The VLDB Journal (2010) 19:735–758
DOI 10.1007/s00778-010-0187-5

REGULAR PAPER

Predicate-based indexing for desktop search

Cristian Duda · Donald Kossmann · Chong Zhou

Received: 21 January 2009 / Revised: 22 January 2010 / Accepted: 14 March 2010 / Published online: 13 May 2010

Abstract Google and other products have revolutionized the way we search for information. There are, however, still a number of research challenges. One challenge that arises specifically in desktop search is to exploit the structure and semantics of documents, as defined by the application program that generated the data (e.g., Word, Excel, or Outlook). The current generation of search products does not *understand* these structures and therefore often returns wrong results. This paper shows how today's search technology can be extended in order to take the specific semantics of certain structures into account. The key idea is to extend inverted file index structures with predicates which encode the circumstances under which certain keywords of a document become visible to a user. This paper provides a framework that allows to express the semantics of structures in documents and algorithms to construct enhanced, predicate-based indexes. Furthermore, this paper shows how keyword and phrase queries can be processed efficiently on such enhanced indexes. It is shown that the proposed approach has superior retrieval performance with regard to both recall and precision and has tolerable space and query running time overheads.

Work performed at ETH Zurich, Switzerland, supported by China Scholarship Council.

C. Duda (✉) · D. Kossmann
ETH Zurich, Zurich, Switzerland
e-mail: cristian.duda@inf.ethz.ch
URL: http://www.systems.ethz.ch

D. Kossmann
e-mail: donald.kossmann@inf.ethz.ch
URL: http://www.systems.ethz.ch

C. Zhou
Huazhong University of Science and Technology, Wuhan, China
e-mail: chong.zhou@inf.ethz.ch
URL: http://www.systems.ethz.ch

Keywords Search · Application data · Desktop search · Information retrieval · Databases

1 Introduction

Current search engines, such as Google [28], Yahoo! [46], Live Search [36], solve well the problem of crawling and searching the Web. However, searching application data on the desktop is still an unsolved problem. Users access data through an application such as Word, Excel, Wiki (Web browser) or an E-mail client. The problem is that Google Desktop, Apple SpotLight, and related products do not see the application data with the eyes of the user. The reason is that applications encode properties with special semantics into the data using annotations. Typical examples are comments or footnotes in text documents, E-mail threads, and history information of versions in Wikis. The semantics of these properties and annotations are important for search. Comments, for instance, need to be interpreted correctly when users search for a specific phrase in a text document.

Figure 1 shows the problem of existing search engines: They index the raw data as stored in the file system. In contrast, users see the data using applications (e.g., Word, Outlook, or a Wiki system). As a result, traditional search engines return wrong and unexpected results in a number of scenarios.

1.1 Motivating examples

Example 1 Bulk Letters (Word and Excel) Traditional search engines fail to return correct results if the data is partitioned into several files. The *Mail merge* functionality of office applications such as Word and Excel used in order to create bulk letters demonstrates this deficiency of traditional

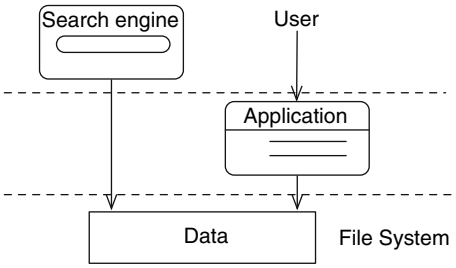
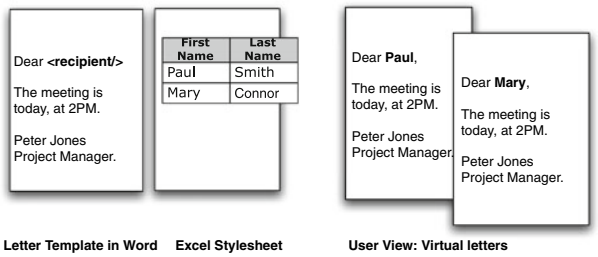


Fig. 1 Search engine view vs. User view



(a) Application Data **(b)** User View

Query	Trad.	Desired
Dear Paul	-	*letter.doc* [Instance 1]
Dear Mary	-	*letter.doc* [Instance 2]
Paul, Mary	*names.xls*	*letter.doc, names.xls*

(c) Query Results

Fig. 2 Example 1: bulk letters (word ⋈ excel)

search engines. Figure 2a shows a letter written in Word, containing placeholders for the recipient name. Furthermore, Fig. 2a shows an Excel spreadsheet containing the names of the recipients who are supposed to receive the letter. When the *Mail merge* function is applied using the Word application, the bulk letters are instantiated so that from a user's and application perspective the letters shown in Fig. 2b are generated. Correspondingly, the queries *Dear Paul* and *Dear Mary* should match the Word document, as illustrated in the third column of Fig. 2c. More specifically, the query *Dear Paul* matches the first letter (indicated as *Instance 1* in Fig. 2c) and *Dear Mary* matches the second letter. Unfortunately, traditional search engines do not find these results because the keywords *Dear* and *Paul* are not part of the same file. For the same reason, traditional search engines return no results for the *Dear Mary* query, as shown in the second column of Fig. 2c.

Figure 2c lists a third example query: *Paul, Mary*. Naturally, traditional search engines return the Excel file as the only result because the Excel file is the only file that contains both keywords. In many usage scenarios, however, it may be desirable to return the Word letter in addition (or instead of) the Excel spreadsheet. The Word letter provides valuable

```
<deleted id="1"> <info time="3/28/2009"/> </deleted>
<inserted id="2"> <info time="3/28/2009"/> </inserted>
<delete id="1">Mickey</delete>
<insert id="2">Donald</insert>
  likes
<delete id="1">Minnie</delete>
<insert id="2">Daisy</insert>.
```

(a) Application Data

Id	Text Subpart	Comment
Instance 1:	Mickey likes Minnie.	Time: $<$ 3/28/2009
Instance 2:	Donald likes Daisy.	Time: $\geq$ 3/28/2009

(b) User View

Query	Trad.	Desired
Mickey likes Daisy	*disney.doc*	-
Mickey likes Minnie	*disney.doc*	*disney.doc [Inst. 1]*
"Mickey likes Minnie"	-	*disney.doc [Inst. 1]*

(c) Query Results

Fig. 3 Example 2: versioned data (OpenOffice)

context by showing that both Paul and Mary were invited to the same meeting. This work will show how search can be parameterized so that a search engine returns either the Excel file, the Word file, or both files, depending on the user's intentions.

In summary, traditional search engines fail in this example because they do not realize that the Word and Excel files represent a set of letters. Rather than indexing the Word and Excel files, the letters should be indexed.

Example 2 Versioning (Multiple Worlds) Another scenario in which traditional search engines fail is when different *worlds* are encoded into a single document. A classic example are documents which encode their own history of insertions and deletions. Figure 3a shows an OpenOffice document. The initial version of the document was *Mickey likes Minnie*. On March 28, 2009, a user deleted *Mickey* and *Minnie* and replaced them by *Donald* and *Daisy*. The user actions were traced in the document using `deleted` and `inserted` annotations. Figure 3b shows the two versions of the document as they are seen by a user.

Again, traditional search engines return wrong results because they do not interpret the `deleted` and `inserted` annotations correctly. Figure 3c lists three queries. These three queries show how users who want to search in the versions of the document can get confused by traditional search engines. The first query is *Mickey likes Daisy*. All three keywords can be found in the document so that a traditional search engine naturally returns the document as a match for this query. However, at no point in time were all three keywords part of the document. In other words, no version of the document (Fig. 3b) contains all three keywords. So, the

desired result should be that the document does *not* match this query. The second query asks for *Mickey likes Minnie*. Indeed, the document matches this query. Nevertheless, the result of a traditional search engine is confusing: When the user opens the document using OpenOffice, the user will see the latest version, *Donald likes Daisy*, and wonder why the document was returned as a result. The correct answer is more specific and indicates that the first version of the document should be considered. The third query carries out a phrase search for *"Mickey likes Minnie"* (indicated with quotation marks). According to a traditional search engine, the document does *not* match because the three keywords are not stored adjacently in the document. However, the user would expect the document to match.

In summary, traditional search engines fail in this example for similar reasons as in Example 1: A traditional search engine should index the set of *instances* (or versions) encoded in the document rather than the raw data as stored in the file system. Traditional search engines only do well in this example if the user intends to do an *inter-version* search. In this scenario, indeed, interpreting the document as a single instance is the right interpretation. Nevertheless, even in such a scenario, phrase searches are not processed correctly.

1.2 Problem statement

The examples of the previous subsection showed that a document is best represented as a set of *instances*. Each instance (rather than the raw data stored in the file system) represents a specific variant of the document as seen by the user. Search engines should, therefore, index this set of instances rather than the raw data.

A naïve approach to index instances is to materialize all instances and index the materialized instances. Unfortunately, this approach is not viable because the number of instances can become very large. Concretely, the number of instances grows exponentially with the number of *rules* that are needed to describe the instances. (Rules are described in Sect. 3.) For example, Word documents can represent bulk letters (as in Example 1) and be versioned (as in Example 2): In such a scenario, the number of instances is the product of the number of bulk letters and the number of versions. The goal of this paper is to index these instances with linear space requirements and process queries on instances in linear time.

The examples demonstrate another important requirement: search modes. In some scenarios, users wish to control the set of instances considered for a search. For instance, a user might toggle between searching across versions or searching each version individually in Example 2. Furthermore, a user may want to control whether the *Paul, Mary* query returns the Word letter, the Excel spreadsheet, or both in Example 1. The goal is to have a unified framework and a *single* index which allows to specify and process all these queries in all modes.

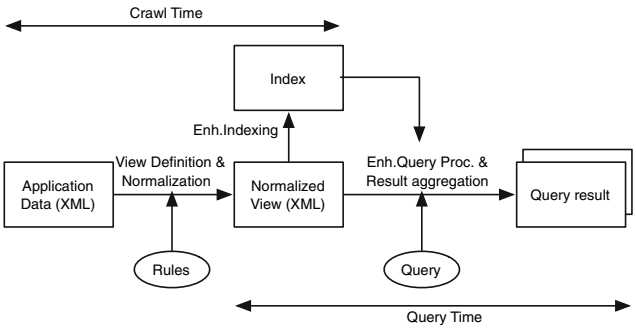


Fig. 4 Enhanced desktop search

The focus of this work is on a Boolean information retrieval model. That is, the semantics of queries are defined such that a document / instance either matches a query or it does not. The framework presented in this paper can be extended to various scoring schemes and we will briefly indicate how that can be done. Studying such scoring schemes in detail, however, is beyond the scope of this paper. Furthermore, this paper focuses on *desktop search*; that is, search through personal data such as E-Mail, Office documents (e.g., Word, Excel, Powerpoint, Latex), and Wikis. It is shown (and experimentally evaluated) how the framework can be used to index and search through object-oriented code with inheritance (e.g., Java and C++ code). How to adapt the proposed framework for more general Web applications has been studied in [22].

1.3 Plan of attack

Figure 4 shows the proposed architecture of an enhanced search engine. Just like any other search engine, it is composed of two sub-systems: (a) the crawler and (b) the query processor. The crawler scans through the data, thereby, creating the search index. The query processor uses the index in order to process keyword and/or phrase searches.

When compared to a traditional search engine, the architecture of Fig. 4 has the following differences:

- *View definition:* Application-specific rules describe how to create instances for a document. For example, there is a specific set of rules that can be applied to all Word documents.
- *Normalization:* We propose a special representation of the set of all instances of a document. We call this representation the *normalized view* of a document. The magic is that it encodes all the variable parts of a document and stores the parts that are common to all instances only once.

Table 1 Inverted file for versions (Fig. 8)

Keyword	DocId	Predicate
Mickey	*disney.doc*	$R_2 < 3/28/2009$
likes	*disney.doc*	$true$
Minnie	*disney.doc*	$R_2 < 3/28/2009$
Donald	*disney.doc*	$R_2 \geq 3/28/2009$
Daisy	*disney.doc*	$R_2 \geq 3/28/2009$

Table 2 Query language on application data

Operator	Name	Description
w_1 AND w_2	Conjunction	Instances containing both keywords.
w_1 OR w_2	Disjunction	Instances containing either keyword.
NOT w_1	Negation	Instances not containing the keyword.
w_1 NEXT w_2	Phrase Search	Instances where keywords are adjacent.

