Bringing Precision to Desktop Search: A Predicate-based Desktop Search Architecture

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

Recently, derivatives of Google and MSN Search have become popular in order to search for documents on the desktop of a PC. However, here is the problem that motivates this work: 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 encoded into the data or into the views provided by the applications - a situation shown in Figure 1. As a result, search engines potentially return documents that are not relevant or miss documents even though they are relevant.

This work defines an architecture which extends 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

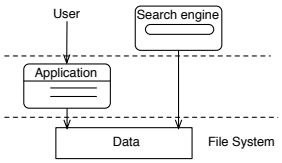

Figure 1. User View vs. Search Engine View

data. We experimentally show improvement in accuracy, with only reasonable performance overhead.

Motivating Example: Bulk Letters. Figure 2 shows a snippet of a letter as Word document (in simplified XML format), and a spreadsheet containing names. 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

From this logical perspective, 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. Another example is Versions: which generate several instances with many commonalities.

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. 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. A similar example is the Version pattern, described in Section 3. □

2 Overview of Our Solution

In order to solve the problems stated above, we propose to extend desktop search engines as shown in Figure 4. Crawling and querying of the data is performed in the steps described below.

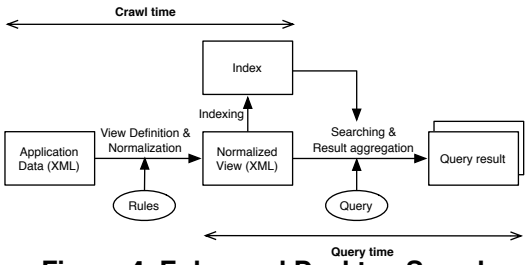

Figure 4. Enhanced Desktop Search

Normalization. Since the application views may be very large, they are never materialized. Instead, these views are directly normalized. The normalization is described in Section 4 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 7) 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 5. Again, these extended inverted files are moderately larger than inverted files for traditional information retrieval (Section 7).

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 6.

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 which makes it easier to define views. In practice, this assumption can easily be met because almost all applications provide an XML interface.

3 View Definitions. Patterns in 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. The exact definition of the rule language is given in [4].

Alternatives. The *Alternative* pattern specifies that one out of several options of a text is chosen. For example it is useful to specify that a Web page is available 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 5 gives a simple example. The original document contains markup with `option` elements that indicates the world to which the text fragments belong to. If the text is viewed from the perspective of the world of Mickey Mouse it should read "Mickey likes Minnie" (Figure 5b); otherwise it should read "Donald likes Daisy" (Figure 5c).

```
<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)

Figure 5. Alternative Example

The number of instances in a view specified using this pattern grows linearly with the number of worlds specified in the document. The Alternative pattern may also applied to different *files* with commonalities, and acts as a correlating factor (`GROUP BY`) (as shown in Section 7).

Versions. The *Version* pattern is useful for example for for Wiki data and for office documents which allow the tracking of changes. The Version pattern orders all matching elements on a time scale and specifies that the view contains an instance for each subsequence of moments.

Figure 6 gives an example using the OpenOffice data format. The initial document, "Mickey likes Minnie" (Figure 6b) becomes "Mickey Mouse likes Minnie Mouse" by a sequence of insertions. If each version of the document can be queried separately, it is possible to see only the period in which a document contained a certain phrase.

Other Patterns. A few more patterns are defined in [4], namely, *Excluded, Annotation, Placeholder, Field*, and

```
<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)

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

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

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

Figure 6. Version Example

Join, which are useful in order to declare application views. If more patterns apply, the the instances on the application view correspond to all combinations of instances for each pattern. This is a reason for which materialization should be avoided.

4 Normalization

The challenge of normalization is to keep the application view as compact as possible, which is done by *factoring out* the common parts of a document which are not affected by the rules. Such a compact representation exists for all the patterns described in Section 3.

Normalization Examples. Figure 7 shows the normalized view for the example of Figure 5. This normalized view encodes the two instances "Mickey likes Minnie" (Figure 5b) and "Donald likes Daisy" (Figure 5c). 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 7. Normalized View - Alternatives

As shown in Figure 7, `select` elements contains predicates that specify in which 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 only be included into the instance that represents Mickey Mouse's perspective on the original data. *R3* is a variable with a generic name that is generated for this pattern occurence.

