Keyword Search in Application Data [*]

[Technical Report. November 2006]

Jens-Peter Dittrich, Cristian Duda, Björn Jarisch,
Donald Kossmann, Marcos Antonio Vaz Salles
ETH Zurich, Switzerland
http://www.dbis.ethz.ch

Abstract

Google and other products have revolutionized the way we search for information on the Internet, Intranet, and on our desktop. However, the current generation of search products does not exploit the structure and semantics of data, as defined by application programs (e.g., Word or Excel) that generate the data. This paper shows how search technology can be enhanced with implicit predicates, in order to take into account the structure and semantics defined by applications. Better search results are produced with tolerable performance overhead, while at the same time maintaining the simplicity of the keyword search interface.

1 Introduction

Almost everybody uses web search engines, like Google and MSN Search, to find documents on the Internet. Recently, derivatives of those products have also become popular in order to search for documents on the desktop of a PC. These desktop search engines use inverted indexes that relate keywords to the documents in which they appear [6].

The problem that motivates this work is that Google and related search products do not *see the data that they index with the eyes of the user*. Users access data through an application such as Word, Excel, Wiki (Web Browser), or an E-Mail client. These applications store the data in a proprietary format, encoding certain properties of the data such as annotations (e.g., comments in Word), versions (e.g., in a Wiki), E-Mail threads and folders, and so on. Each application then provides an *application view* of that data through an interface to navigate in it. In contrast to that, search engines crawl the file system and index the data without knowing about the properties (e.g., annotations and versions) encoded into the data or into the views provided by the applications. As a result, search engines potentially return documents that are not relevant or miss documents even though they are relevant. This situation is shown in Figure 1.

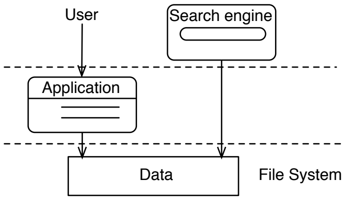

Figure 1. User View vs. Search Engine View

This work aims to extend today's search technology so that a search engine *sees* the data with the eyes of an application. Rather than indexing raw data, the extended search engines index *application views* on that data. As the data in these views may contain a significant amount of redundant information, one important challenge is to provide techniques to materialize and index those views. In order to achieve acceptable performance, the key idea of this paper is to *normalize* the application views and index these normalized views. In spirit, this normalization is related to the normalization of relational [10] and XML data [5]; the difference is that it is applied to text rather than to tables and XML. This paper shows how normalized

[*] A short version was published in ICDE 2007 as: "Bringing Precision to Desktop Search: A Predicate-based Desktop Search Architecture

views can be created and indexed for information retrieval, and how keyword search queries can be processed on them. This paper is focused on improving desktop search, but the principles can be extended to other application classes.

This paper is structured as follows: Section 2 gives motivating examples that demonstrate why today's search engines often return the wrong results on application data. Section 3 gives an overview of the proposed solution. Section 4 shows possible application views that can be generated on application data for desktop search. Sections 5 to 7 describe the main contributions of this work: normalization of application views, indexing of normalized views, and keyword search query processing. Section 8 contains the results of performance experiments with various types of data (i.e., LaTeX, Wiki, and E-Mail). Section 9 discusses related work. Section 10 concludes and suggests avenues for future work.

2 Motivating Examples

Example 1 (Bulk Letters) Figure 2 shows a snippet of a Word document (in simplified XML format) and a spreadsheet. The Word document contains a letter to be sent to several people. The spreadsheet specifies the names of the people that should receive the letter. Logically, the Word application generates a letter for each row in the spreadsheet, shown in Figure 3.

```
Dear <recipient/>
 The meeting is today at 2PM.
Peter Jones, Manager
```

(a) Bulk Letter

FirstName	LastName
Paul	Smith
Michael	Connor

(b) Excel Spreadsheet

Figure 2. Word Bulk Letter

```
Dear <recipient>Paul</recipient>,
 The meeting is today at 2PM.
Peter Jones, Manager.
```

(a) Bulk Letter: Instance 1

```
Dear <recipient>Michael</recipient>,
 The meeting is today at 2PM.
Peter Jones, Manager.
```

(b) Bulk Letter: Instance 2

Figure 3. Instances of Bulk Letter in Figure 2

Such Word and Excel documents can be found frequently in an office environment. Nevertheless, the current generation of desktop search engines returns wrong results for such data. For example, a Boolean retrieval query that asks for all documents that involve the terms *Paul* and *meeting* fails because neither the Word document nor the spreadsheet contain both terms. For the user, however, the Word document is relevant because the user did invite Paul to a meeting. Furthermore, the user might wish to see the letter shown in Figure 3a, in addition to or instead of the original Word document of Figure 2a.