– *Predicate-based Index:* Furthermore, we propose to extend traditional inverted indexes (e.g., Lucene) by an additional column that contains a predicate. This predicate encodes in which instances of a document a certain keyword can be found. Furthermore, different ways of applying this predicate allow to toggle between different search modes.
– *Result aggregation:* As shown in the two examples, it is often important to identify the specific instance that matches the query. For instance, only the letter to Paul matches the query *Dear Paul* in Example 1. Again, the predicates in the index help to identify these instances if not all instances match the query.

Arguably the most striking difference to traditional search is the use of predicates in inverted indexes. To get a feeling for this mechanism, we would like to show how it works for Example 2. For this example, this mechanism is particularly simple and intuitive. Table 1 shows the extended inverted file for this example. The predicate encodes under which conditions a document involves a keyword. In other words, the predicate encodes the set of instances of the document that involve the keyword. Here, R_2 is a variable that represents the rule that created this predicate (details described later).

Using the index of Table 1, the query *Mickey likes Daisy* is processed as follows:

1. Each keyword is looked up individually. The results are the following three inverted lists: (The three dots indicate that the inverted index may contain references to other documents which contain these keywords.)

$\langle Mickey, disney.doc, R_2 < 3/28/2009\rangle,\ldots$
$\langle likes, disney.doc, true\rangle,\ldots$
$\langle Daisy, disney.doc, R_2 \geq 3/28/2009\rangle,\ldots$

2. The intersection of the three lists is computed, thereby computing a logical AND of the predicates. The result is:

$\langle disney.doc,$
$R_2 < 3/28/2009 \wedge true \wedge R_2 \geq 3/28/2009\rangle$

3. Postprocessing: The predicate is evaluated. In this case, the reduction of the predicate is *false*. Consequently, *disney.doc* is not a match to this query.

The other queries of Example 2 can be processed correspondingly. For phrase search, position information needs to be kept in the inverted index (detailed in Sect. 9). To rank the relevance of a document, the index needs to be extended with scoring information (outlined in Sect. 10); however, the bulk of this paper will assume a Boolean information retrieval model without scores. In order to toggle between search modes, certain predicates need to be suppressed; to carry out an *inter-version search*, for instance, all R_2 predicates which encode the versioning need to be suppressed. If the R_2 predicates are suppressed, *disney.doc* would be returned as a result for the *Mickey likes Daisy* query. Even though this predicate-based indexing mechanism looks simple, it is extremely powerful: It can handle all use cases that we have encountered for desktop search (E-Mail, Wikis, any kind of Office document). Furthermore, this mechanism is composable so that different patterns such as placeholders of Example 1 and versions of Example 2 can be combined. Finally, as will be shown, it can be implemented efficiently.

1.4 Contributions and overview

In summary, the main contributions of this work are as follows:

– *Rule language:* A declarative rule language that is able to specify the instances of any kind of desktop documents that we have encountered so far.
– *Normalized view:* An algorithm to apply the rules to a document in order to create a normalized view of the document. The normalized view encodes all instances of the document. The running time of the algorithm is linear with the size of the document. Furthermore (and consequently), the size of the normalized view is linear with the size of the document.
– *Predicate-based index:* An algorithm to construct a predicate-based index from the normalized view. The index can be extended with positioning and scoring information

for more sophisticated information retrieval tasks (e.g., phrase searches and ranking).
– *Enhanced query processing:* An efficient sweep-line algorithm in order to process keyword and phrase search queries on predicate-based indexes.

The remainder of this paper is organized as follows: Sect. 2 discusses related work. Section 3 presents the rule language. Section 4 presents the query language. Section 5 details the normalization algorithm. Section 6 shows how the predicate-based indexes are constructed. Section 7 presents the sweep-line algorithm for query processing. Section 8 shows how individual instances of a query result are referenced (i.e., result aggregation). Section 9 shows how positioning information is encoded in a predicate-based index. Section 10 outlines how the framework could be extended for scoring and ranking. Section 11 gives the results of experiments that study the retrieval quality (precision and recall) and performance (space and running time overheads) of the proposed framework when compared to traditional search engines. Section 12 contains conclusions and possible avenues for future work.

2 Related work

This work is based on results from several other research projects on databases and information retrieval. To the best of our knowledge, however, no other existing system covers all the use cases we have looked at and is, for instance, able to handle the two motivating examples of the introduction and the other examples studied in this paper and the experiments.

Several use cases addressed in this paper have been studied in separate projects. For example, the PIX project at AT&T [4] has studied phrase search in the presence of comments and formatting instructions (i.e., the *Excluded* pattern described in Sect. 3). Phrase search has also been extensively studied in [29]. Information retrieval across versions has been investigated in [9]. Again, to the best of our knowledge, our work is the first comprehensive framework to deal with all these (and many other) use cases.

The Semantic Web also tries to provide a framework for adding meta-data in order to enhance search and query processing. Specifically, the Semantic Web defines new meta-data languages (i.e., RDF [41] and OWL [37]) and a new query language that operates on that meta-data (i.e., SPARQL [23]). Our work is less ambitious and more focused. Our goal is to merely annotate the data in order to describe semantic properties of the data and to have a simple query language based on keywords and phrases. Another, more theoretical, difference is that the Semantic Web relies on *description logic*, whereas our work relies on *propositional logic*. Description logic is much more powerful and undecidable in general. Propositional logic is not Turing-complete and can always be evaluated in polynomial time. We exploit this property for efficient query processing (Sect. 7). Propositional logic works for our purposes because we are focused on defining user views. We do not need to reverse engineer entire applications and describe their semantics—we only need to describe what a user *observes*.

Another line of related work are semantic search engines for XML; e.g., [19]. These engines extend traditional IR techniques for defining correlations between a query and the returned XML fragments. Another line of related work is Colorful XML [31]. Colorful XML encodes multiple *views* (or instances) into a single document and is, therefore, related to the normalized view devised in this work. The foundations for all these approaches are the possible worlds semantics which were first defined in [26]. Possible worlds are also the basis for probabilistic database systems such as TRIO [1]. As a result, work on storage structures for probabilistic database is related to our work on encoded multiple worlds (or instances) into a single document. The work of Koch et. all [5], for example, defines a similar storage structure as the one used in this work for the *Alternative pattern*. The problem of shared content in documents was also addressed by [11]. Finally, superimposed information is another technique which is applicable in this context [38].

With regard to the rule language used in this work, our language can be seen as a special case of a transformation language such as XSLT [17] or XQuery [10]. Indeed, we could have used one of these standard languages. We decided to use a simplification (Sect. 3) in order to ease implementation and because the full (Turing-complete) power does not seem to be needed.

The term *normalization* is borrowed from database theory [18]. Its goal is to eliminate redundancy. XML Normalization has been studied in [6]. Normalization is also a compression techniques for XML (e.g., XMill [35] or [13]). In order to store a *normalized view*, several XML serialization techniques are applicable; e.g., Colorful XML [31] and binary XML [24]. XML Views and XML summaries are also studied in [42] and DataGuides [27]. Special positioning techniques are part of work on relational [44], XML [39], or XML-IR systems [30].

Advanced indexing techniques for XML are proposed by [3], while query relaxation techniques are presented in [14,34]. The aim of those techniques is to increase specificity of results and to infer the semantics of the data by using IR-specific techniques. As opposed to [3], we use the Dewey ID positioning technique instead of structural indexes in order to optimize the computation of results with nested XML elements. Optimal join processing algorithms have been studied in [2,16,32]. For ranking, Top-k search (e.g., [45]) and ranking strategies in XML (e.g., [30]) have been

extensively studied in the literature. We have adapted [7] for the purpose of our prototype implementation.

3 View definition: patterns in desktop application data

As shown in the introduction, the challenge of desktop search is to understand and optimize the semantics of the data. Concretely, these semantics are expressed as an application view, consisting of a set of instances. The purpose of this section is to define a comprehensive set of rules that declaratively specify the set of instances for each document. These rules should be defined per application (e.g., Word, Wiki, Outlook, etc.) and the rules of an application should be applicable to all documents generated by that application.

In our experience, desktop data exhibits re-occurring patterns. That is, a few types of view definitions are sufficient to define the application views of all desktop data that we are aware of. We identified these patterns and created an XML-based rule language which can declaratively associate patterns to the data. The syntax of the language is not important; important is that a few patterns are sufficient. We used XML in order to represent documents because most document formats are either based on XML (e.g., all Office documents according to the latest Microsoft and OpenOffice specifications, Wikis, etc.) or have wrappers that transform the documents into XML (e.g., PDF, Latex, etc.). As an expression language for the rules, we use XQuery [10].

In our experience, it is rare that rules need to be customized for individual documents; instead, a small set of rules can be devised for all documents generated by an application. For example, only seven rules are needed for all Word documents. A different set of rules applies to the E-Mail repository of a client-like Outlook or Thunderbird. Again, it is not important that the rules model the whole application logic of a complex application like Word or the E-Mail client: The rules merely need to model the view users have on the data.

The patterns defined in this work are: *Excluded*, *Comment*, *Alternative*, *Version*, *Placeholder*, and *Field*. For each pattern, we provide syntax in order to specify rules for that pattern. Patterns and rules are explained in this section following the order of complexity. As a reference, the patterns which describe Examples 1 and 2 from Sect. 1 are *Field* and *Version*.

3.1 Excluded pattern

The first pattern is *Excluded*. It specifies that parts of a document are invisible to the user. These parts should not be considered in a search query. Examples are formatting instructions which a user is typically not interested in for searching. Users who wish to search for all documents that use a particular font would surpress the Excluded rule for fonts during query processing, as described in Sect. 7.3. An f pattern to exclude certain fragments from an XML document for phrase search has been proposed in the PIX project [4]. Figure 5 gives an example. Figure 5a shows the original data with the formatting instructions. Figure 5b shows the *view* (without formatting instructions) as it is relevant for most users. Figure 5c shows the rule that is used in order to declare that the formatting instructions of the original document should not be included in the view. The name of a rule describes the pattern and the *match* attribute contains an XQuery expression that defines to which elements the rule should be applied. Here and in the remainder of this paper, XML namespace declarations are omitted for brevity. Of course, the rule language defines its own namespace in order to avoid ambiguity with other XML names.

```
<office:font-decls>
   <font-decl style:name="Lucida Grande"/>
</office:font-decls>
Mickey likes Minnie.
```
(a) Original Data

```
Mickey likes Minnie.
```
(b) Instance

```
<excluded match="//office:font-decls" />
```
(c) Rule (R1)

Fig. 5 Excluded example

3.2 Comment pattern

The second pattern is *Comment*. Examples for Comments are footnotes or inlined annotations in a text. Figure 6 gives an example. Figure 6a shows the original document. Figure 6b and 6c show two instances that could be of interest to the user: one which contains the comment (identical to the original) and one that does not contain the comment. Figure 6d shows the rule that users can use in order to declare that they are interested to query the two instances separately. As for the *Excluded* pattern, the motivation for this pattern was provided by the PIX project [4]: Only the application of this pattern makes it possible to apply a phrase search for "Mickey likes Minnie", since the original document would not match. Both instances in Fig. 6 are therefore part of the view.

Comments are an example that demonstrate how the views can become larger than the original data. The number of instances in the view grows exponentially with the number of comment rules that declare comments in a document (e.g., in OpenOffice, separate rules would indicate that `note` and `comment` elements should be treated as comments).

```
Mickey <note id="ftn0">He is a Disney Character.</note>
likes Minnie.
```
(a) OpenOffice Document with a Footnote (Original Data)

```
Mickey <note id="ftn0">He is a Disney Character.</note>
likes Minnie.
```
(b) Instance including the Comment (Instance 1)

```
Mickey likes Minnie.
```
(c) Instance without the Comment (Instance 2)

```
<comment match="//note"/>
```
(d) Comment Rule

Fig. 6 Comment example

```
<option world="mouse">Mickey Mouse</option>
<option world="duck">Donald Duck</option>
likes
<option world="mouse">Minnie Mouse</option>
<option world="duck">Daisy Duck</option>.
```
(a) Original Data

```
<option world="mouse">Mickey Mouse</option>
likes <option world="mouse">Minnie Mouse</option>.
```
(b) World of Mouse (Instance 1)

```
<option world="duck">Donald Duck</option>
likes <option world="duck">Daisy Duck</option>.
```
(c) World of Duck (Instance 2)

```
<alternative match="//option" key="$m/@world"
                              optional="false" />
```
(d) Alternative Rule

Fig. 7 Alternative example

3.3 Alternative pattern

The *Alternative* pattern specifies that one out of several options of a text is chosen. For example, a song could contain markup that specifies for which audience (e.g., adults or general public) certain parts are targeted. Likewise, an electronic health record could contain markup that indicates which doctor is allowed to get what kind of information from the patient. The Alternative pattern is also useful to specify that a Web page could have versions in English and Chinese. The images and tables of the Web page would be identical for both versions, but the text should either be only in English or only in Chinese.