Another example of a normalized view is given in Figure 8. This normalized view encodes all the instances of the view described in Figure 6. Again, `select` elements define the variable parts of the document and predicates specify the inclusion of such variable content in instances. In case of versions, the predicates are "$\geq$" predicates (rather than equalities for the Alternative pattern). Normalization brings an important space gain in this case (in contrast to materializing all versions) since overlappings are stored just once (Section 7).

```
<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 8. Normalized View - Versions

4.1 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 very big) application view. This way, the normalized view can serve as a basis for indexing and all further query processing.

Compactness. 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 (e.g., dictionary-based compression for world values) as we did in our implementation.

Instance Computation. The key idea of normalization is to tag variable parts of a document with predicates. 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 7, for example, can be retrieved by instantiating the *R3* variable to "mouse".

5 Enhanced Indexing

The key idea to achieve both fine-grained indexing (level of instances) and tolerable index sizes (and thus good query performance) is to index the normalized view instead of individual instances (i.e., index common parts once, and index only the variable parts individually). A naïve approach of creating all entries for each instance is clearly not viable.

Indexing Example. Figure 9 shows a logical representation of an extended inverted file for the normalized view of Fig-

ure 8. Our extension is simple: the enhanced inverted file keeps a predicate that logically specifies in which instances the keyword appears.

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 9. Inverted File - Version

The keywords *likes, Mickey*, and *Minnie* appear in all instances, so the predicate is *true* for these keywords, indicating that there is no restriction on the instances.The keyword *Mouse* only occurs in certain instances and occurs more often in the latest version than in the earlier one. As a result, two entries are made for this keyword, specifying the corresponding instance and the score (e.g., count for each instance in this example).

5.1 Benefits of Enhanced Indexing

The enhanced index for application views brings the following benefits:

Compactness. We showed that we must attach predicates for each keyword in the inverted file. However, by merging predicates, we maximize the number of instances expressed by each index entry, and minimize the total index size.

Ease of Implementation. Existing search engine implementations may be easily extended to include in the generated index the additional predicate information. Section 6 shows that query processing on the enhanced index may also be implemented with just a minimal additional effort.

Fuzziness. The nature of predicates might allow us to fuzzify entries (i.e., discard some exact entries and include just approximate predicates in a compact entry. This might result in false positives and false negatives. However, we did not investigate this for now.

6 Query Processing

Similar to a traditional information retrieval system, enhanced query processing is a merge between the inverted lists of keywords. In addition, predicates should also be merged in the process. For example, the results to the query: *Mickey Mouse* are obtained after merging the lists $\langle d_1, 1, 3/16/2006 \leq R2 < 3/28/2006\rangle$, $\langle d_1, 2, R2 \geq 3/28/2006\rangle$ and $\langle d_1, 1, \textit{true}\rangle$. The results is:

$\langle d_1, 1, 3/16/2006 \leq R2 < 3/28/2006\rangle$
$\langle d_1, 2, R2 \geq 3/28/2006\rangle$

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

7 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

7.1 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.

7.2 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

	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

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)

7.3 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 6).

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)

8 Related Work

This work was inspired by several other research projects. First, we are related to Colorful XML [5], which was proposed in order to give an XML document several interpretations. 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 [1]. We generalize PIX by allowing more patterns and not focusing on just one, such as in [2]. The problem of duplicates and common content was also defined by [3]. Our framework generalizes their idea.

9 Conclusions

The key idea of this work is to extend current search engine technology to index and search application views. The main contribution was to define such views, normalize them, construct extended inverted files and define the extensions to the query processor for keyword search. The performance results are encouraging because better results are produced with tolerable performance overhead. As future work we focus on applying other XML information retrieval approaches, such as query relaxation and structure-based search. Currently, we 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, M. F. Fernández, D. Srivastava, and Y. Xu. PIX: A System for Phrase Matching in XML Documents. In *ICDE*, 2003.

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

[3] 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.

[4] 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/appdatarule-lang.pdf.

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