: Running Times for 20 Results per Document, with Positions

Figure 72 shows again that, for queries with a large number of results, running times can be lowered substantially by not returning – and therefore not generating – all of these results. Using the hot cache, the numbers for running Q1 and Q2 against the rules-based index go down by almost half.

	Hot Cache in ms			Cold Cache in ms		
	Baseline	Rules	Difference	Baseline	Rules	Difference
Q1	0.01	6.08	+60700%	0.87	7.49	+761%
Q2	0.01	1.83	+18200%	0.86	3.23	+276%
Q3	0.01	0.83	+8200%	0.78	1.94	+149%
Q4	0.02	0.03	+50%	0.80	0.93	+16%

Figure 73: Running Times for All Results, without Positions

As can be seen in Figure 73, the hot cache numbers for queries against the rules-based index go down by about half when compared to the index with positions, because the positions for each tuple do not have to be read from the index.

For the baseline index, not storing positions in the index results in query running times that are drastically lower. However, all these results state is that the words in the query appear somewhere inside a single huge document, so they are not very useful as a comparison.

	Hot Cache in ms			Cold Cache in ms		
	Baseline	Rules	Difference	Baseline	Rules	Difference
Q1	0.01	1.63	+16200%	0.85	2.60	+206%
Q2	0.01	0.87	+8600%	0.85	1.59	+87%
Q3	0.01	0.53	+5200%	0.77	1.59	+106%
Q4	0.01	0.02	+100%	0.77	0.91	+18%

Figure 74: Running Times for 20 Results per Document, without Positions

Similar to the results for the indexes with positions, hot cache running times on the rules-based index go down by about half for Q1 and Q2 when only 20 results are returned.

In conclusion, considering the improved quality of results, we think that the query running times against the rules-based indexes are very reasonable for all sets of documents.

Chapter 8

Related Work

8.1 Semantic Web

This work has been inspired by several other research projects. Adding metadata like rules to documents in order to support more powerful queries is one of the main ideas of the *Semantic Web* [BHL01, W3C01, MH04]. There are two important differences, however. First, the purpose of the Semantic Web is to describe relationships between resources or documents. In contrast, our work aims to annotate documents in order to specify the semantics of their content. Second, an important component of the Semantic Web vision is a powerful query language that allows reasoning [PS06], while our work was for the most part concerned with simple keyword and phrase search queries. In all, the Semantic Web is a much more ambitious project, and the purpose of our work is to contribute to a much more focused problem.

8.2 Colorful XML

Recently, *Colorful XML* was proposed in order to give an XML document several interpretations [JLS+04]. The key idea is to define multiple hierarchies in an XML document – in some sense, Colorful XML is another way to define documents with multiple instances. However, the target applications, the capabilities of the language, and the query processing techniques are very different from this work.

8.3 PIX

The project that is probably most closely related to this work is the *PIX* project at AT&T. In fact, the Excluded class of our rule language was adopted from that project. Nevertheless, most of the work on PIX is orthogonal to our work. PIX devised techniques for phrase search on XML documents, which can be applied in the indexing and query processing steps of the workflow shown in Section 1.4.

8.4 Others

There has been, of course, a great deal of other work on various aspects of XML query processing, XML keyword search, and XML information retrieval [GSB+03, TSW05]. The combination of structured and keyword queries has also been studied before [KKN+04]. Furthermore, the emerging *XQuery FullText* standard [ABB+05] is relevant. Again, all this work is orthogonal and can be applied in the indexing and query processing steps.

As mentioned in Chapter 2, rules can be seen as a way to define XML views, and *XSLT* and *XQuery* are natural alternatives to express such views. Clearly, *XSLT* and *XQuery* are

more powerful than the rules in our rule language. However, normalization and the special techniques described in Chapter 3 cannot immediately be applied to general XSLT or XQuery views.

Chapter 9

Conclusion and Future Work

9.1 Conclusion

This work was motivated by the failure of today's generation of search engines to understand the semantics of document content. Documents are created by different applications such as Word, Excel, or scientific databases, but search engines treat all documents in the same syntactic way. For example, search engines do not realize that a revised document might encode several versions and that each version should potentially be indexed and queried individually. As a result, current search engines might return documents that are not relevant for a query and, at the same time, might miss relevant documents.

The key goal of our work is to extend the capabilities of these search engines. Our two main contributions are the definition of a *rule language* and the design of a scheme to *normalize* documents created through the application of such rules.

In addition to content, documents can also contain semantics, and rules make it possible to describe these semantics. A search engine can interpret these rules in order to virtually instantiate, index, and query the original documents in the way they are seen or understood by the user or other applications. The rules we devised are specifically geared towards patterns that commonly occur in documents of word processing applications, in web sites, and in e-health and scientific data. They make it possible to handle properly irrelevant formatting instructions in documents (the Excluded rule), annotations such as comments (Annotation), variants (Alternative), references to text inside the same or other documents (Placeholder and Field) as well as versioning (Version and Versionref). Rules can also be correlated, and multiple rules can be applied at once to a document.

Our approach to normalizing documents allows for the compact representation of different document instances – variants created by the application of rules to original documents – in a format that can be used as a basis for indexing and query processing, as demonstrated by our implementations. The beauty of normalization is that it factors out the common content of different document instances so it has to be indexed only once. Only when necessary are individual parts of certain instances indexed, individually and in a fine-grained manner. Our experiments show that this approach is effective and competitive with traditional information retrieval. The space and time overhead seems affordable, considering the increased quality of query answers.