Figure 7 gives a simple example. The original document contains markup with *o*ption elements that indicate whether text fragments belong to the world of Mickey Mouse or Donald Duck. If the text of Fig. 7a is viewed from the perspective of the world of Mickey Mouse, it should read "Mickey Mouse likes Minnie Mouse" (Fig. 7b); otherwise it should read "Donald Duck likes Daisy Duck" (Fig. 7c).

The rule in Fig. 7d specifies exactly these semantics. It contains three properties:

– *match*: The match attribute contains an XQuery expression that describes the target items of the document that are affected by the rule; *option* elements in this example. Furthermore, the XQuery expression in the match attribute binds a variable, *$m*, that is used in the *key* specification.
– *key*: The key attribute contains an XQuery expression that uniquely identifies an alternative. For example, all text fragments that belong to the world of Mickey Mouse can be identified by the value "mouse" in the *world* attribute (the key) of *option* elements (the target of the rule).
– *optional*: This attribute contains a Boolean value and specifies whether an alternative should be generated that does not contain any *option* elements. In this example, the optional attribute is set to *false*. If it had been set to true, a third instance with content "likes". would have been generated (not shown in Fig. 7).

Logically, Alternative rules are evaluated in the following way. First, the whole document is inspected in order to find all key values (typically, atomic values such as strings and numeric values). For each such key value, an instance is generated by inspecting the document again and ignoring all target nodes that do not match that key value. If the *optional* attribute is set to true, an additional instance is generated that ignores (i.e., excludes) all target nodes.

Alternatives partition the document. In order to draw an analogy, the expression in the *key* attribute corresponds to the expression of a `GROUP BY` clause of a SQL query with the match expression acting as the `FROM` clause. As a result, the number of instances in a view specified using this pattern grows linearly with the number of key values specified in the document. It is worth mentioning that the Comment pattern is a special case of an Alternative pattern. It is given by an alternative with two options, one of which is empty.

3.4 Version pattern

The *Version* pattern is useful for use cases such as Example 2 of the introduction. In addition to tracking changes in office documents (Word and OpenOffice), this pattern can also be found in Wiki data. In contrast to the Alternative pattern which *partitions* the elements that match an Alternative rule, the Version pattern orders all matching elements and specifies that the view contains an instance for each subsequence.

The rule shown in Fig. 8e declares that elements selected by the *match* expressions (i.e., *insert* and *delete* elements) are versioned. The moments of time when versions occurred are defined by the *key* expression, which selects the values *6am* and *7am*.[1] The *action* attribute specifies where on the timeline, the content in the *insert* and *delete* elements is valid. For example, the content of *insert* elements is valid on and after the *key* time, specified by the literal *AFTER_AT* in the

[1] For brevity, we omit the date in the timestamps in this example.

```
<inserted id="1"> <info time="6am"/> </inserted>
<inserted id="2"> <info time="7am"/> </inserted>
<deleted id="1"> <info time="6am"/> </deleted>
Mickey <insert id="2">Mouse</insert>
likes <delete id="1">Minnye</delete>
<insert id="1">Minnie Mouse</insert>.
```
(a) Versioned OpenOffice Document (Original Data)

```
Mickey likes <delete id="1">Minnye</delete>
```
(b) "Mickey likes Minnye" (Instance 1 - misspelling)

```
Mickey likes <insert id="1">Minnie Mouse</insert>.
```
(c) "Mickey likes Minnie Mouse" (Instance 2 - correction)

```
Mickey <insert id="2">Mouse</insert> likes
<insert id="1">Minnie Mouse</insert>.
```
(d) "Mickey Mouse likes Minnie Mouse" (Instance 3)

```
<version match="//insert" action="AFTER_AT"
         key="//inserted[@id=$m/@id]/info/@time"/>
<version match="//delete" action="BEFORE"
         key="//deleted[@id=$m/@id]/info/@time"/>
```
(e) Version Rules

Fig. 8 Version example

action attribute. Similarly, the content of the *delete* elements is valid before the time indicated by the value of the *key* attribute, indicated by the literal *BEFORE* in the *action* attribute. The complete set of values for the *action* attribute are the literals *BEFORE*, *BEFORE_AT*, *AFTER_AT*, *AFTER* corresponding to the $<$, $\leq$, $\geq$, $>$ operators.

Figure 8 gives an example, again using the OpenOffice data format. Initially (before moment "6am"), the document was "Mickey likes Minnye" (Fig. 8b). Then, by a sequence of insertions and corrections (deletions), the document became "Mickey likes Minnie Mouse" (at moment "6am") and "Mickey Mouse likes Minnie Mouse" (at moment "7am"). The two rules shown in Fig. 8e capture exactly these semantics and ensure that each version can be queried as a separate instance, as outlined in the introduction.

3.5 Placeholder pattern

Placeholders specify that data stored in a different document or data stored within the same document at a different location should be inlined. In addition to a *match* attribute which specifies the target of the rule (as for all other rules), a Placeholder rule has a *ref* attribute that contains an XQuery expression, which specifies the data that should be inlined. Figure 9 gives a simple example. It shows an Office document with a footnote (Fig. 9a). In this document, the texts of footnotes are stored separately from the actual text. The Placeholder rule (Fig. 9c) brings a copy of the footnote text to the right location (Fig. 9b). As in the Alternative rule, the *match* expression binds a variable, $m, which can be used in the *ref* expression.

An application of a Placeholder rule transforms an XML document into exactly one instance. It is possible that the

```
<note id="ftn0">He is a Disney Character.</note>
Mickey Mouse <ref id="ftn0"/> likes Minnie.
```
(a) OpenOffice Document with a Footnote (Original Data)

```
<note id="ftn0">He is a Disney Character.</note>
Mickey Mouse
<ref id="ftn0">
   <note id="ftn0">He is a Disney Character.</note>
</ref> likes Minnie.
```
(b) Instance with a Copy of the Footnote Text

```
<placeholder match="//ref" ref="//note[@id=$m/@id]"/>
```
(c) Placeholder Rule

Fig. 9 Placeholder example

ref expression refers to a different document (e.g., a spreadsheet). Any XQuery expression is allowed in the *ref* expression. Furthermore, it is possible that the *ref* expression evaluates to a sequence of several items (values and nodes, possibly hundreds or even thousands). In that event, the whole sequence is inlined. How to generate a set of instances if the *ref* expression evaluates to a sequence is described in the next subsection (Field pattern).

In addition to Office documents, the Placeholder pattern occurs in object-oriented code such as Java. If a class, A, is derived from a class, B, and inherits properties (e.g., methods and class variables), these properties become searchable in the context of the source code of class A by using the Placeholder pattern.

3.6 Field pattern

A *Field* rule combines the behavior of an Alternative and a Placeholder rule. It is a useful rule to implement use cases such as that of Example 1 of the introduction. Figure 10a shows a Word document (a letter) referring to the Excel spreadsheet of Fig. 10b that contains a list of names; one instance of the letter is generated for each name (Fig. 10c, d). Figure 10e gives the Field rule that generates these two instances.

Like a Placeholder rule, a Field rule has a *match* and a *ref* expression. Furthermore, a Field rule has a *key* and an *optional* expression like an Alternative rule. The Field rule of Fig. 10e specifies that for each *line* in the Excel stylesheet "names.xls" an instance should be generated. In a Field rule, the *ref* expression binds a variable, $r, which can be used in the *key* expression. In this example, the key of each instance that is generated is the line number (i.e., the attribute id).

As in all other rules, the *match* expression binds a variable, $m, which can be used in the *ref* and *key* expressions. The use of the $m variable is not shown in the example rule of Fig. 10e. Furthermore, this example rule does not specify the *optional* attribute. By default, the *optional* attribute is set to *false* so that it need not be specified if that is the desired value.

```
<word>
Dear <recipient/>
The meeting is today at 2am.
Peter Jones Project Manager.
</word>
```
(a) Word document (Original Data)

```
<excel>
 <line id="1">Paul</character>
 <line id="2">Mary</character>
</excel>
```
(b) Excel Spreadsheet (Original Data)

```
<word>
Dear <recipient>Paul</recipient>
The meeting is today at 2am.
Peter Jones Project Manager.
</word>
```
(c) "Dear Paul..." (Instance 1)

```
<word>
Dear <recipient>Mary</recipient>
The meeting is today at 2am.
Peter Jones Project Manager.
</word>
```
(d) "Dear Mary..." (Instance 2)

```
<field match="//recipient" key="$r/@id"
ref="doc("names.xls")//line />
```
(e) Field Rule

Fig. 10 Field example

Figure 10c and d show the two instances that are generated if the Field rule of Fig. 10e is applied to the document of Fig. 10a. Just like Alternative rules, the number of instances generated by a Field rule grows linearly with the domain of the key expression.

3.7 Multiple rules and conflicts

The rule framework presented in this section extends naturally to applications in which several different rules are applicable to one document. It is even possible that the same or different rules are applied to nested elements inside a document. In such a scenario, two questions arise: (a) in which order should rules be applied; and (b) what happens if two (conflicting) rules are to be applied to the same matching element.

Following the SQL standard [43] and the proposal for the XQuery Update Facility of the W3C [15], we use Snapshot semantics. Snapshot semantics specify that rules are applied in two phases. In the first phase, all matching elements are marked and the expressions referring to each matching element are pre-computed using the original version of the document(s). If two rules are in conflict and match the same element in the first phase, then the process aborts with an error. In the second phase, the instances of the document are constructed, thereby combining the pre-computed transformations of the matching elements of the first phase. This step is implementation dependent. As shown in the following sections, the instances are not physically constructed in our implementation.

Snapshot semantics served the purposes of all use cases that we have looked at so far. Nevertheless, it is worth-while to study alternative models. For instance, a model that supports composability of conflicting rules can reduce the number of classes that are needed; e.g., the Field rule could be expressed as a composition of Alternative and Placeholder rules, applied to the same elements. We plan to study such models as one avenue for future work.

4 Query language

This section defines the operators of the query language studied in this work. As opposed to traditional keyword search, these operators are applied to instances, instead of files. Consequently, the results of queries are instances (not files). Other than that, the query language is straight-forward and strongly resembles the query language used in traditional search products such as Google.

Retrieval model. This work focuses on the Boolean retrieval model. According to this model, a search engine returns an instance if and only if the query keywords appear in the instance's content. This work also briefly addresses ranking, but the main contribution lies in techniques to effect Boolean information retrieval. Studying issues of ranking in more detail is an important avenue for future work.

The main operators of the query language for Boolean retrieval are presented in Table 1 and explained in the following.

AND. This operator corresponds to the "," operator in search engines like Google. The input of this operator is a list of two or more keywords. The result is a list of *instances* which contain all of the keywords given as input.

OR. This operator implements disjunctions. The input of this operator is a list of two or more keywords. The result is a list of *instances* which contain either of the keywords given as input.

NOT. This operator implements negation. The input of this operator is a single keyword. The result is a list of *instances* which do *not* contain the keyword.

NEXT. This operator implements phrase search. As opposed to keyword search, this operator returns all *instances* in which all keywords given as input are contained and adjacent. For brevity, this paper often uses quotation marks as an equivalent syntax to represent phrase search. The NEXT

operator can be composed with all other operators as part of complex search queries and queries which combine keyword and phrase search.

5 Normalization of documents

The first step in the system outlined in Fig. 4 of the introduction is the creation of a *normalized view* for every document. The normalized view is the basis for all further steps: indexing and query processing. The goal is to have a representation of the documents that captures the affects of all rules that are applicable to the document so that all further steps can be carried out independent of the presence of any rules. As stated in the introduction, the naïve approach of simply generating all instances is not viable because the number of instances grows exponentially with the number of rules, thereby resulting in wasted space for storing the instances, large crawl times to generate the instances, and large query processing times to scan indexes that reference all materialized instances. Instead, this section shows how such normalized views can be created without actually creating any instances and with linear space and time overhead in the number and size of the documents. The key idea is to identify the common parts of a document which are not affected by any rules and *factor out* all variable parts of the document which are affected by the rules. Syntactically, the variable parts are annotated in the document in a uniform way that encodes the effects of applying a rule.