This example demonstrates why desktop search engines should index *application views* rather than the raw data. An application view is a set of *instances* which can be generated from the original data. Furthermore, this example shows the need to *normalize* (compress) the data in an application view. It would be wasteful to index both instances of Figure 3 individually because the two letters are almost identical. The common parts of the two instances should be *factored out* and indexed only once; only the *variable* parts (the *recipient*) should be indexed separately. The more instances are generated from the letter (i.e., the larger the Excel spreadsheet), the more important this optimization becomes. □

```
<deleted id="1"> <info date="11/14/2005"/> </deleted>
<inserted id="2"> <info date="11/14/2005"/> </inserted>
<delete id="1">Mickey likes Minnie</delete>
<insert id="2">Donald likes Daisy</insert>
```

Figure 4. Versioned OpenOffice Document

Example 2 (Versioned Documents) Figure 4 shows a snippet of an OpenOffice document. OpenOffice (like Word) supports the tracking of changes and the incorporation of comments. This added information might, however, mislead a search engine. In Figure 4, the original version of the document was *"Mickey likes Minnie"*. Then, by a deletion and an insertion, the document was changed to *"Donald likes Daisy"*. Today's desktop search engines would consider this document to be relevant for the query *"Mickey likes Daisy"* because all three keywords can be found in the document. However, in fact, no version of this document was relevant for this query. Again, the solution is to index the *instances* of the *application view*; that is, each version of the document, rather than the raw data. Again, normalizing is important because such instances potentially have a great deal of commonalities that should be factored out in order to provide efficient search. □

3 Overview of Our Solution

In order to solve the problems stated above, we propose to extend desktop search engines as shown in Figure 5. Crawling and querying of the data is performed in the steps described below.

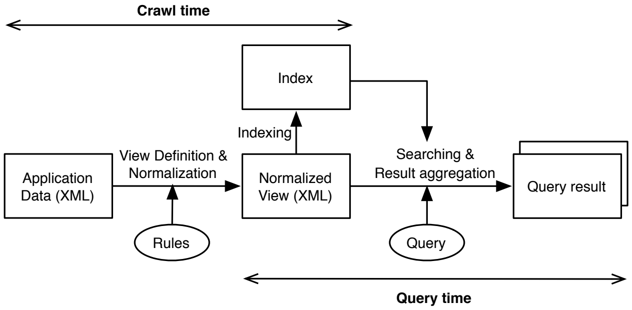

Figure 5. Enhanced Desktop Search

View Definition. Rules are applied that define the views on the application data. The rule language that is used in this work is based on XSLT and XQuery. It is sketched in Section 4 and fully described in [11].
Normalization. Since the application views may be very large (even for only a few Megabytes of raw data (Section 8)), they are never materialized. Instead, these views are directly normalized. The normalization is described in Section 5 and results in one normalized view for each application view. The normalized view is typically only slightly larger than the original data (as shown in Section 8) so that it can be materialized.
Indexing. Indexes (extensions of inverted lists) are built on the normalized view. The structure of these extended inverted files is described in Section 6. Again, these extended inverted files are moderately larger than inverted files for traditional information retrieval (Section 8).
Searching. At query time, the extended inverted files are used in order to find matching documents for a keyword query. Again, an extended version of the classic *merge* algorithm of inverted lists is used. This extended algorithm is described in Section 7.
Result Aggregation. The normalized view is used in order to compute the query results. Using the normalized view, it is possible to generate the *instance* of Figure 3a (i.e., the letter to Paul, as sent to Paul and defined in the application view) and refer to the Word document of Figure 2 (i.e., the definition of the bulk letter). Both kinds of results may be relevant for the user.

Throughout this work, we assume that the application data is represented using XML. This assumption simplifies the definition of the rules; i.e., variations of XSLT and XQuery can be used. In practice, this assumption can easily be met because almost all applications provide an XML interface. Microsoft Office and OpenOffice, for instance, are capable of producing XML; furthermore, Wiki data and E-Mail can easily be serialized into XML. In our experiments (Section 8), we also used LaTeX documents and converted them to XML.

4 Patterns in Application Data

From our experience, desktop data exhibits certain patterns, i.e., a few types of view definitions (rules) are sufficient to define the application views on this data. Further, typically the same set of rules can be applied to all documents of the same kind. Only one set of rules is necessary for all Word documents; a different set of rules would apply to the E-Mail repository of an IMAP server. In our experience, it is very rare that rules need to be defined for individual documents. In general, rules could be defined by the application provider and made available to users. This section gives a brief overview on some of these rules by examples. The exact definition of the rule language is given in [11]. Our language is based on XSLT and XQuery, but, as it is not a transformation language, it does not have (or need) the full expressive power of XSLT and XQuery.

4.1 Excluded content

The simplest pattern is Excluded. It specifies that parts of a document should not be considered in a search query. Examples are formatting instructions which a user is typically not interested in for searching.[1] A similar pattern to exclude certain fragments from an XML document for phrase search has been proposed in the PIX project [2]. Figure 6 gives an example. Figure 6a shows the original data with the formatting instructions. Figure 6b shows the *view* (without formatting

[1] Users who wish to, say, search for all documents that use a particular font would use a different view to query the data.

```
 <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)

Figure 6. Excluded Example

```
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 Annotation (Instance 1)

```
Mickey likes Minnie.
```

(c) Instance without the Annotation (Instance 2)

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

(d) Rule (R2)

Figure 7. Annotation Example

instructions) as it is relevant for most users. Figure 6c 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* attributes contains an XQuery expression that defines to which elements the rule should be applied.[2]

4.2 Annotations

Another basic pattern is Annotation. Examples for annotations are footnotes or comments in a text. Figure 7 gives an example. Figure 7a shows the original document. Figures 7b and 7c show two instances that could be of interest to the user. Figure 7d shows the rule that users can use in order to declare that they are interested to query the two instances separately. Again, the motivation for this pattern was provided by the PIX project [2]: Only the application of this pattern makes it possible to apply a phrase search for “Mickey likes Minnie”; the original document would not match. Since a user might also be interested in the text of the footnote, the Excluded pattern is not appropriate in this example. Both instances in Figure 7 are therefore part of the view.

