Konrad-Zuse-Zentrum
für Informationstechnik Berlin

Takustraße 7
D-14195 Berlin-Dahlem
Germany

FELIX HUPFELD

Hierarchical Structures in Attribute-based Namespaces and their Application to Browsing

ZIB-Report 03-06 (March 2003)

Hierarchical Structures in Attribute-based Namespaces and their Application to Browsing

Felix Hupfeld
Zuse Institute Berlin
Takustraße 7, 14195 Berlin,
Germany
hupfeld@zib.de

ABSTRACT

While attribute-value pairs are a popular method to name objects, information retrieval from those attribute-based namespaces is not an easy task. The user has to recall correct attribute names and values and master the syntax and semantics of query formulation. This paper describes hierarchical structures in attribute-based namespaces, shows how to extract them efficiently and evaluates the quality of these structures in an user experiment. It proposes an user interface for browsing attribute-named object sets which makes this task resemble today's file-system browsers and compares the usability of this interface to normal form-based methods in an user study.

TOPIC AREAS

Information Retrieval, Knowledge Management, User Interfaces, Attribute-based Naming, Attribute-named Data

1. INTRODUCTION

Attribute-value pairs are a popular method for naming objects and representing meta-data. They are used in digital library catalogues, for describing properties of objects in component systems and of network services ("yellow pages"), for tagging images in image libraries, for metadata in file systems and for many other applications.

In most cases, retrieval of information from attribute-named data sets is done via form-based methods. As seen for example with many library online catalogues, an on-screen form provides a template where the user can fill in attribute names and values he wants to restrict his query to.

However, the retrieval of attribute-based data is not an easy task. It requires the user to be familiar with the syntax and semantic of query formulation. To be able to query the system, the user has to recall correct attribute names and values or choose them from a long list of alternatives and know how to formulate constraints and how to combine them using boolean expressions.

ZIB Report 03-06

If the user is not familiar with the attribute namespace, a form-based query interface does not allow him to just browse the namespace to find the desired information. This is due to the lack of an inherent structure of the attribute space, which means that there is no natural way of browsing an attribute-named data set by following some given structure. Browsing attribute-named data with a form interface means iteratively formulating queries and modifying them while looking at their results.

The query of attribute-based namespaces would be simplified if the query interface could propose relevant query extensions in each retrieval step, so that the user does not have to recall valid attribute names and values or pick them from a huge selection of mostly irrelevant attributes available in the system. If these query extension proposals additionally had certain relationships between each other and the current query (which we will describe later), they could be used for browsing the attribute-named object space hierarchically.

In this paper, we will show how to extract attribute-value pairs from a given object namespace which summarize parts of the object set. These attribute-value pairs may be used to aid the user in query formulation and object naming by reducing the choice of possible query extensions and naming possibilities. Additionally, they introduce a structure in the object set which allows the object set to be browsed hierarchically. We will evaluate these attribute-value pairs in an user experiment in order to find out whether they provide a reasonable abstraction from the namespace's structures. Furthermore, we will present an user interface which uses this hierarchical structure to provide the user with an interaction that strongly resembles the file system browsers of today's operating systems. An usability test in the context of file-system browsing shows that it is to be preferred to form-based interfaces.

2. STRUCTURES IN ATTRIBUTE-BASED NAMESPACES

2.1 Basic Definitions

We will start with some basic definitions to introduce our notation.

Def. An *attribute A* is a tuple (name, value).

Def. An *object* is a named set of attributes.

Def. A *query predicate QP* is a tuple (name, operator, value), where name is an attribute name, operator an operator appropriate for the value's type, and value is an attribute value.

Def. The *induced set of the query predicate QP*, *o(QP)* is the set of objects which satisfy the constraint given by the query predicate.

Def. A *query* is a set of query predicates. It may be empty.