5.1 Examples

Normalization can be described best using an example. Figure 11 shows the normalized view for the example of Fig. 7. This normalized view encodes the two instances "Mickey likes Minnie" (Fig. 7b) and "Donald likes Daisy" (Fig. 7c). The key idea of normalization is as follows: markup the *variable* parts of the data using special `select` elements. In this example, all *option* elements are tagged in this way, indicating that all *option* elements are variable. Common parts of the original data (e.g., "likes"), which are not affected by any rule, are not tagged.

As shown in Fig. 11, each `select` element contains a predicate that specifies in which kind of instances of the view its content should be considered.[2] The predicate `R3=mouse`, for instance, specifies that the `<option world = "mouse"> Mickey </option>` and `<option world = "mouse"> Minnie </option>` elements should be included into the instance that represents Mickey Mouse's perspective on the original data; however, these two elements should not

```
<select pred="R3 = mouse">
   <option world="mouse">Mickey Mouse</option>
</select>
<select pred="R3 = duck">
   <option world="duck">Donald Duck</option>
<select> likes
<select pred="R3 = mouse">
   <option world="mouse">Minnie Mouse</option>
</select>
<select pred="R3 = duck">
   <option world="duck">Daisy Duck</option>
<select>.
```

Fig. 11 Normalized view—alternatives

```
<inserted id="1"> <info time="6am"/> </inserted>
<inserted id="2"> <info time="7am"/> </inserted>
<deleted id="1"> <info time="6am"/> </deleted>
Mickey
<select pred="R4 >= 7am">
  <insert id="2">Mouse</insert>
</select>
likes
<select pred="R4 < 6am">
  <delete id="1">Minnye</delete>
</select>
<select pred="R4 >= 6am">
  <insert id="2">Minnie Mouse</insert>
</select>
```

Fig. 12 Normalized view—versions

be included into the instance that represents Donald Duck's view of the world. Here, *R3* is a variable with a generic name that is generated for the Alternative rule of Fig. 7d. In general, each rule is uniquely identified in this way and the identifiers of rules are used as variables in the `pred` attributes of a `select` annotation generated by the rule. *mouse* and *duck* are the values of the evaluation of the *key* expression of the Alternative rule of Fig. 7d. The predicate specifies an equality because the rule of Fig. 7d is an Alternative rule, and equality corresponds to the semantics of the Alternative pattern.

Another example of a normalized view is given in Fig. 12. This normalized view encodes all the instances of the view described in Fig. 8. Again, `select` elements define the variable parts of the document and predicates specify the inclusion of such variable content in instances. In case of *insert* elements, which correspond to the *match* expression of the Version rule with an `AFTER_AT` action (Fig. 8e), the predicates involve the "$\geq$" operator. Similarly, *delete* elements match the Version rule with an *BEFORE* action so that the predicates involve the "$<$" operator. Here, *R4* is the identifier generated for *both* Version rules of Fig. 8e. The time values in the predicates are computed from the *key* expressions of those rules. The results of the *key* expression must belong to a domain on which a total ordering can be defined.

5.2 Normalization algorithm

Algorithm 5.1 shows the details on how the normalized view of a document is constructed, given the document and a

[2] Again, the namespaces of the qualified names of `select` elements and `pred` attributes are omitted for brevity.

Algorithm 5.1 Normalization Algorithm

```
1: Input: Document d, Rule[] R
2: Output: Document NV {Normalized View}

3: Function constructTaggingTable(Document d, Rule[] R): TT
4: Output: TT(ruleId, pattern, nodeId, keyvalue, op, optional) {Tagging Table}
5: LET matches = { }
6: for all rule ∈ R do
7:   if rule.type = "Alternative" OR rule.type = "Comment" OR rule.type = "Excluded" then
8:     rule.operator = "="
9:   end if
10:  for all m ∈ rule.match(d) do
11:    if $m ∈ matches then
12:      'ERROR: Conflict!' {Node already matched by other rules.}
13:    else
14:      if rule.ref = NULL then
15:        key = rule.key($m)
16:        if key != NULL OR rule.type = "Comment" then
17:          INSERT INTO TT(ruleID, pattern, nodeId, keyvalue, op, optional)
           VALUES (rule.ID, rule.pattern, $m.nodeId, key, rule.operator, rule.optional)
18:        end if
19:      else
20:        refs = rule.ref($m) {Evaluate key on each referenced node}
21:        for all $r ∈ refs do
22:          key = rule.key($m, $r)
23:          if key != NULL then
24:            INSERT INTO TT(ruleID, pattern, nodeId, refId, keyvalue, operator, optional)
             VALUES (rule.ID, rule.pattern, $m.nodeId, key, rule.operator, rule.optional)
25:          end if
26:        end for
27:      end if
28:    end if
29:  end for
30:  matches := matches ∪ rule.match(d) {Adding matches of current rules to set of matched items.}
31: end for
32: end Function

33: Function constructNormalizedView(Document d, Rule[] R, TaggingTable TT): NV
34: Output: Document NV {Normalized View}
35: for all $n ∈ $d do
36:   if ∃ $row ∈ TT where $n.nodeID = $row.nodeID then
37:     if $row.pattern = "Alternative" OR $row.pattern = "Version" OR $row.pattern = "Comment" OR $row.pattern = "Field" then
38:       $select = new Node(<option pred = "$row.ruleID $row.operator $row.keyvalue"/>)
39:       $select.content = $n
40:       NV.append($select)
41:       if $row.pattern = "Comment" OR $row.optional = "true" then
42:         $select := new Node(<option pred = "$row.ruleID $row.operator NULL"/>)
43:         NV.append($select)
44:       end if
45:     else if $row.pattern != "Excluded" then
46:       NV.append($n)
47:     end if
48:   end if
49: end for
50: end Function
```

set of rules as input. According to the snapshot semantics described in Sect. 3.7, the algorithm operates in two phases. The first phase constructs the so-called *tagging table*. The tagging table has an entry for each element of the document that matches the *match* expression of a rule. In addition to the identifier of the element, each entry of the tagging table contains the identifier of the rule and the *pattern* of the rule. Depending on the pattern, in addition, the entry can contain the *key* value, as computed by evaluating the *key* expression of the rule (Line 15 or 22 of Algorithm 3.7). Furthermore, an entry of the tagging table has an *operator* which is set to *equality* for all patterns except for the Version pattern. For the Version pattern, the operator is set according to the `action` specified in the Version rule (e.g.,

Table 3 Tagging table (alternatives example)

Rule	Pattern	NodeId	KeyValue	Op
R3	Alternative	1	mouse	=
R3	Alternative	2	duck	=
R3	Alternative	4	mouse	=
R3	Alternative	5	duck	=

Table 4 Tagging table (versions example)

Rule	Pattern	NodeId	Key value	Op
R4	Version	2	7am	>
R4	Version	5	6am	<
R4	Version	7	6am	≥

$<$ or $\geq$). If an element matches more than one rule, the process is terminated and an error is returned (Lines 11 and 12 of Algorithm 5.1), according to the semantics of conflicts defined in Sect. 3.7.

Continuing the examples from Figs. 7 and 8 of Sects. 3.3 and 3.4, Tables 3 and 4 show the corresponding tagging tables. It can already be seen that the entries of the tagging table contain all the information needed in order to annotate the elements in the normalized view.

Indeed, the second phase of normalization (function constructNormalizedView in Algorithm 5.1) is merely a traversal of the tagging table, thereby creating a `select` element for each entry of the tagging table and inserting that `select` element into the document in order to annotate the matching element (referenced by its NodeId). The `pred` attribute of the `select` element is generated by the identifier of the rule (e.g., *R*3), the operator (e.g., =), and the KeyValue (e.g., "mouse"). This logic is conceptually straightforward and described in Lines 37–47 of Algorithm 5.1.

It should be noted that the application of rules to nested elements results in nested `select` annotations. Logically, such nesting corresponds to a conjunction of conditions under which the inner element occurs in instances of the document. This aspect is revisited in Sect. 6 when the construction of the predicate-based index is described.

Even though Algorithm 5.1 contains a procedural notation for the implementation of normalization, both the construction of the tagging table and the construction of the normalized view can be implemented quite easily using a declarative language such as XSLT or XQuery. For instance, the implementation used for the performance experiments (Sect. 11) was based on Microsoft's XSLT processor which is integrated into the .NET framework.

5.3 Discussion

The normalized view, as described in this section, has the following advantages:

Completeness. The normalized view encodes all the instances of the (potentially huge) application view, in a way which is independent of the rules that specify the view. This way, the normalized view can serve as a basis for indexing and all further query processing. Having such a rule-independent representation of data makes it possible to extend the rule language as new patterns become important, without adjusting the indexing and query- processing components.

Compactness. The normalized view is a compact representation of the original data. As each element of the document is annotated at most once, the size of the normalized view is linear to the size of the original document (at most a factor of two with regard to the number of elements). It is even possible that the normalized view is smaller than the original document if the Excluded pattern is applied.

The normalized view can be further compressed in order to reduce its size. For example, we used dictionary-based compression in the implementation of our prototype. As a result, the two selections *mouse* and *duck* can be represented by a single Bit in the example of Fig. 11. Furthermore, it is not necessary to serialize the normalized view as XML; in our implementation, we used a serialized XML format that is based on tokenization [25]. Such an implementation reduces the size of the normalized view and significantly speeds up query processing because no parsing is required.

Instance Computation. From the normalized view, all instances can be computed by instantiating the variables (e.g., *R*3 and *R*4) and evaluating the `pred` attributes accordingly. The "Mickey likes Minnie" instance of the normalized view of Fig. 11, for example, can be retrieved by instantiating the *R3* variable to "mouse". Similarly, the second version of the document in Fig. 8 can be computed by instantiating the *R4* variable to "6am". As part of instantiation, all elements which are embraced by `select` elements whose `pred` evaluates to *false*, are removed. This way of referencing individual instances is exploited for indexing (next section) and result aggregation (Sect. 8).

6 Enhanced indexing

This section describes how to extend a conventional inverted file index in order to index the normalized views of a collection of documents. In other words, this section describes how to index all the instances of a collection of documents. Again, the naïve approach is to index each instance separately. This approach is impractical for the same reasons as the naïve approach to materialize all instances is impractical (discussed in the previous section). Again, as with normalized views, the key idea is to *factor out* the common and variable parts in the index and to encode the instances in

Table 5 Enhanced inverted file—alternative (Fig. 7)

Keyword	DocId	Predicate
Mickey	d_1	$R3 =$ "*mouse*"
Donald	d_1	$R3 =$ "*duck*"
likes	d_1	*true*
Minnie	d_1	$R3 =$ "*mouse*"
Daisy	d_1	$R3 =$ "*duck*"

Table 6 Enhanced inverted file for versions (Fig. 8)

Keyword	DocId	Predicate
Mickey	d_2	*true*
likes	d_2	*true*
Minnye	d_2	$R4 <$ 6am
Minnie	d_2	$R4 \geq$ 6am
Mouse	d_2	$R4 \geq$ 6am
Mouse	d_2	$R4 \geq$ 7am

which variable parts occur using predicates. This way, common parts which are present in all instances of a document are indexed only once.

6.1 Example

Table 5 shows the extended inverted file of the example of Fig. 7. In addition to the document ID (d_1 is used in this example to refer to the original data of Fig. 7a), keyword and possibly other properties such as position information and scores (not shown), the enhanced inverted file keeps a predicate that logically specifies in which instances the keyword appears. These predicates are derived from the predicates specified in the `select` elements of the normalized view. The keyword "likes" appears in all instances; correspondingly, it is associated with the predicate *true*. All other keywords inherit the predicate(s) of their surrounding `select` element(s).

Table 6 shows the inverted file for the versions example of Fig. 8. The keywords *Mickey* and *likes* appear in all instances. As a result, the predicate is *true* for these keywords. The keyword *Minnie* only occurs after the typo has been corrected at timestamp 6am. Correspondingly, the condition of this entry is $R4 \geq 6am$. Likewise, the typo *Minnye* only occurs in instances with timestamp before 6am. For the keyword *Mouse*, two entries are constructed; one for each occurrence. Judging from Table 6, it may seem overkill to have two entries for the same keyword and document. For an exact Boolean retrieval model with phrase search, however, it is important to have both entries because both entries have different positioning information. For instance, it is important to have both entries in order to infer that the phrase query "Mickey likes Minnie" only matches Version 2 of the document (between 6am and 7am), whereas the phrase query "Mickey Mouse likes Minnie Mouse" only matches the latest version of the document (after 7am). Positioning information is not shown in Table 6. It is discussed in detail in Sect. 9. Two index entries may only be merged if they involve the same document, keyword, and positioning information.