Annotations are one 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 annotation rules that declare annotations in a document (e.g., in OpenOffice, separate rules would indicate that `note` and `comment` elements should be treated as annotations).

4.3 Alternatives

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 8 gives a simple example. The original document contains markup with `option` elements that indicates whether text fragments belong to the world of Mickey Mouse or Donald Duck. If the text is viewed from the perspective of the world of Mickey Mouse it should read “Mickey likes Minnie” (Figure 8b); otherwise it should read “Donald likes Daisy” (Figure 8c).

[2] 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.

```
<option world="mouse">Mickey</option>
<option world="duck">Donald</option>
likes
<option world="mouse">Minnie</option>
<option world="duck">Daisy</option>.
```

(a) Original Data

```
<option world="mouse">Mickey</option>
likes <option world="mouse">Minnie</option>.
```

(b) World of Mouse (Instance 1)

```
<option world="duck">Donald</option>
likes <option world="duck">Daisy</option>.
```

(c) World of Duck (Instance 2)

```
<alternative match="//option" key="$m/@world"
                              optional="false" />
```

(d) Rule (R3)

Figure 8. Alternative Example

The rule that declares which alternatives belong together for this example is given in Figure 8d. It contains a *key* attribute which contains an XQuery expression that specifies how the matching tags are correlated. 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. When applied to a set of different documents which share commonalities (elements with the same key), the Alternative rule acts as a correlating factor (`GROUP BY`) between documents. This allows search to be performed across groups of documents instead of single documents, as described in Section 8. A special case of the Alternative pattern is the *Annotation* pattern described in the PIX project [2]. It is given by an alternative with two options with one empty [11].

4.4 Versions

The *Version* pattern is useful for Example 2 of Section 2. It is also useful for Wiki data and for documents generated by Word and OpenOffice which allow the tracking of changes. The Alternative pattern *partitions* the elements that match an Alternative rule. In contrast, the Version pattern orders all matching elements and specifies that the view contains an instance for the each subsequence.

Figure 9 gives an example, again using the OpenOffice data format. Initially (before 3/16/2006), the document was “Mickey likes Minnie” (Figure 9b). Then, by a sequence of insertions, the document became “Mickey Mouse likes Minnie Mouse”. The rule shown in Figure 9e declares that each version of the document can be queried separately.[3] This way, it is possible to see the period in which a document contained a certain phrase. Furthermore, it is possible to query only the latest version of the document, thereby omitting keywords that were deleted from earlier versions.

4.5 Other Patterns

A few more patterns are defined in [11], namely, *Excluded, Annotation, Placeholder, Field,* and *Join*, which are useful in order to declare application views. For instance, the Field pattern is useful for implementing Example 1 of Section 2. This pattern allows to include content from other documents and associate the content to different groups as in the Alternative pattern. Note that several rules may be applied to the same input document (e.g., a document in which both an Alternative and a Version rule are specified). In that case, the instances on the application view correspond to all combinations of instances for each rule, resulting in potentially an exponential growth of the number of instances with the number of rules. As a result, potentially huge views can be defined. From a usability perspective, patterns are generic enough so that application can themselves create the rules for the types of data they generate.

5 Normalization

In this section, we describe the process of normalization (see Figure 5), which gets a document and a set of rules as input and generates a normalized document that represents all instances of the corresponding application view on that document. The challenge of normalization is to keep the normalized view as compact as possible, which is done by *factoring out* the

[3] Rules for deletion work in an analogous way and can be combined with rules for insertion. Furhtermore, it is possible to correlate rules for deletion and insertion so that the exact history of a document can be specified [11].

```
<inserted id="1"> <info date="3/16/2006"/> </inserted>
<inserted id="2"> <info date="3/28/2006"/> </inserted>
Mickey <insert id="1">Mouse</insert> likes
Minnie <insert id="2">Mouse</insert>.
```

(a) Versioned OpenOffice Document (Original Data)

```
<inserted id="1"> <info date="3/16/2006"/> </inserted>
<inserted id="2"> <info date="3/28/2006"/> </inserted>
Mickey likes Minnie.
```

(b) "Mickey likes Minnie" (Instance 1)

```
<inserted id="1"> <info date="3/16/2006"/> </inserted>
<inserted id="2"> <info date="3/28/2006"/> </inserted>
Mickey <insert id="1">Mouse</insert> likes Minnie.
```

(c) "Mickey Mouse likes Minnie" (Instance 2)

```
<inserted id="1"> <info date="3/16/2006"/> </inserted>
<inserted id="2"> <info date="3/28/2006"/> </inserted>
Mickey <insert id="1">Mouse</insert> likes
Minnie <insert id="2">Mouse</insert>.
```

(d) "Mickey Mouse likes Minnie Mouse" (Instance 3)

```
<version match="//insert" action="AFTER_AT"
        key="//inserted[@id=$m/@id]/info/@date"/>
```

(e) Rule (R4)

Figure 9. Version Example

common parts of a document which are not affected by the rules. Such a compact representation exists for all the patterns described in Section 4 and in more details in [11].

5.1 Examples

Normalization can be best described using an example. Figure 10 shows the normalized view for the example of Figure 8. This normalized view encodes the two instances "Mickey likes Minnie" (Figure 8b) and "Donald likes Daisy" (Figure 8c). 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.