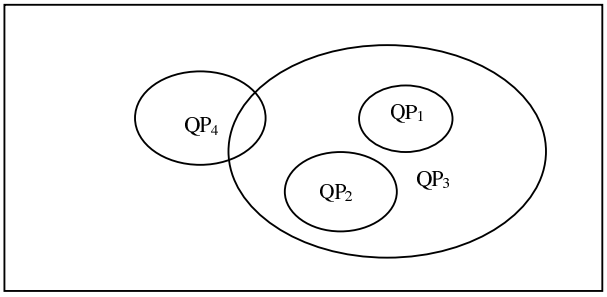
Def. The *induced set of a query Q*, *o(Q)* is the intersection of the o(QP) for all query predicates QP ∈ Q. o(∅) is the object set itself.

We assume a recursive retrieval process, ie. the induced subset of a query on the current object set is the underlying object set of the next query step.

2.2 Structures in the attribute space

If we visualize some sets induced by various predicates in a Venn diagram (Figure 1), we observe that they have distinct relationships to each other. These relationships can be used to select predicates that could be of more interest to the user for extending his current query.

Figure 1. An example object set with four result sets



For example, Figure 1 shows the four induced subsets of the query predicates QP_1, QP_2, QP_3 and QP_4. The subsets of QP_1 and QP_2 are fully contained inside QP_3. QP_3 and QP_4 intersect, but are not fully contained in a subset of any other query predicate. If we would face this situation during a retrieval task and have to propose some query predicates to extend the user's query, we can argue that QP_1 and QP_2 are not interesting in this retrieval step as their induced object sets are fully contained inside QP_3. They become important as soon as the user would have extended his query by QP_3.

To illustrate, let the object set be the set of all files in a user's home directory, QP_4 be the query predicate "filetype=picture", QP_3 the predicate "filetype=music", QP_1 the predicate "genre=rock" and QP_2 the predicate "genre=hip hop". If the user looks for a file, we would present him with the filetype choice, and ask for the genre as soon he restricted his query to be about music files.

We will now look at five important structures of query predicates which can be used to structure the attribute space.

- If the induced set of a query predicate QP_A is fully contained in the induced set of another predicate QP_B, we call QP_A a *sub-predicate* of QP_B. In our example, the music genre predicates are sub-predicates of "filetype=music".
- If the induced set of a query predicate QP_A is equal to the induced set of a query predicate QP_B, we call QP_A *synonymous* to QP_B.
- If the induced set of a query predicate QP is equal to the whole object set, we call QP a *context predicate*. All non-context predicates are sub-predicates of the context predicates.
- If the induced set of a query predicate QP is empty, we call QP an *out-of-context predicate*.
- There are non-context predicates which are only sub-predicates of context predicates. We call them *top-level predicates (TLPs)*. Apart from the context predicates, the induced set of a top-level predicate is not contained in an induced set of any other query predicate of the chosen set. In our example, the predicates QP_3 and QP_4 containing the filetype are top-level predicates in the situation given. They are not sub-predicates of any other predicate.

3. EXTRACTION OF HIERARCHICAL STRUCTURES

In the last section, we defined the relationships of query predicates by relationships and sizes of their induced sets. The induced sets of the interesting predicates were either equal or subsets of others.

A simple way to extract these relationships is to query by the respective predicates and then compare the resulting object sets element by element whether they are contained in one another or whether they are equal. This needs a query per predicate, a sorting step of O(n log n) if the result set is not already sorted and a linear comparison.

We can detect those relationships more efficiently by doing queries on the induced set of a query predicate. Instead of querying the database directly, the process uses an intermediate representation of the data set's contents to be independent from the physical organization of the data set and to be able to make use of the CPU's ability to do fast operations on bitstrings.

3.1 An example

Let us compare the relationship between the predicates QP_1 and QP_2. We want to know whether their induced sets are fully contained in one another or whether they are equal.

If we query the induced set of QP_1 by QP_2 and vice versa, we can decide whether one of the five structures is given by looking on the size of the resulting sets.