9.2 Future work

There are several avenues for future work: There are possible improvements to our implementations of the indexer and query processor, such as the ability to create incremental indexes or optimize query plans based on the word frequency. Another avenue is to apply more powerful query paradigms to normalized documents – that is, in addition to keyword search, say, full-fledged XQuery expressions could be applied. Furthermore, the application

of modern approaches to XML information retrieval like query relaxation and structure-based search could be explored. Another idea is to apply the proposed techniques to the hidden web – rules could be seen as a much improved version of `robots.txt` files that describe how to interpret the data found by a crawler. In order to be practical, it will also be important to create libraries of suitable rules for documents generated by popular applications like Microsoft Office, Sun’s OpenOffice, CVS, Wiki, et cetera.

References

- [ABB+05] S. Amer-Yahia, C. Botev, S. Buxon, P. Case, J. Doerre, D. McBeath, M. Rys, and J. Shanmugasundaram. *XQuery 1.0 and XPath 2.0 Full-Text*. <http://www.w3.org/TR/2005/WD-xquery-full-text-20050404/>.
- [AFS+03] S. Amer-Yahia, M.F. Fernández, D. Srivastava, and Y. Xu. *PIX: A System for Phrase Matching in XML Documents*. In *ICDE*, pages 768–776, 2003.
- [ALP04] S. Amer-Yahia, L. V. S. Lakshmanan, and S. Pandit. *FleXPath: Flexible Structure and Full-Text Querying for XML*. In *SIGMOD Conference*, pages 83–94, 2004.
- [BCC94] E. W. Brown, J. P. Callan, and W. B. Croft. *Fast Incremental Indexing for Full-Text Information Retrieval*. In *VLDB*, pages 192–202, 1994.
- [BHL01] T. Berners-Lee, J. Hendler and O. Lassila. *The Semantic Web*. In *Scientific American*, 284(5):34–43, 2001.
- [BR99] R.A. Baeza-Yates and B.A. Ribiero-Neto. *Modern Information Retrieval*. ACM Press/Addison-Wesley, 1999.
- [DDJ+05] J.-P. Dittrich, C. Duda, B. Jarisch, D. Kossmann, and M. Vaz Salles. *Keyword Search on Application Data*. Unpublished manuscript, 2005.
- [DST+02] J.-P. Dittrich, B. Seeger, D.S. Taylor, and P. Widmayer. *Progressive Merge Join: A Generic and Non-blocking Sort-based Join Algorithm*. In *VLDB*, pages 299–310, 2002.
- [FMM+05] M. Fernández, A. Malhotra, J. Marsh, M. Nagy, and N. Walsh. *XQuery 1.0 and XPath 2.0 Data Model (XDM)*. 2005. <http://www.w3.org/TR/2005/CR-xpath-datamodel-20051103/>.
- [Gra96] G. Graefe. *Iterators, Schedulers, and Distributed-memory Parallelism*. In *Software – Practice & Experience*, 26(4):427–452, 1996.
- [Gri03] J. Grimm. *Tralics, a LaTeX to XML Translator*. In *TUGboat*, 24(3):377–388, 2003.
- [GSB+03] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram. *XRANK: Ranked Keyword Search over XML Documents*. In *SIGMOD Conference*, pages 16–27, 2003.
- [JLS+04] H. V. Jagadish, L. V. S. Lakshmanan, M. Scannapieco, D. Srivastava, and N. Wiwatwattana. *Colorful XML: One Hierarchy Isn't Enough*. In *SIGMOD Conference*, pages 251–262, 2004.
- [KKN+04] R. Kaushik, R. Krishnamurthy, J. F. Naughton, and R. Ramakrishnan. *On the Integration of Structure Indexes and Inverted Lists*. In *ICDE*, page 829, 2004.
- [Meg04] D. Megginson. *Simple API for XML (SAX 2.0)*. 2004. <http://sax.sourceforge.net/>.
- [MH04] D. L. McGuinness and F. van Harmelen. *OWL Web Ontology Language Overview*. 2004. <http://www.w3.org/TR/2004/REC-owl-features-20040210/>.
- [Mic05] Microsoft. *.NET Framework Developer's Guide*. 2005. <http://msdn.microsoft.com/library/en-us/cpref/html/frlrfssystemiobinarywriterclasswrite7bitencodedinttopic.asp>, <http://msdn.microsoft.com/library/en-us/cpref/html/frlrfssystemiobinarywriterclasswrite7bitencodedinttopic.asp>.

- [PS06] Eric Prud'hommeaux and A. Seaborne. *SPARQL Query Language for RDF*. 2006.
<http://www.w3.org/TR/2006/WD-rdf-sparql-query-20060220/>.
- [TSW05] M. Theobald, R. Schenkel, and G. Weikum. *An Efficient and Versatile Query Engine for TopX Search*. In *VLDB*, pages 625–636, 2005.
- [Twi06] TWiki. *TWiki – An Enterprise Collaboration Platform*. 2006.
<http://twiki.org/>.
- [W3C01] World Wide Web Consortium. *Resource Description Framework*. 2001.
<http://www.w3.org/RDF/>.