6.2 Index creation algorithm

The algorithm to construct an enhanced index with predicates is straightforward and omitted for brevity. Again, the logic looks complicated, but the principles are straightforward. Essentially, the algorithm scans through the normalized view, picks up all keywords, adds an entry for each occurrence of the keyword and the conjunction of the predicates of all embracing `select` elements. That is, if a keyword, k, occurs in three nested `select` elements with, say, predicates p_1, p_2, and p_3, then an index entry for k with predicate $p_1 \wedge p_2 \wedge p_3$ is generated.

This way to construct indexes is compact and efficient in the same way as the normalized view is a compact and efficient way to construct and represent the set of instances associated with a document. Just as in traditional information retrieval, it is possible to *fuzzify* the index for a more compact representation and better query performance. In many scenarios, it is too wasteful to keep multiple entries for the same keyword and document and keep exact positioning information. Traditional *fuzzification* techniques can be applied to such enhanced, predicate-based indexes in the same way as to traditional inverted file indexes. Studying the impact of such *fuzzifications* is beyond the scope of this paper and an important avenue for future work. The performance experiments reported in Sect. 11 indicate that acceptable performance can be achieved for desktop search even with precise indexes and without fuzzifications.

7 Enhanced query processing

This section discusses how simple keyword queries (without the NEXT operator for phrase search) can be processed using the enhanced inverted file index described in the previous section. Phrase search is discussed in Sect. 9. The basic algorithms are almost the same as for traditional information retrieval systems [8]. In this work, we have adopted the approach to partition the inverted file by keyword and to sort the resulting inverted lists by document id. The inverted lists for a specific query can then be merged in a similar way as a merge join in a relational database system with the document id being the join key [12]. The difference to traditional information retrieval is that the predicates of the index entries need to be processed when index entries of different inverted lists need to be merged. Furthermore, we propose a special

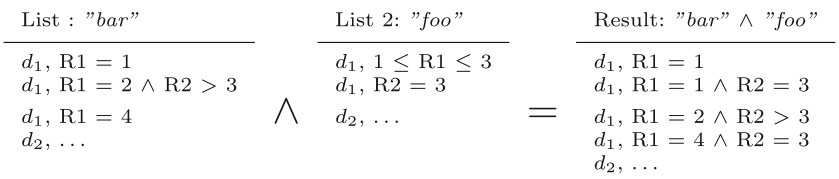


Fig. 13 Processing conjunctions with multiple rules

merge algorithm in order to deal with multiple index entries for the same keyword and document (e.g., there are multiple entries for *Mouse* in Table 6).

7.1 Example

Two simple examples that demonstrate how the predicates of index entries are used during query processing were already given in the introduction (Sect. 1.3). Figure 13 shows a more complex example that involves multiple index entries in each inverted list and more complex predicates that involve more than one variable. For each document (only the entries for Document d_1 are shown in Fig. 13), all combinations of index entries are considered when the two lists are merged. Since our framework is constrained to propositional logic (all predicates are conjunctions of the form *variable operator constant*), consolidating the predicates of two (or more) index entries is simple and can be carried out in linear time with the number of conjuncts. In particular, it is simple to detect the infeasibility of a conjunction such as $R1 = 4 \wedge 1 \leq R1 \leq 3$ when merging the second entry of the *bar* list and the first entry of the *foo* list in Fig. 13. Even processing disjunctions as part of *OR* queries (Sect. 4) can be carried out in a linear and straightforward way. In any way, the result lists (as the input lists) are kept in disjunctive normal form so that each entry is a conjunction of predicates.

With regard to performance, it is critical to process a large number of entries for the same document efficiently. In order to do that, we propose the use of a sweepline algorithm which is described in the remainder of this section.

7.2 Sweepline algorithm

This section describes the algorithm used to intersect multiple lists of keywords. As mentioned in the beginning of this section, the inverted lists are sorted by document id so that they can be merged efficiently. For extended information retrieval, this sort-merge algorithm needs to be extended. In addition to the document ids, the predicates in the entries of the inverted lists must be processed. This extension can be achieved as follows:

1. Each predicate on each variable is represented as an interval. For instance, the predicate R1 = 1 can be represented as R1 ∈ [1], where [1] is a point in the domain of Variable R1. Likewise, the predicate $1 \leq R1 \leq 3$ can be interpreted as R1 ∈ [1; 3]. If R1 is not constrained, then R1 ∈ $(-\infty; +\infty)$ holds (or more generally, using the minimum and maximum values of the domain).
2. At crawl time, when the index is created, one variable is selected to be the sort key for each document. Then, all entries for that document are sorted by the lower left bound of the corresponding interval. For instance, if an inverted list has several entries for a document and Variable R3, ([7;10], [3;4], [8;9], [6], [2;4], [5;9]), then these entries are sorted in the following way: L1 = ([2;4], [3;4], [5;9], [6], [7;10], [8;9]).
3. At query time, an interval join is applied in order to merge two interval lists for the same document. We chose an interval join algorithm that is based on the plane sweep paradigm [40]. That is, a sweep area (as defined in [21]) is used to efficiently perform the merge on the two interval lists. For instance, if we want to intersect list L1 (from the example of the previous paragraph) with a second list L2=([1;3], [3;7], [5;6], [6;7], [8;10]), the intervals can be visualized as shown in Fig. 14.
   The intervals in the upper part of Fig. 14 represent intervals of L2 and the intervals in the lower part of the figure represent intervals of L1. The interval join moves a sweep line along the R3 axis. Whenever it hits the lower left bound of an interval (representing a predicate), it inserts that interval into a so called sweep area structure [21]. Whenever the sweep area hits the upper bound of an interval, it removes that interval from the appropriate sweep area structure. Details of this algorithm can be found in [7,21,40].

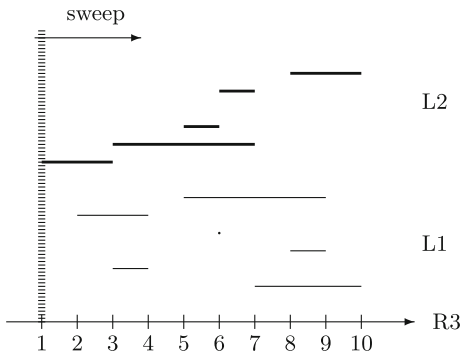


Fig. 14 Intervals on sweep area

This sweepline algorithm merges index entries according to a single dimension (e.g., variable *R*3 in the example of Fig. 14). As shown in Fig. 13, index entries may involve multiple variables and it is possible that two index entries overlap

in one dimension and do not overlap in another dimension. In order to finalize the merge between two index entries, therefore, the other dimensions need to be taken into account, too. This situation is analogous to a multi-dimensional sort-merge join in relational databases: One join predicate, the so-called primary join predicate, is favored and the other join predicates need to be applied separately for all tuples that match the primary join predicate. As for relational sort-merge joins, performance is best if the most selective join predicate is chosen as the primary join predicate. Likewise, the most selective dimension (or variable) should be selected in order to sort the index entries of a document and for carrying out the sweepline algorithm.

In all experiments that we conducted so far, this algorithm was efficient enough to meet the performance requirements of modern information retrieval (Sect. 11). There are, however, several additional optimizations that can be applied. First, if only points (rather than intervals) are present, then a traditional sort-merge join can be applied. This opportunity arises if no Version rules have been applied to the relevant documents. Second, in some situations, multi-dimensional join techniques (e.g., [7,20]) can be applied. We plan to study the trade-offs of the different join techniques as part of future work.

7.3 Search modes

As mentioned in the introduction, it is possible to toggle between different search modes. It is possible, for instance, to let the user decide whether the search should be carried out across all versions of a document or in each version of the document individually, depending on the intent of the user or the requirements of an application. Implementing this toggling is straightforward by *disabling* rules during query processing. For instance, inter-version search can be carried out by disabling all Version rules. During query processing, a rule can be disabled by simply ignoring all predicates that involve the identifier of that rule; in other words, predicates that involve the identifier of that rule are set to *true*. Doing so, can be done on the fly as part of merging the inverted lists. In particular, this toggling can be implemented using the same enhanced index. That is, no dedicated indexes for different search modes need to be constructed: One enhanced index for all search modes is sufficient.

8 Result aggregation

The last step of query processing is *result aggregation* (Fig. 4 in Sect. 1.3). The goal of result aggregation is to provide users with links to specific instances that match a query, if not all instances of a document match the query. Result aggregation can be carried out in a straight-forward way. A result tuple contains a predicate which can be used to instantiate variables in the normalized view. For instance, the first entry of the result list of Fig. 13 indicates that Document d_1 matches the *foo, bar* query, if $R1 = 1$. Consequently, the normalized view of d_1 can be instantiated using $R1 = 1$ and random settings for all other variables. Just like the predicates in the `select` elements of a normalized view are a compact representation of all instances of a document, the predicate of a query result is a compact representation of all the instances of a document that match a query.

9 Phrase search

Processing phrase queries is an important part of modern Information Retrieval systems. Unfortunately, the traditional flat positioning scheme does not work well with predicate-based indexing: Adjacency must be defined taking the predicates and the *levels* of `select` elements in the normalized view into account. In order to encode adjacency in the presence of `select` elements (whose contents may or may not appear in an instance of the document), we propose to use a hierarchical positioning scheme with *Dewey codes*.

9.1 Example

To illustrate the need for a new positioning scheme, Fig. 15 shows the traditional (flat) positioning information for the normalized view of our running versioning example (Fig. 12). We omit the *insert* and *delete* elements of the normalized view for brevity; they do not contain any keywords and are therefore not relevant. Figure 15 shows why flat positioning does not work in our context. Although the phrase "Mickey Mouse likes Minnie Mouse" matches Instance 3 of the document (any version after 7am), there is not enough information in the flat positioning scheme to detect this. The keyword *likes* is at Position *2* and the keyword *Minnie* is at Position *4*. In Instance 3 (after the deletion of *Minnye*), however, these two keywords are adjacent. This deficiency of the traditional positioning scheme is not an artifact of using the normalized view: The same problem arises if positions are associated with keywords in the original document. Furthermore, the same kind of deficiency arises for all the other patterns. The remainder of this section presents a better encoding of positioning information that allows to reason about potential *gaps* such as the one between *likes* and *Minnie*.