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

Figure 10. Normalized View - Alternatives

As shown in Figure 10, each `select` element contains a predicate that specifies in which kind of instances of the view its content should be considered. 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 be included into the instance that represents Donald Duck's view of the world. Note that *R3* is a variable with a generic name that is generated for the Alternative rule of Figure 8d. *mouse* and *duck* are the values of the evaluation of the *key* expression of the rule of Figure 8d. The predicate specifies an equality because the rule of Figure 8d is an Alternative rule, and equality corresponds to the semantics of the Alternative pattern.

Another example of a normalized view is given in Figure 11. This normalized view encodes all the instances of the view described in Figure 9. Again, `select` elements define the variable parts of the document and predicates specify the inclusion of such variable content in instances. Since this example is based on a Version rule with an `AFTER_AT` action (Figure 9e), the predicates are "$\geq$" predicates (rather than equalities for the Alternative pattern). *R4* is an identifier generated for the Version rule of Figure 9e; the dates in the predicates are computed from the *key* expression of that rule.

```
<inserted id="1"><info date="3/16/2006"/></inserted>
<inserted id="2"><info date="3/28/2006"/></inserted>
Mickey
<select pred="R4 >= 3/16/2006">
  <insert id="1">Mouse</insert>
</select>
likes Minnie
<select pred="R4 >= 3/28/2006">
  <insert id="2">Mouse</insert>
</select>.
```

Figure 11. Normalized View - Versions

5.2 Normalization Algorithm

The algorithm to compute a normalized view from a document and a set of rules operates in two phases. They are described in the procedures below.

Phase I - ConstructTaggingTable(d, R): constructs a *Tagging Table* T from the original document d and the set of rules R. The Tagging Table specifies which elements of the original document need to be tagged with a `select` element and its predicate. We employ the following steps:

1. For each node $n \in d$ returned by the *match* expression of a rule $r \in R$, insert a tuple t in T and make $t.RULE \leftarrow r$, $t.PATTERN \leftarrow r.PATTERN$, and $t.NODEID \leftarrow n$. The Tagging Table of Figure 12 is constructed for the example of Figure 10. Every tuple in that Tagging Table contains the id of a matched node (e.g., 1 is the NodeId for the `<option world="mouse"> Mickey </option>` element in Figure 8a) and the identifier of the rule that matches that node.
2. Evaluate the *key* expression of r on n and set $t.KEYVALUE$ to the value of the *key* expression and $t.OP$ to the operator implied by rule r. For instance, in Figure 12, for node id 1, we set the key value to `"mouse"` and, as rule R3 is an Alternative, the operator to equality.
3. If a rule $r' \in R$ matches a node n previously matched by a rule $r \in R$, then a *conflict* occurs. As mentioned in [11], one possible way to deal with conflicts is to return an error during normalization. For the purpose of this work, we follow exactly these semantics, as they are compliant with the snapshot semantics which are currently under discussion for the XML Update language of the W3C [9].

For completeness, we show on Figure 13 the Tagging Table for the example of Figure 11. The two examples above showed how tuples of the Tagging Table are generated for the Alternative and Version patterns. The same technique can be applied to any pattern described in Section 4 and [11].

Rule	Pattern	NodeId	KeyValue	Op
R3	Alternative	1	mouse	=
R3	Alternative	2	duck	=
R3	Alternative	3	mouse	=
R3	Alternative	4	duck	=

Figure 12. Tagging Table - Alternatives

Rule	Pattern	NodeId	KeyValue	Op
R4	Version	5	3/16/2006	$\geq$
R4	Version	6	3/28/2006	$\geq$

Figure 13. Tagging Table - Versions

Phase II - ConstructNormalizedView(d, T): constructs the *normalized view* V by scanning through each tuple t of Tagging Table T and executing the following transformations on document d, depending on $t.PATTERN$:

1. If $t.PATTERN$ is either Alternative or Version, then a `select` element is generated that embraces the node of d which is specified by $t.NODEID$. The `pred` attribute of this `select` element is set to “$t.RULE\ t.OP\ t.KEYVALUE$” (e.g., R4 $\geq$ 3/16/2006 from Figure 13).
2. If the rule specifies an Excluded pattern, then the matching node is simply not included in the normalized view document. (No `select` element is generated in this case.)
3. If the rule specifies an Annotation pattern, then two `select` elements are generated. The first `select` element contains the matching node of the original document; the second `select` element is empty. The `pred` attribute of the first `select` element is set to “Rule = INCLUDE” (e.g., “R2 = INCLUDE”, if R2 is the rule identifier); the `pred` attribute of

the second `select` element is set to “Rule = EXCLUDE” (e.g., “R2 = EXCLUDE”, if R2 is the rule identifier). Figure 14 shows the normalized view for the Example of Figure 7.

4. If $t.PATTERN$ is any of the other patterns given in [11], then we address them accordingly in a straightforward way.

```
Mickey
<select pred="R2 = INCLUDE">
  <note id="ftn0">He is a Disney Character.</note>
</select>
<select pred="R2 = EXCLUDE"/>
likes Minnie.
```

Figure 14. Normalized View - Annnotation

Both the construction of the Tagging Table and the construction of the normalized view can be implemented quite easily using XSLT or XQuery. For instance, the implementation used for the performance experiments (Section 8) was based on Microsoft’s XSLT processor which is integrated into the .NET framework.

5.3 Benefits of Normalization

The normalized view, as described in this section, brings the benefits described below.

Completeness. The normalized view encodes all the instances in 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. It is important to have such a generic, rule-independent representation of the view because the rule language must be extensible as new patterns might become important for new kinds of data. Having such a rule-independent representation of data makes it possible to extend the rule language without adjusting the indexing and query processing components.