If the size of both resulting sets is equal, QP_1 is synonymous to QP_2.

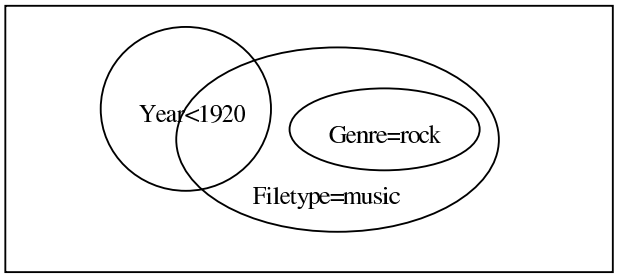


Figure 2. The size of the result sets of adjacent queries

Assume that if we query the induced set of QP_1 by QP_2, the size of the result set does not change relative to QP_1. If we query then QP_2 by QP_1 and the result does change relative to QP_2, then QP_1 is a sub-predicate of QP_2.

For example, if we query the set of "genre=rock" by "filetype=music" in the situation shown in Figure 2, the result set does not change relative to the query "genre=rock" as all files with the attribute "genre=rock" are music files. But if we query the set of "filetype=music" by "genre=rock", the size of the result set changes as the "genre=rock" files are a subset of the "filetype=music" files.

Context predicates can be detected easily as the size of their induced set is equal to the size of the current object set. Out-of context predicates are characterized by the fact the size of their induced set is null.

3.2 Definitions

We will now formally define our observations. First, we formalize the notion of the size of the result set of a query:

Def. #(Q) of a query Q is |o(Q)|, the number of objects in o(Q).

Then, we define the notion of querying two predicates and setting the result in relation to the result of the query of the first predicate:

Def. $p(QP_1 \mid QP_2) := \#(\{ QP_1, QP_2 \}) / \#(\{ QP_2 \})$.

We assume a finite set P of chosen query predicates, the *candidate predicates* and the object set O.

Def. QP_1 is a *sub-predicate* of QP_2, or $QP_1 < QP_2$, if $p(QP_1 \mid QP_2) = 1$ and $p(QP_2 \mid QP_1) != 1$.

Def. QP_1 and QP_2 are *synonymous* if $p(QP_1 \mid QP_2) = 1$ and $p(QP_2 \mid QP_1) = 1$.

Def. QP is a *context predicate* if #(QP) = |O|.

Def. QP is an *out-of-context predicate* if #(QP) = 0.

Def. QP is a *top-level predicate (TLP)* if there is no non-context predicate $QP_i \in P$ with $QP_i > QP$.

The relation p(|) has one property which allows us to save some processing steps later. Assume that we have two predicates QP_1 and QP_2 whose induced sets do no intersect. This implies that both #({ QP_1, QP_2 }) = 0 and #({ QP_2, QP_1 }) = 0 as the query's contents are linked with an boolean *and* which is commutative. If we now look at the definition of p(|), we see that this fact implies that if $p(QP_1 \mid QP_2) = 0$ then $p(QP_2 \mid QP_1) = 0$ for any pair QP_1, QP_2.

3.3 Extracting the top-level predicates

Now we will show how to extract the top-level predicates (TLPs) in a systematic manner working on a fixed set P of candidate predicates.

As we have to compare each pair of elements of P, we will work on a virtual matrix M whose columns *c* and rows *r* are labeled with the elements of P and whose entries are the values of $p(QP_r \mid QP_c)$. As computing p(|) is not a cheap operation, we will try to keep the number of computed entries of the matrix at a minimum.

In a first step we compute #(QP_i) for each $QP_i \in P$ to find all context and out-of-context predicates. We do this so that the context predicates do not interfere with the detection of the TLPs as we have to analyze all sub-predicate relationships to find them. We will not include context predicates in any further operation. We remove out-of-context predicates too, as they are not a candidate for being a TLP or being synonymous.