9.2 Enhanced positioning scheme

The proposed positioning scheme is based on the following principles:

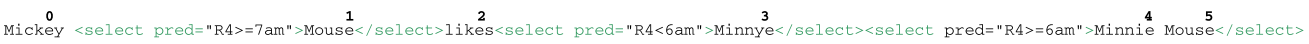


Fig. 15 Traditional (flat) positioning scheme (Fig. 8)

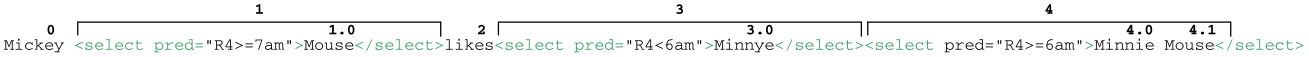


Fig. 16 Dewey IDs (Fig. 8)

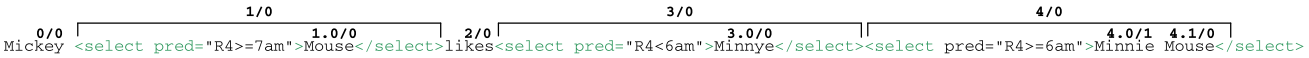


Fig. 17 Dewey IDs/next level (Fig. 8)

Table 7 Inverted file with Dewey IDs (version example)

Keyword	DocId	Predicate	Dewey Pos.	Next level
Mickey	d_2	*true*	0	0
likes	d_2	*true*	2	0
Minnye	d_2	$R4 < 6\text{am}$	3.0	0
Minnie	d_2	$R4 \geq 6\text{am}$	4.0	1
Mouse	d_2	$R4 \geq 7\text{am}$	1.0	0
Mouse	d_2	$R4 \geq 6\text{am}$	4.1	0

Table 8 Skip spaces (version example)

Keyword	DocId	Predicate	Dewey Pos.	Next level
" "	d_2	$R4 < 7\text{am}$	1	0
" "	d_2	$R4 \geq 6\text{am}$	3	0
" "	d_2	$R4 < 6\text{am}$	4	0

1. **Each** `select` **element has a position.** A select element and all its descendants consume exactly one position at the level at which the `select` occurs. Figure 16 shows the corresponding numbering scheme of (top-level) keywords and `select` elements.
2. **Hierarchical positions**. Keywords within a `select` element are numbered according to a Dewey encoding scheme [39]. For instance, *Minnie* is associated to the position *4.0* because it is the first keyword in the `select` element at position *4*. Nested `select` elements (not shown in this example) would be numbered in the same way as keywords in `select` elements. In the presence of nested `select` elements, the Dewey codes can become arbitrarily long. Table 7 shows the extended inverted index with the Dewey position of each keyword. (The *level* column of Table 7 is described in Point 4, below.)
3. **Skip Spaces**. For each `select` element of the normalized view an additional *Skip Entry* is inserted into the inverted index. The three skip entries for the three `select` elements of the example of Fig. 16 are shown in Table 8. A skip entry encodes under which condition a `select` element can be skipped; i.e., which instances do *not* contain the contents of the `select` element. Therefore, a skip entry has the *negation* of the predicate associated to the `select` element. Skip entries have no keyword because they represent *empty* content.
   As a technicality, negating a predicate that is composed of several conjuncts results in a predicate with several disjunctions. In order to ensure that all predicates in the inverted file have conjunctions only (no ORs), several skip entries are generated if the predicate associated to a `select` element involves several conjuncts: one for each conjunct.
4. **Levels**. There are two cases in which two keywords may be adjacent:
   - The two keywords are at the same *level* (i.e., embraced by no or the same `select` element) and are next to each other. In this case, their Dewey code only differs in the last position. Examples are *Minnie* (Dewey code 4.0) and *Mouse* (Dewey code 4.1).
   - They are at different levels and the levels are compatible.

In order to deal with the second case, the *next level* at which all adjacent keywords can be found is recorded for each keyword. Figure 17 shows this *next level* information for each keyword. For instance, any adjacent (NEXT) keyword to *Mickey* must be found at Level 0, either within the following `select` element (i.e., *Mouse*) or after this (and potentially other) `select` elements (i.e., *likes*). In general, two keywords, p and q with Dewey IDs $p_0.p_1.\ldots.p_n$ and $q_0.q_1.\ldots.q_m$ and levels l_p and l_q are possibly adjacent if and only if

$$(\forall i = 0..n-1 : p_i = q_i) \wedge p_n + 1 = q_m \wedge l_p = l_q \wedge n = m$$
$$\text{OR } (\exists l : \forall i = 0..l-1 : p_i = q_i) \wedge p_l + 1 = q_l$$

There are several simplifications which can be used in order to reduce computation and storage requirements: First, the *next level* can be inferred from the Dewey ID and the

presence of an adjacent (following) keyword within the same `select` element. A single bit is, therefore, sufficient to store this information. Second, Dewey IDs can be compressed as proposed in [39]. Third, there are various ways to speed up the evaluation of the adjacency equations. Trailing zeroes can be ignored on the right side of the adjacency equation. Furthermore, the next level information is only needed when the next keyword is at a higher level.

It should be noted that the positioning scheme described in this section has some shortcomings. For instance, it does not support distance functions that ask for keywords that are within a certain range (greater than 1). Furthermore, the scheme is not update friendly: If a document is updated, the whole document needs to be re-indexed. In order to decrease the overhead of incremental updates, interval-based ID schemes such as XISS [33] could be used. Studying these extensions is beyond the scope of this paper.

9.3 Processing phrase search queries

This section describes how the query processing phase described in Sect. 7 is extended in order to take Dewey IDs and skip spaces into account for phrase search queries. Logically the query $w_1 NEXT w_2$ is equivalent to a (recursive) join query between the inverted list of all entries that match w_1, zero, one, or more times the skip entries, and the inverted list of all entries that match w_2. Denoting the skip entries as " ", this recursive query can be described by the following regular expression:

$w_1(\text{""})^* w_2$

As join predicates, the adjacency predicate of the previous subsection needs to be considered. Furthermore, of course, the logic of consolidating predicates described in Sect. 7 needs to be applied. In particular, the sweep line algorithm of Sect. 7 continues to be applicable.

In the worst case, this operation (i.e., a recursive query) is quite expensive, but it is always polynomial with the size of the document. In practice, the overhead is tolerable because the number of adjacent skip entries (i.e., adjacent `select` elements in the normalized view) is limited. We will revisit query processing performance in Sect. 11 when we present the results of performance experiments.

10 Ranking

The focus of this work has been on Boolean retrieval. According to that model, a document (or an instance of a document) either matches a query or it does not. Nevertheless, the model can naturally be extended to a model that assigns a score to each document / instance, determining how well the document / instance matches a query. Such an approach is particularly beneficial if the inverted index is *fuzzified* for efficiency as described in Sect. 6.

Adding scoring and ranking of search results is (almost) orthogonal to the techniques presented in this paper. In particular, many different scoring schemes can be incorporated. Doing so involves adding *weights* to entries in the inverted index. For instance, such weights could be based on the *term frequency (tf)* and *inverse document frequence (idf)*, as in traditional information retrieval systems [8]. During query processing, these weights need to be merged in addition to the predicates and the positioning information. Section 11.7 presents some experimental results we carried out with such a scoring and ranking scheme. Overall, we believe that ranking is less relevant for desktop search because the number of results that match according to a Boolean retrieval model is limited in this context.

11 Experiments and results

We implemented the techniques presented in the previous sections (normalization, indexing, and extended query processing with and without phrase search) and experimentally compared our implementation with traditional search techniques, the naïve approach to integrate rules, and a commercial search engine. Correspondingly, we refer to these approaches as "Enhanced" (with rules), "Naïve" (with rules), and "Traditional" (no rules). As a commercial search engine *Windows Desktop Search* was used (denoted as WDS). The goal of the experiments was to show that (a) our extended approach gives better search results than traditional search; (b) the space and processing overhead over traditional search is tolerable; and (c) the overhead of the naïve approach is not tolerable.

11.1 Experimental setup

Software and hardware used. The indexing engine (i.e., crawler, normalization and index creation) and query processor were implemented in Visual Studio.NET using the Microsoft .NET Framework 2.0. All experiments were performed on an IBM ThinkPad T42, with a Pentium-M 1.7 GHz processor and 1 GB of RAM under Windows XP Professional.

Experimental data. For the experiments reported in this paper, we crawled data collections from the authors' personal desktops. The data involved E-Mail (Outlook), CVS (Java source code), and LaTeXfiles. Furthermore, we crawled TWiki and Wikibooks data. For all these kinds of data, open-source tools were available in order to convert them to XML so that rules according to the syntax of Sect. 3 could be applied.

Table 9 Patterns in experimental data sets

Data collection	Patterns	Description	Trad.search produces false positives	Trad.search produces false negatives
LaTeX	*Annotation,*	Annotated footnotes,		Yes
	Placeholder	External includes		Yes
E-mail	Alternative	Group by message and by conversation thread		Yes
Java code	*Placeholder,* *Annotation*	Modeling inheritance		Yes
Twiki	*Version*	Versioned documents	Yes	
CVS	*Version*	Versioned software repos.	Yes	
Wikibooks	*Version*	Versioned information	Yes	

Table 10 Experimental data collections

Data collection	Size (MB)	No of files	No.of instances
LaTeX	13.2	1093	1473
E-Mail	45.9	1	26692
Java code	182	6624	8005
Twiki	4.54	410	3743
CVS	5.92	827	1442
Wikibooks	79.7	9846	72576

Table 9 shows the most important patterns that can be found in each kind of data. For example, the LaTeXdata involves footnotes and an incorrect interpretation of footnotes results in *false negatives* for phrase searches. That is, traditional search engines will miss a document even though it matches a query and expose poor recall. The Version pattern (as of Example 2 in the introduction) can be found in the Twiki, CVS, and Wikibooks data. If that pattern is not interpreted correctly, *false positives* may be returned. As a result, traditional search engines may have poor precision for this kind of data. Java code has the Placeholder pattern; due to inheritance, a class may have more properties than listed in its source code (similar to Example 1 of the introduction). Again, misinterpreting inheritance may result in poor recall.

The size of the data collections is given in Table 10. The data varies from 4.54 MB in case of TWiki to 182 MB in case of Javacode and contains several levels of nesting of the elements carrying patterns.

Experimental queries. Since the data was personal data from the authors' desktops, the queries were handcrafted. The complete list of queries is given in Fig. 18. The queries tried to test a variety of scenarios, including queries with a high and a low recall and queries that involve frequent and rare keywords and any combination of these. Furthermore, the queries tried to test the correct interpretation of the patterns. All queries contained between two and five keywords. Precision and recall were computed relative to the enhanced, predicate-based index approach presented in this paper.

11.2 Precision and recall

The main goal of this work is to increase the quality of queries in terms of precision and recall. Figure 19 shows the precision and recall of the three alternative approaches (Traditional, Enhanced, Naïve) for the various data sets. Again, the precision and recall of Enhanced and Naïve were set to 1 and the precision and recall of Traditional was computed relative to Enhanced. One way to compute the precision and recall of Traditional search is to consider how many relevant documents are returned as compared to the Enhanced and Naïve methods, as shown in Fig. 19a and b. These figures show that Traditional has merely a recall of 50% for the Java code. Again, the reason is that Traditional misinterprets inheritance. For the LaTeXand E-Mail collections, the recall of Traditional is 80%. Again, misinterpreting the placeholder pattern in this data is the cause for this sub-optimal recall.

With regard to precision, Traditional is fine for all data that does not exhibit the Version pattern (i.e., LaTeX, E-Mail, and Java code) and shows fairly high precision if the precision metric is regarded in the granularity of whole documents (Fig. 19a). An alternative way to measure precision is to take the number of instances as a baseline, as shown in Fig. 19c. This figure shows that the precision drops dramatically if this way of computing precision is used. This phenomenon indicates that Traditional search is indeed able to locate the right documents, but it is very difficult to find the right version of the document that matches a query with Traditional search engines. Again, this phenomenon has been described as part of both examples in the introduction.

For completeness, Fig. 19d shows the recall of all three approaches on a per-instance basis. Obviously, Traditional performs better in terms of recall if a per-instance metric is used because for Traditional *all* instances of a document are considered to be part of a query result. (Traditional search is not able to identify individual instances of a document.)