Compactness. The normalized view is a compact representation of the original data. If materialized in an unnormalized way, the space requirements for storing the application view grow exponentially with the number of rules. In contrast, the size of the normalized view is always in the same order as the size of the original document: at most two additional `select` elements are generated for each element of the original document. In addition, the normalized view can be compressed in order to reduce its size. For example, dictionary-based compression can be used in order to represent the two selections *mouse* and *duck* by a single Bit in the example of Figure 10. In our implementation, we applied such compression techniques; for obvious reasons, all examples in this paper are given in an uncompressed way. 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 [13]. Again, such an implementation reduces the size of the normalized view and significantly speeds up query processing because no parsing is required.

Instance Computation. The key idea of normalization is to tag variable parts of a document, and to give predicates that specify which instances of the view involve which parts of the document. As a consequence, specific instances of the normalized view can be retrieved by instantiation of the variables of these predicates. The “Mickey likes Minnie” instance of the normalized view of Figure 10, for example, can be retrieved by instantiating the *R3* variable to “mouse”. Analogously, the “Mickey Mouse likes Minnie Mouse” instance of the normalized view of Figure 11 can be retrieved by instantiating the *R4* variable to, say, “3/31/2006”. This way of referencing individual instances is exploited for indexing, which is described in the next section.

6 Enhanced Indexing

In this section, we describe how to extend a conventional inverted file to provide a fine-grained representation for individual instances. A naïve approach towards fine-grained indexing is to create index entries for each instance of the application view separately and to include all these entries into the inverted file. However, this approach is equivalent to naïve materialization of the application view and is clearly not viable. The key idea to achieve both fine-grained indexing and tolerable index sizes (and thus good query performance) is, again, to factor out common parts, index those only once, and to index only the variable parts individually.

6.1 Example

Figure 15 shows a logical representation of an extended inverted file for the normalized view of Figure 11. In addition to the document id (d_1 is used in this example to refer to the original data of Figure 9a), keyword, and score (plus possibly other properties such as position information for phrase search), 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.

DocId	**Keyword**	**Score**	**Predicate**
d_1	likes	1	true
d_1	Mickey	1	true
d_1	Minnie	1	true
d_1	Mouse	1	$3/16/2006 \leq R2 < 3/28/2006$
d_1	Mouse	2	$R2 \geq 3/28/2006$

Figure 15. Inverted File - Version

The keywords *likes*, *Mickey*, and *Minnie* appear in all instances. As a result, the predicate is *true* for these keywords, indicating that there is no restriction on the instances. The keyword *Mouse*, however, only occurs in certain instances and occurs more often in the latest version than in an earlier version. As a result, two entries are made for this keyword, each one specifying the corresponding instance and the score (i.e., count in this example) for that keyword. Of course, this example only serves for illustration. Typically, inverted files index many documents. Furthermore, inverted files are usually partitioned into inverted lists (one per keyword) in order to speed up query processing.

6.2 Index Creation Algorithm

In order to create an inverted file for a set of normalized views, each normalized view is processed separately (as in traditional information retrieval). The order in which the normalized views are processed is irrelevant (also as in traditional information retrieval). The processing of a normalized view V is carried out in the following two phases.

Phase I - CreateIndexEntries(V): creates an entry e in the inverted file for each occurrence of each keyword k (possibly disregarding stop words and applying stemming as in traditional IR). The predicate for e is set in the following way:

1. If the keyword is not inside a `select` element in V, then the predicate of e is simply *true*. Examples are *likes*, *Mickey*, and *Minnie* in Figure 15.
2. If the keyword is inside a `select` element in V, then e inherits the predicate of this `select` element. Examples are the two occurrences of *Mouse* in Figure 15.
3. If the keyword is inside nested `select` elements (not shown in our example), then the predicate of e is the conjunction of the predicates of the `select` elements of the normalized view. Details on this situation are provided in [16].

Initially the score of all entries is 1; it is adjusted when the entries are merged.

Phase II - MergeIndexEntries(I): For an index I obtained as a result of Phase I, it is possible that several such entries involve the same keyword (or stem) and have equivalent or overlapping predicates. Entries are merged using the following strategies:

1. If two entries for the same keyword have equivalent predicates, then these two entries can be merged into one entry; the score of the merged entry is adjusted accordingly, using the search engines specific scoring functions (for simplicity, this paper uses the term frequency as score, but other metrics could also be used [6]).
2. If the predicates of two entries, say p_1 and p_2 overlap, then three entries are generated: one entry for $p_1 \wedge p_2$, one entry for $p_1 \wedge \neg p_2$, and one entry for $\neg p_1 \wedge p_2$ (with scores assigned accordingly). If p_1 implies p_2, then $p_1 \wedge \neg p_2$ is always *false* and this entry can be discarded so that only two entries are generated. Furthermore, $p_1 \wedge p_2$ is equivalent to p_1 if p_1 implies p_2. For example, if p_1 = “R4 $\geq$ 3/28/2006” and p_2 = “R4 $\geq$ 3/16/2006”, then the two entries shown in the inverted file of Figure 15 for the keyword “Mouse” are generated. Since the predicates in the index entries are always conjunctions and negations of simple variable/value comparisons, the equivalence, implication, and overlap of predicates can easily be computed.

Merging index entries that contain positional information (e.g., for phrase search) is more challenging. We have devised an approach to do that along the lines of proposals for positional search in the IR research community (e.g., PIX [2]), but describing it is beyond the scope of this paper (see [16]).