Fig. 18 Experimental queries

Queries Latex		Queries E-Mail		Queries Javacode	
Q1	query data shipping	Q1	meeting sigmod	Q1	public hashcode
Q2	query shipping	Q2	kossmann meeting	Q2	main class thread run
Q3	database settings	Q3	important message	Q3	abstract query
Q4	sans serif typeface	Q4	delos vorschlaege cnr pisa	Q4	interface equals
Q5	data	Q5	ecdl healthcare phone conf	Q5	abstract getclass
Q6	query	Q6	sigmod conference	Q6	public static void main
Q7	data and query shipping	Q7	reviewer feedback	Q7	getsource main
Q8	query data structure	Q8	important message	Q8	interface class protected public
Q9	database structure	Q9	phone conference	Q9	init close abstract
Q10	setting the structure	Q10	meeting conference	Q10	public private protected getstore

Queries Twiki		Queries CVS		Queries Wikibooks	
Q1	hrs ifw	Q1	if then else	Q1	ewan
Q2	main peterfischer	Q2	hashcode interface	Q2	columbus
Q3	printer install	Q3	public static class interface	Q3	chocolate
Q4	author	Q4	main gettokeniterator	Q4	chemistry
Q5	new version	Q5	getstore interface	Q5	karl wick
Q6	create database	Q6	class interface	Q6	visual basic
Q7	main peterfischer cristianduda	Q7	localcall	Q7	american culture
Q8	main peterfischer printer install	Q8	rtscontrol rts	Q8	new mexico chile
Q9	main peterfischer create database	Q9	public void int	Q9	kwhitefoot programming
Q10	main peterfischer hrs ifw	Q10	public static void main int	Q10	create media

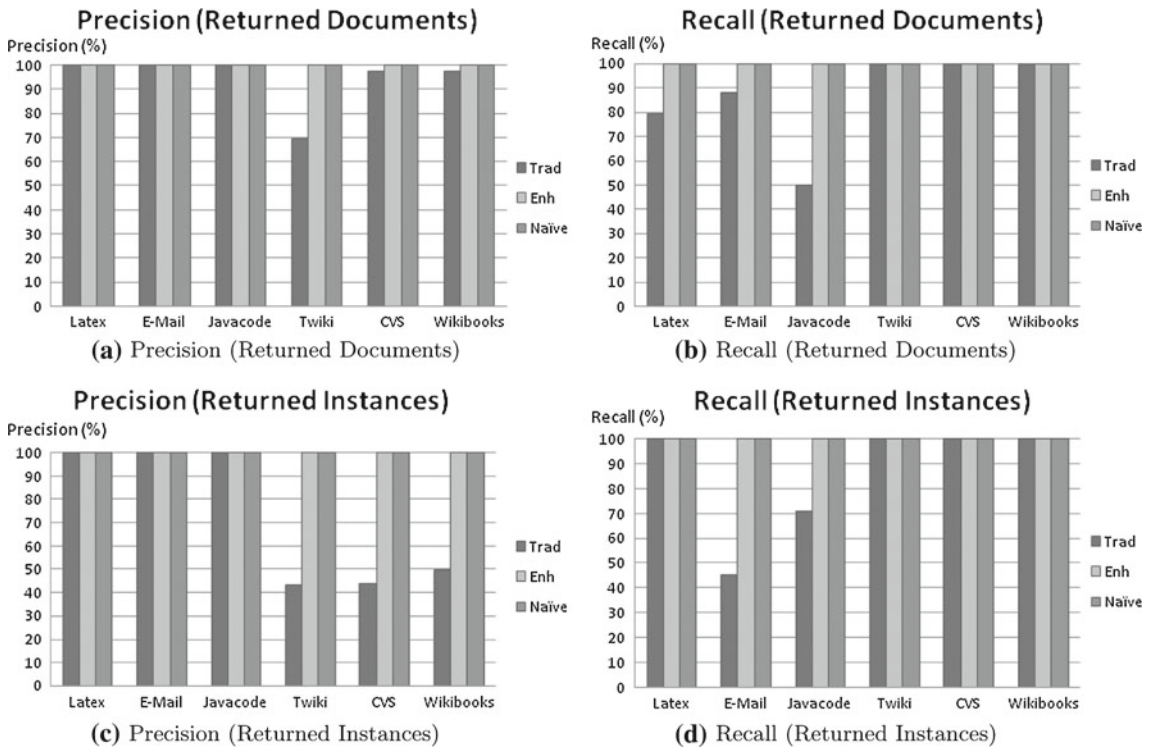


Fig. 19 Precision and recall for experimental data sets

11.3 Index Size

The second set of experiments assessed the space overhead of the Enhanced and Naïve approaches. Tables 11 and 12 summarize the results. Table 11 gives the index sizes for an index without positioning information; Table 12 gives the results including positioning information (Sect. 9). As a baseline, these two tables also include the index size of WDS

Table 11 Index size (MB)

Collection	Trad index	Enh index	Naïve index	WDS index Enh/Trad	Overhead Enh/Naïve (%)	Space Gain Enh/Naïve (%)
LaTeX	2.19	2.81	3.7	19.1	×1.28	24
E-mail	21.1	33.4	50.4	484	×1.58	34
Java Code	6.6	7.08	8.16	68	×1.07	13
Twiki	0.70	1.30	4.62	27.3	×1.86	72
CVS	1.05	1.18	2.23	17.7	×1.12	47
Wikibooks	22.1	43.6	256	664.5	×1.97	83

Table 12 Index size with positioning info (MB)

Collection	Trad index	Enh index	Naïve index	WDS index	Overhead Enh/Trad	Space gain Enh/Naïve (%)
LaTeX	6.81	10.3	13	19.1	×1.51	21
E-mail	33.9	60.8	96.8	484	×1.79	37
Java Code	22.9	27.1	32.2	68	×1.83	15
Twiki	2.23	4.68	11.7	27.3	×2.09	60
CVS	2.77	3.51	6.91	17.7	×1.26	49
Wikibooks	54.7	207	581	664.5	×3.78	64

Table 13 Index creation time (s)

Collection	Trad	Enh. +Normaliz.	Naïve+Inst. materializ.	Overhead Enh/Trad	Gain Enh/Naïve (%)
LaTeX	18.15	23.83	33.33	×1.31	29
E-mail	110	163	21080	×1.49	99
Java Code	125.8	149	204.5	×1.18	27
Twiki	5.25	7.12	40.126	×1.36	82
CVS	7.9	8.36	30	×1.06	72.13
Wikibooks	129.21	219.107	1605.9	×1.70	86

Table 14 Index creation time with positioning info (s)

Collection	Trad	Enh. +Normaliz.	Naïve+Inst. materializ.	Overhead Enh/Trad	Gain Enh/Naïve (%)
LaTeX	20.27	28.62	37.63	×1.41	24
E-mail	148	182	21116	×1.23	99
Java Code	131.63	162.06	233	×1.23	19
Twiki	6	8.23	41.611	×1.39	80
CVS	8.98	9.79	31.276	×1.09	68
Wikibooks	123.12	525	1731.5	×4.26	70

(a commercial product) and of a handcoded traditional index. For WDS, the use of positioning information could not be configured.

The results are pretty straight-forward. Without positioning information, the space overhead of Enhanced when compared to the (handcrafted) Traditional approach is tolerable; it is a factor of two in the worst case (Twiki). With positioning information, the overhead of Enhanced as compared to Traditional is up to a factor of four in the worst case (Wikibooks). The Naïve approach is consistently worse than Enhanced. In the worst case by a factor of almost six (Wikibooks, no position information).

11.4 Index creation time

The third set of experiments assessed the time to create the index and the normalized view. Tables 13 (without positioning) and Table 14 (with positioning) show the results; all times are given in milliseconds. The overhead of Enhanced versus Traditional is still acceptable in both cases. Only for

Table 15 Normalized view vs. materialized instances

Collection	Trad:Original size (MB)	Enh:Norm. view (MB)	Naïve:Mat. instances (MB)	Gain(%) by norm.
LaTeX	13.2	19.4	25.4	23.62
E-mail	45.9	46.8	5252.12	99.11
Java Code	182	201	307.72	34.68
Twiki	4.54	4.3	22.7	81.06
CVS	5.92	6.9	16.5	58.18
Wikibooks	79.7	102	1117	90.90

Table 16 Query proc. time (ms)

Collection	Trad.	Enh.	Naive	WDS	Overhead Enh/Trad	Overhead Naïve/Enh (%)
LaTeX	1.38	1.61	2.47	29.31	×1.17	34.8
E-mail	4.17	4.73	11.08	7.28	×1.13	57.3
Java Code	3.91	4.84	6.04	23.65	×1.24	19.9
Twiki	0.82	1.24	2.68	31.56	×1.51	53.7
CVS	1.75	2.01	2.81	14.27	×1.48	28.5
Wikibooks	2.35	3.20	12.86	26.46	×1.36	75.1

Table 17 Query proc. time (ms), phrase search

Collection	Trad.	Enh.	Naive	WDS	Overhead Enh/Trad	Overhead Naïve/Enh (%)
LaTeX	6.03	12.76	8.41	21.86	×2.12	0
E-mail	4.17	12.88	13.45	8.94	×3.08	4.2
Java Code	7.88	15.69	5.72	23.65	×1.99	0
Twiki	1.38	9.28	3.43	29.64	×6.72	0
CVS	3.72	8.64	2.75	11.86	×2.32	0
Wikibooks	3.69	8.62	14.47	26.46	×2.34	40

the Wikibooks data set, the overhead is substantial (factor of 4). The Wikibooks data involves applications of rules to nested elements; the maximum depth of nested `select` elements is 500. In all other cases, the data is fairly flat and the overhead of the Enhanced approach is below a factor of 1.5. The Naïve approach, again, is consistently worse than the Enhanced approach. In the extreme case (E-Mail), it took two orders of magnitudes more to materialize the instances and index the materialized indexes than it took to construct the normalized views and index that.

11.5 Space gain through normalization

Table 15 shows that the size of the normalized view is comparable with that of the original data (denoted as *Trad:Original* in Table 15). The space required to store the materialized instances using the Naïve approach, however, is prohibitive. For the E-Mail data, the compression rate of normalization is above 99%.

11.6 Query processing time

Table 16 presents the average running times (in milliseconds) of keyword queries (no positional information needed). Table 17 shows the query processing times for phrase search when the positional information is used. It is obvious that the running times of an Enhanced search are longer as compared to the running times of a Traditional desktop search engine because the indexes that need to be scanned are larger, and the logic that merges two inverted lists is more complex (Sect. 7).

The differences in running times depend on the data set, on the rules applied, and more importantly, on the type of query. In almost all cases, however, the overhead of the Enhanced approach is tolerable. The times are given in milliseconds and all times are clearly fast enough for human interaction. More specifically, the response times of the Enhanced approach are always below 15 ms, even for phrase search. Even the Naïve approach has acceptable response times of always less than 15 ms.

11.7 Ranking

Throughout this work, we focused on the Boolean retrieval model. In order to study the impact of ranking, Table 18 presents the recall results of a small Top k experiment. For this experiment, we only considered the Top k results returned by the Enhanced approach using the traditional tf/idf

Table 18 Recall of enhanced, *Topk* Results

Collection	Top 1 (%)	Top 2 (%)	Top 3 (%)	Top 4 (%)	Top 5 (%)
LaTeX	6	12	18	23	29
E-mail	16	30	32	34	35
Javacode	9	19	28	34	37
Twiki	18	30	35	43	45
CVS	31	61	63	65	66
Wikibooks	22	29	29	30	30

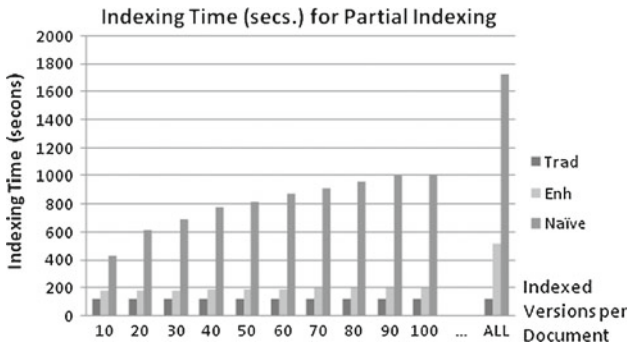


Fig. 20 Indexing time (s), partial indexing

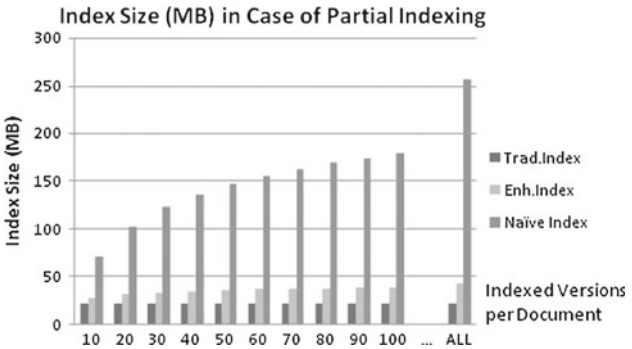


Fig. 21 Index size (MB), partial indexing

ranking model. This experiment studied how fast the relevant results are returned. The results are encouraging: the recall is what could be expected from similar experiments with traditional search engines. Therefore, it seems that the Enhanced indexing and query processing mechanisms introduced in this work have no impact on the scoring and ranking mechanism so that scoring and ranking can be seen as an orthogonal issue and improvements in scoring and ranking models are directly applicable to our approach, too.

11.8 Partial indexing

As shown in the previous section, there is a tradeoff between the quality of query results (precision and recall) and the space and time overhead of index creation and query processing. The Traditional and Enhanced approaches represent extreme points in the spectrum of possible alternatives. An approach to trade better performance for worse query results is to *fuzzify* the Enhanced index. To study the impact of such a fuzzification, we studied variants of the Enhanced index for the Wikibooks data set. In these variants, we varied the number of versions of each Wikibook document from 10 to 100. The results are shown in Figs. 20, 21, 22, 23, 24. As a baseline, these figures also show the complete Enhanced approach that indexes all variants, denoted as *all* in the figures.

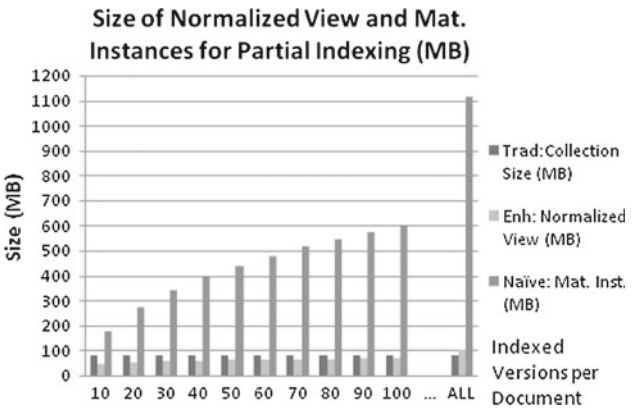


Fig. 22 Size of normalized view, partial indexing

Index creation time. Figure 20 shows the index creation time for 10 to 100 indexed versions per Wikibook document. The results are not surprising; the effort grows roughly linearly with the number of versions that are indexed. For the Enhanced approach, the index creation time increases from 105 s (for 10 versions) to 206 s (for 100 versions). Again, the Enhanced approach is much better than the Naïve approach and has tolerable overhead as compared to the Traditional approach. Obviously, the index creation time of the Traditional approach is constant, independent of the number of indexed versions.

Index size. Figure 21 shows the size of the index as a function of the number of indexed versions. Again, the results are

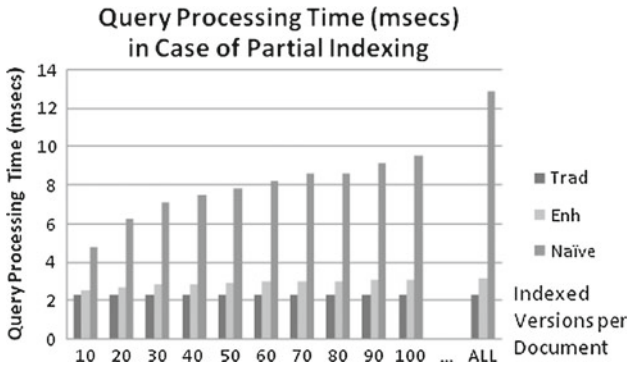


Fig. 23 Query processing time (s), partial indexing

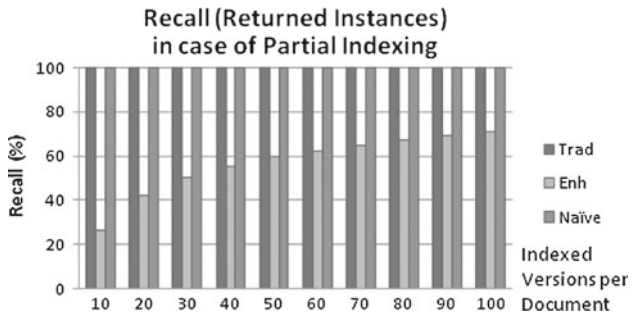


Fig. 24 Recall (returned instances), partial indexing

as expected. The size of the index of the Enhanced approach grows from 25 MB (for 10 versions) to 50 MB (for 100 versions). As in all other experiments, the Enhanced approach significantly outperforms the Naïve approach and is worse (yet, tolerably) than the Traditional approach. Comparing the fuzzified index sizes with the size of the full Enhanced index, it can be seen that the savings of fuzzification are not significant.

Space gain by normalization. Figure 22 shows the size of the normalized view (Enhanced) as compared to the size of the original data (Traditional) and the space requirements of materialzing all instances (Naïve). Again, there are no surprises. Normalization is important even for fuzzified indexes that only contain partial information.

Query processing time. Figure 23 shows the average running time of queries (keyword queries, no positioning information needed) with a varying degree of fuzzification. Again, it can be seen that the savings that can be achieved by partial indexing are not significant.

Result quality. Finally, Fig. 24 shows the recall of the three alternative approaches with a varying degree of fuzzification of the index. Here, it can be seen that for the Enhanced approach, indeed, the recall drops significantly if less versions are indexed. In summary, we conclude that partial indexing does not seem to be beneficial. The performance improvements are moderate, whereas the reduction in query result quality is substantial.

12 Conclusions and future work

This work was motivated by the observation that current desktop search engines see data with different eyes than users. As a result, traditional desktop search engines return wrong results in many scenarios. The main contribution of this work was to provide a framework that teaches desktop search engines to see the data with the same eyes as the user and to extend search indexes and query processing to efficiently implement this framework. Experiments showed that indeed the improvements in the quality of query results can be substantial and that the space and time overhead is tolerable.

There are several avenues for future work. One avenue is to apply more powerful query paradigms to normalized views, i.e., extend keyword search to XQuery. Furthermore, XML information retrieval approaches, such as query relaxation and structure-based search, could be explored, as described in Sect. 2. Adjusting indexing and search based on the type of the annotations is another optimization idea that we plan to study as part of future work. Another important direction for future work is to apply these techniques to other classes of applications, such as Scientific data, electronic health records, and even enterprise application data.

Another direction is to apply the techniques proposed in this work to the "Hidden Web". Rules could be seen as a modern version of `robots.txt` files that describe how to interpret the data found by the crawler. In order to be practical, it is important to create libraries of rules for documents generated by popular applications (e.g., Microsoft Office, Sun's OpenOffice, Twiki, LaTeX, E-Mail, CVS, etc.).

References

1. Agrawal, P., Benjelloun, O., Sarma, A.D., Hayworth, C., Nabar, S., Sugihara, T., Widom, J.: Trio: a system for data, uncertainty, and lineage. In: VLDB (2006)
2. Al-Khalifa, S., Jagadish, H.V., Patel, J.M., Wu, Y., Koudas, N., Srivastava, D.: Structural joins: a primitive for efficient XML query pattern matching. In: ICDE (2002)
3. Amer-Yahia, S., Curtmola, E., Deutsch, A.: Flexible and efficient XML search with complex full-text predicates. In: SIGMOD (2006)
4. Amer-Yahia, S., Fernández, M.F., Srivastava, D., Xu, Y.: PIX: a system for phrase matching in XML documents. In: ICDE (2003)
5. Antova, L., Koch, C., Olteanu, D.: 10^{10^6} Worlds and beyond: efficient representation and processing of incomplete information. In: ICDE (2007)
6. Arenas, M., Libkin, L.: A normal form for XML documents. TODS **29**, 195–232 (2004)

7. Arge, L., Procopiuc, O., Ramaswamy, S., Suel, T., Vitter, J.S.: Scalable sweeping-based spatial join. In: VLDB (1998)
8. Baeza-Yates, R.A., Ribeiro-Neto, B.A.: Modern Information Retrieval. ACM Press/Addison-Wesley, New York (1999)
9. Berberich, K., Bedathur, S.J., Neumann, T., Weikum, G.: A time machine for text search. In: SIGIR (2007)
10. Boag, S., Chamberlin, D., Fernndez, M.F., Florescu, D., Robie, J., Simon, J.: XQuery 1.0: an XML query language W3C candidate recommendation, 3 November (2005). http://www.w3.org/TR/xquery/
11. Broder, A.Z., Eiron, N., Fontoura, M., Herscovici, M., Lempel, R., McPherson, J., Qi, R., Shekita, E.J.: Indexing shared content in information retrieval systems. In: EDBT (2006)
12. Brown, E.W., Callan, J.P., Croft, W.B.: Fast incremental indexing for full-text information retrieval. In: VLDB (1994)
13. Buneman, P., Grohe, M., Koch, C.: Path queries on compressed XML. In: VLDB (2003)
14. Carmel, D., Maarek, Y.S., Mandelbrod, M., Mass, Y., Soffer, A.: Searching XML documents via XML fragments. In: SIGIR (2003)
15. Chamberlin, D., Florescu, D., Robie, J.: XQuery update facility W3C working draft 27 January (2006). http://www.w3.org/TR/xqupdate/
16. Chang, K.C.C., won Hwang, S.: Minimal probing: supporting expensive predicates for top-k queries. In: SIGMOD Conference (2002)
17. Clark, J.: XSL Transformations (XSLT), Version 1.0. http://www.w3.org/TR/xslt
18. Codd, E.F.: Further normalization of the data base relational model. IBM Research Report **RJ909** (1971)
19. Cohen, S., Mamou, J., Kanza, Y., Sagiv, Y.: XSEarch: a semantic search engine for XML. In: VLDB (2003)
20. Dittrich, J.P., Seeger, B.: GESS: a scalable similarity-join algorithm for mining large data sets in high dimensional spaces. In: KDD (2001)
21. Dittrich, J.P., Seeger, B., Taylor, D.S., Widmayer, P.: Progressive merge join: a generic and non-blocking sort-based join algorithm. In: VLDB (2002)
22. Duda, C., Graf, D.A., Kossmann, D.: Predicate-based indexing of enterprise web applications. In: CIDR (2007)
23. Eric Prud'hommeaux, A.S.: SPARQL Query Language for RDF, W3C Working Draft 23 November (2005). http://www.w3.org/TR/rdf-sparql-query/
24. Florescu, D., Hillery, C., Kossmann, D., Lucas, P., Riccardi, F., Westmann, T., Carey, M.J., Sundararajan, A., Agrawal, G.: The BEA/XQRL streaming XQuery processor. In: VLDB (2003)
25. Florescu, D., Hillery, C., Kossmann, D., Lucas, P., Riccardi, F., Westmann, T., Carey, M.J., Sundararajan, A., Agrawal, G.: The BEA/XQRL streaming XQuery processor. In: VLDB (2003)
26. Fuhr, N., Rölleke, T.: A probabilistic relational algebra for the integration of information retrieval and database systems. ACM Trans. Inf. Syst. **15**(1), 32–66 (1997)
27. Goldman, R., Widom, J.: DataGuides: enabling query formulation and optimization in semistructured databases. In: Jarke, M., Carey, M.J., Dittrich, K.R., Lochovsky, F.H., Loucopoulos, P., Jeusfeld, M.A. (eds.) VLDB (1997)
28. Google Search Engine. http://www.google.com
29. Detecting spam documents in a phrase based information retrieval system. United States Patent Application
30. Guo, L., Shao, F., Botev, C., Shanmugasundaram, J.: XRANK: Ranked Keyword Search over XML Documents. In: SIGMOD (2003)
31. Jagadish, H.V., Lakshmanan, L.V.S., Scannapieco, M., Srivastava, D., Wiwatwattana, N.: Colorful XML: One Hierarchy Isn't Enough. In: SIGMOD (2004)
32. Kaushik, R., Krishnamurthy, R., Naughton, J.F., Ramakrishnan, R.: On the integration of structure indexes and inverted lists. In: ICDE (2004)
33. Li, Q., Moon, B.: Indexing and querying XML data for regular path expressions. In: VLDB 2001 (2001)
34. Li, Y., Yu, C., Jagadish, H.V.: Schema-Free XQuery. In: VLDB (2004)
35. Liefke, H., Suciu, D.: XMill: an efficient compressor for XML data. In: SIGMOD (2000)
36. Live Search. http://www.live.com
37. McGuinness, D.L., van Harmelen, F.: OWL Web ontology language overview. http://www.w3.org/TR/owl-features/
38. Murthy, S., Delcambre, L., Maier, D.: Representing superimposed information in a conceptual model. In: ACM Document Engineering DocEng (2006)
39. O'Neil, P., O'Neil, E., Pal, S., Cseri, I., Schaller, G., Westbury, N.: ORDPATHs: insert-friendly XML node labels. In: SIGMOD '04 (2004)
40. Preparata, F.P., Shamos, M.I.: Computational Geometry: an Introduction. Springer, New York (1985)
41. RDF, Resource Description Framework. http://www.w3.org/RDF/
42. Shanmugasundaram, J., Kiernan, J., Shekita, E.J., Fan, C., Funderburk, J.E.: Querying XML Views of Relational Data. In: VLDB (2001)
43. SQL. ISO/IEC. SQL 1999. 9075-1:1999
44. Tatarinov, I., Viglas, S., Beyer, K.S., Shanmugasundaram, J., Shekita, E.J., Zhang, C.: Storing and querying ordered XML using a relational database system. In: SIGMOD (2002)
45. Theobald, M., Schenkel, R., Weikum, G.: An efficient and versatile query engine for TopX search. In: VLDB (2005)
46. Yahoo Search Engine. http://www.yahoo.com