List : *"bar"*		List 2: *"foo"*		Result: *"bar"* $\wedge$ *"foo"*
d_1, R1 = 1		d_1, $1 \leq$ R1 ≤ 3		d_1, R1 = 1
d_1, R1 = 2 $\wedge$ R2 > 3		d_1, R2 = 3		d_1, R1 = 1 $\wedge$ R2 = 3
d_1, R1 = 4	$\bigwedge$	$d_2, \ldots$	$=$	d_1, R1 = 2 $\wedge$ R2 > 3
$d_2, \ldots$				d_1, R1 = 4 $\wedge$ R2 = 3
				$d_2, \ldots$

Figure 16. Processing Conjunctions

6.3 Benefits of Enhanced Indexing

The enhanced index for application views, as described in this section, brings the benefits described below.

Compactness. If correct query results are to be produced, for each keyword in the inverted file, we must attach information about in which instances that keyword appears. The merging phase of index construction expands the set of instances covered by each individual predicate. Thus, the necessary number of entries in the index is minimized.

Ease of Implementation. An enhanced index extends a traditional inverted file by associating predicates to keyword entries. Therefore, existing search engine implementations may be easily extended to include in the generated index the additional information. As we will see in Section 7, query processing on the enhanced index may also be implemented with reasonable additional effort.

Fuzziness. The procedures described in the previous section build a space-efficient index over *all* instances of the input application views. One approach to reduce the number of entries in an enhanced index is to *fuzzify* the index; for example, some entries could simply be discarded. This fuzzification might result in false positives and false negatives (such as those described in Section 2), thereby trading index accuracy for index size. Since the performance experiments reported in Section 8 show that even a precise index scales well and has a tolerable size, we did not study this fuzzification in more detail. It is, however, an interesting avenue for future research.

7 Query Processing

As in traditional information retrieval, this work focuses on queries that are composed of keywords. Also, these queries can be processed using an extended inverted file, in almost the same way as in any traditional information retrieval system [6]. As an example, a user might be interested in all documents (i.e., instances) that contain the keywords *Mickey* and *Mouse*. First, the (extended) inverted file is queried in order to find all entries for the keyword *Mickey*. Using the inverted file of Figure 15, the result is $\langle d_1, 1, \textit{true} \rangle$ (the keyword can be omitted at this point). Likewise, the inverted file is queried in order to find all entries for the keyword *Mouse*. The result is $\langle d_1, 1, 3/16/2006 \leq R2 < 3/28/2006 \rangle, \langle d_1, 2, R2 \geq 3/28/2006 \rangle$. In the next step, these two lists are merged. In traditional information retrieval, the merge for conjunctive queries implements a join (or interesection), thereby trying to find matching documents from both lists. For the extended inverted files (e.g., Figure 15), this logic must be extended in order to process the predicates that specify the relevant instances. Therefore, in addition to a join on the document identifiers, a conjunction of the predicates must be carried out. As a result, the following list is constructed for the query *Mickey Mouse*:

$$\begin{array}{l} \langle d_1, 1, 3/16/2006 \leq R2 < 3/28/2006 \rangle \\ \langle d_1, 2, R2 \geq 3/28/2006 \rangle \end{array}$$

From this list and the normalized document, result construction can generate the result instances and sort the results according to their score. Different scores are possibly computed depending on the information retrieval system that is used.

Figure 16 shows a more complex example of how two lists are merged for a conjunctive query (keywords and scores are ignored in this example). This example involves two rules so that the predicates involve two variables (R1 and R2). For each document, the same set of variables is used in the predicates of all inverted lists because the same rules were applied to the document. In Figure 16, for example, instances of Document d_1 are specified by constraining the values of Variables R1 and R2. The first inverted list (List 1) has three entries for Document d_1; the second list (List 2) has two entries for d_1. Logically, all six combinations of entries of the first list and entries of the second list are considered. Some of these combinations result in *false*; e.g., R1=2 $\wedge$ R2 > 3 $\wedge$ R2 = 3 is *false*. In an inverted list, a predicate with *false* indicates the empty set of instances, so such entries can be evicted. As a result, only four entries are part of the result of this merge operation.

In order to optimize the implementation for complex predicates that involve many variables, conjunctions, and disjunctions, a disjunctive normal form is maintained. That is, predicates in the inverted file and in intermediate results of query processing are always conjunctions. Disjunctions are modelled using several entries. For example, the two rows for *Mouse* in the inverted file of Figure 15 can be interpreted as a disjunction specifying that d_1 matches *Mouse* for all instances that fulfill $3/16/2006 \leq R2 < 3/28/2006$ *or* $R2 \geq 3/28/2006$.

Disjunctive queries are processed in an analogous way. Essentially, the join (on document identifier) is computed and, in addition, a logical *or* is carried out on the predicates of the entries. Of course, queries can contain many keywords combined by several conjunctions and disjunctions. In the prototype system developed for this work and used for the performance experiments (Section 8), the inverted lists of all these keywords are merged pairwise in the order in which they are specified in the query, i.e., a left-deep merge tree is created. Obviously, more clever, cost-based optimization techniques can be applied, thereby reordering the sequence in which the inverted lists are merged. Studying such optimization techniques is beyond the scope of this paper; essentially, the same techniques are applicable as in traditional information retrieval systems.

In all experiments that we conducted so far, this algorithm was efficient enough to meet the performance requirements of modern information retrieval (Section 8). There are, however, several additional optimizations that can be applied. First, if only points (rather than intervals) are applied, 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., [12]) can be applied. We plan to study the trade-offs of the different join techniques as part of future work.

8 Experiments

We implemented the techniques presented in the previous sections (normalization, indexing, and extended query processing) and experimentally compared our implementation with **Windows Desktop Search (WDS)**, which provided the best customization possibilities. We call our approach “Enhanced” (with rules) and the baseline inverted index “Traditional” (no rules apply). The goal of the experiments was to show that (a) our extended approach gives better search results than traditional search and (b) the overheads in terms of index size and query processing times are tolerable with rules applied.

Experimental Environment. The indexing engine (i.e., 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 1.7GHz processor and 1GB of RAM under Windows XP Professional.

Experimental Data. For the experiments reported in this paper, we crawled data collections which exhibit the patterns in Table 1. Data sizes appear in Table 3.

Queries on each collection return False Positives or False Negatives, depending on whether traditional search sees the instances correctly or not: e.g., searching versions (Twiki,CVS) produces False Positives, if no version contained the searched keyword, but the document as a whole does. E-mails were grouped by conversation (Alternative pattern): this causes False Negatives since a single E-mail message may not be returned although its conversation contains all keywords.

Data Collection	**Patterns**	**Trad.Search Produces False Positives**	**Trad.Search Produces False Negatives**
Twiki	*Version*	YES	
CVS	*Version*	YES	
LaTeX	*Annotation* *Placeholder*		YES YES
E-mail	*Grouping* by conversation thread		YES
Java code	*Placeholder* (for Inherited code)		YES

Table 1. Problems with the Accuracy of State-of-the-art Desktop Search

8.1 Improvement of Precision and Recall

The goal of our enhanced approach was to improve accuracy of current desktop search technology. Based on the queries run on the data, we computed Precision and Recall of the Traditional approach, relatively to the Enhanced approach (desired). The improvement in quality is reflected in Table 2 and proves the usefulness of the extended desktop search.

	Precision Improvement	**Recall Improvement**
Twiki	+34 %	0%
CVS	+4%	0%
LaTeX	0%	+17%
E-mail	0%	+37%
Java code	0%	+52%

Table 2. Improvement of Precision and Recall

8.2 Index Size and Index Creation Time

As shown in Table 3, 4 and 5, we created indexes for the data set and measured the index size and index creation time using the enhanced approach (which uses rules) as opposed to the traditional approach (no rules, but with imprecise query results) and to the naïve approach (materialized instances). First, Table 3 shows that the size of the normalized view is comparable to that of the original data, shown as a baseline. For each data set, Tables 4 and Tables 5 show that it is more efficient in time and space to normalize and index, than to materialize all instances and then index them. Index creation time is higher than for the traditional approach, but it has lower impact, since it is performed offline. E-Mail is special since there is no overlapping between instances and we do not gain in terms of space.

Collection	Original Size (MB)	Normalized View (MB)	Materialized Instances (MB)	Space Gain(%) through Normaliz.
Twiki	6.10	6.68	54.8	78%
CVS	5.55	5.75	16.5	66%
LaTeX	13.2	19.4	25.5	92.4%
E-mail	45.9	46.4	46.2	0%
Java Code	182	200	307.77	35%

Table 3. Space Gain through Normalization

Collection	Trad Index	Enh Index	Naïve Index	Overhead Enh/Trad	Space Gain Enh/Naïve	Ix.Size WDS
Twiki	3.3	8.03	13.3	x2.43	40%	54.4
CVS	2.75	3.73	6.91	x1.36	46%	21.2
LaTeX	6.82	10.4	13	x1.52	20%	23.8
E-mail	37.8	48.4	48.4	x1.3	0%	53.6
Java Code	22.9	28	32.1	x1.22	14%	105

Table 4. Index Size(MB)

Collection	Trad	Enh. +Normaliz.	Naïve +Inst. materializ.	Overhead Enh/Trad	Gain Enh/Naïve
Twiki	22	27.5	105.06	x1.25	74%
CVS	10	17	52.9	x1.7	68%
LaTeX	32.7	54.56	73.595	x1.67	26%
E-mail	37.8	48.4	48.4	x1.3	0%
Java Code	260	383	520.61	x1.47	26%

Table 5. Index Creation Time(seconds)

8.3 Query Processing Time

Table 6 presents the average running times (in milliseconds) of keyword queries. Queries were chosen to be meaningful with respect to the data collection, and contain from two to five keywords. It is obvious that the running times of an enhanced search are longer than the running times of traditional IR since the indexes that need to be scanned are larger and the logic that merges two inverted lists is more complex (Section 7).

The differences in running times depend on the data set, on the rules applied, and more importantly, on the type of query. The more variable parts of the data are affected by rules, the higher the overhead of an enhanced search gets. In those cases, however, the increased running time is subset by the increased precision of the results returned by an enhanced search. Query processing times are, however, very acceptable for desktop search, and remain below 1s.

Collection	Traditional Search	Enhanced Search	Overhead Enh/Trad	WDS
Twiki	11.91	17.5	x1.47	14.4
CVS	6.22	13.64	x2.20	7.8
LaTeX	5.48	20.47	x4.29	16.76
E-mail	9.13	9.63	x1.06	33.18
Java Code	11.89	16.73	x1.41	17.2

Table 6. Query Processing Time(ms)

9 Related Work

This work was inspired by several other research projects. Semantic Web [20, 19]aims to add meta-data for describing relationships between documents and uses an advanced query language that supports reasoning. This work is different in that it aims to annotate documents for specifying their semantics, and uses simple keyword search.

We are more related to other approaches in the database and IR community: Colorful XML was proposed in order to give an XML document several interpretations [15]. We differ in goal (IR, keyword search) and approach (Normalization, Enhanced Indexing). Some patterns (e.g., Excluded) were inspired by the PIX project at AT&T [2]. We generalize PIX by allowing more patterns and not focusing on just one, such as in [4]. Similarly to PIX, we also developed *phrase search* by combining positioning information with predicates - not present this here for brevity. It adds an overhead to the approach, but it is still affordable for desktop computing. The problem of duplicates and common content was also defined by [7], but this framework generalizes their idea.

There has been a lot of other work on various aspects of XML query processing, XML keyword search, and XML information retrieval [14, 21]. Query relaxation has been proposed in [8, 18, 3]. The combination of structured and keyword queries has been studied specifically in [17]. Furthermore, the emerging XQuery FullText standard [1] is relevant. Again, all this work is orthogonal and can be applied in the *Indexing* and *Searching* phase of Figure 5.

10 Conclusions

This work was motivated by the observation that current desktop search engines look at data differently than applications. As a result, a state-of-the-art search engine might return a document although it is not relevant for a query and, similarly, it might miss relevant documents.

The key idea of this work is to extend current search engine technology to index and search application views. An enhanced search engine uses rules to define those views. A normalized representation for each application view is derived from these rules and the original data. The normalized view encodes the whole application view in a compact and generic way. The main contribution of this work is to define such normalized views, give an algorithm to generate them, show how extended inverted files can be constructed on normalized views, and define the extensions that need to be carried out to the *merge* algorithm of a keyword search query processor. The performance results are encouraging because they demonstrate that the overheads of such an enhanced search engine are 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 Section 9. 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.

References

[1] S. Amer-Yahia, C. Botev, S. Buxon, P. Case, J. Doerre, D.McBeath, M.Rys, and J.Shanmugasundaram. XQuery 1.0 and XPath 2.0 Full-Text, W3C Working Draft, 4 April 2005. http://www.w3.org/TR/2005/WD-xquery-full-text-20050404/.

[2] S. Amer-Yahia, M. F. Fernández, D. Srivastava, and Y. Xu. PIX: A System for Phrase Matching in XML Documents. In *ICDE*, 2003.

[3] S. Amer-Yahia, L. V. S. Lakshmanan, and S. Pandit. FleXPath: Flexible Structure and Full-Text Querying for XML. In *SIGMOD*, 2004.

[4] P. G. Anick and R. A. Flynn. Versioning a full-text information retrieval system. In *SIGIR*, 1992.

[5] M. Arenas and L. Libkin. A normal form for XML documents. *TODS*, 29:195–232, 2004.

[6] R. A. Baeza-Yates and B. A. Ribeiro-Neto. *Modern Information Retrieval*. ACM Press / Addison-Wesley, 1999.

[7] A. Z. Broder, N. Eiron, M. Fontoura, M. Herscovici, R. Lempel, J. McPherson, R. Qi, and E. J. Shekita. Indexing Shared Content in Information Retrieval Systems. In *EDBT*, 2006.

[8] D. Carmel, Y. S. Maarek, M. Mandelbrod, Y. Mass, and A. Soffer. Searching XML documents via XML fragments. In *SIGIR*, 2003.

[9] D. Chamberlin, D. Florescu, and J. Robie. XQuery Update Facility W3C Working Draft 27 January 2006 . http://www.w3.org/TR/xqupdate/.

[10] E. F. Codd. Further Normalization of the Data Base Relational Model. *IBM Research Report*, RJ909.

[11] J.-P. Dittrich, C. Duda, B. Jarisch, D. Kossmann, and M. A. V. Salles. A Rule Language for Defining Views on Application Data. https://www.dbis.ethz.ch/research/publications/appdatarulelang.pdf.

[12] J.-P. Dittrich and B. Seeger. GESS: a Scalable Similarity-Join Algorithm for Mining Large Data Sets in High Dimensional Spaces. In *KDD*, 2001.

[13] D. Florescu, C. Hillery, D. Kossmann, P. Lucas, F. Riccardi, T. Westmann, M. J. Carey, A. Sundararajan, and G. Agrawal. The BEA/XQRL Streaming XQuery Processor. In *VLDB*, 2003.

[14] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram. XRANK: Ranked Keyword Search over XML Documents. In *SIGMOD*, 2003.

[15] H. V. Jagadish, L. V. S. Lakshmanan, M. Scannapieco, D. Srivastava, and N. Wiwatwattana. Colorful XML: One Hierarchy Isn't Enough. In *SIGMOD*, 2004.

[16] B. Jarisch. How to Teach Semantics to a Search Engine. Master's thesis, ETH Zurich, Switzerland, Mar. 2006.

[17] R. Kaushik, R. Krishnamurthy, J. F. Naughton, and R. Ramakrishnan. On the Integration of Structure Indexes and Inverted Lists. In *ICDE*, 2004.

[18] Y. Li, C. Yu, and H. V. Jagadish. Schema-Free XQuery. In *VLDB*, 2004.

[19] D. L. McGuinness and F. van Harmelen. OWL Web Ontology Language Overview. http://www.w3.org/TR/owl-features/.

[20] RDF, Resource Description Framework. http://www.w3.org/RDF/.

[21] M. Theobald, R. Schenkel, and G. Weikum. An Efficient and Versatile Query Engine for TopX Search. In *VLDB*, 2005.