We select a column and process it until we have analyzed all query predicates. Each column is processed row-wise by computing $p(QP_r|QP_c)$ and the corresponding entry $p(QP_c|QP_r)$. Of course, all computations are cached, so that no p(|) is computed twice.

If we detect a sub-predicate relationship for a column, we do not proceed with the column as it is no longer a candidate for being a TLP, and we mark the column accordingly.

If we detect two predicates to be synonymous, we note that property and merge the columns and rows virtually as all of their entries will be the same and should not be computed twice.

If we are finished with this process, we know the predicates of P who are TLPs, those who are context and out-of-context predicates and those who are synonyms.

The process in pseudocode can be seen in Figure 3.

```
1.  Define a set of candidate predicates
2.  Remove all context predicates from it
3.  Remove all non-context predicates from
    it
4.  FOR each candidate predicate c
5.    IF( c is marked as subordinate )
6.      continue with next c
7.    FOR each candidate predicate r
8.      IF( c == r )
9.        continue with next r
10.     get p1 = p( QPr | QPc )
11.     IF( p1 == 0 )
12.       continue with next r
13.     get p2 = p( QPc | QPr )
14.     IF( p1 == 1 and p2 == 1 )
15.       mark c and r as synonyms
16.     IF( p2 != 1 and p1 == 1 )
17.       mark r as subordinate
18.     IF( p2 == 1 and p1 != 1 )
19.       break inner loop, as c < r
20.   IF we checked all r
21.     mark c as TLP
```

Figure 3. The TLP extraction process

This process is quadratic in the number of predicates in P. Computing p(|) is linear in the number of objects.

3.4 Computing p(|)

For a fast extraction of top-level predicates in wide area of applications, we will not work on our original data set to compute p(|).

As stated in the introduction, attribute-based data is used in many contexts. The choice of physical representation of attribute-based data is probably as large as its applications; it ranges from simple record-based structures to RDBMS.

To decouple the speed of our extraction process from the speed of the data storage to do queries, and to use the processor to do certain operations, we will work on bitstrings to compute p(|).

For each query predicate QP, we extract a *predicate map*, which is an uncompressed word-based bitmap index that contains a bit for every object in the system. If the induced set of QP includes the object, its associated bit in the predicate map is 1.

This predicate map is a small cache of the contents of the data storage. If the data storage is changed, the bitstrings must be changed accordingly. The size of a predicate map is *(number of objects) / 8* bytes.

For computing p(|), we need the intersection set of the query predicate's object sets. With the usage of predicate maps, this is just the boolean AND of the associated bitstrings.

3.5 Choosing candidate predicates

Our algorithm assumes that we have a fixed set of query predicates which are candidates for being top-level predicates. We did not give any detail yet on how we choose these predicates.

A first possibility is to build the predicates using all available attribute names, all available attribute values and all defined operators for the attribute value type. This set of predicates is huge, and as our algorithm is quadratic in the number of predicates we want to restrict this number to a feasible amount.

A simple system could use all available attribute names whose value domain is limited to a small number of values. It builds predicates out of them by using the "=" operator and all available values.

A more sophisticated system could look at the distribution of the values of each attribute and divide it into a small number of intervals which include the same number of values.

3.6 Examples

To show that the algorithms extract worthwhile results, we will first show their result sets on simple artificial examples As a first example, we look on a set of two objects,

- object 1 with the attributes {A,X} and
- object 2 with the attributes {A,Y}.

The algorithm will identify A as a context-predicate, and X and Y as top-level predicates. Thus, the object set would be partitioned by the query predicates X and Y.

In the next step, we add a third object which is named with attributes as follows:

- object 3 {B,Z}.

The algorithm would recognize B and Z as being synonyms and return A and B=Z as top-level predicates. X and Y are identified as being sub-predicates of A. Thus, the user would be presented with two choices to extend his query: A and B=Z.

The addition of a fourth object named

- object 4 {B,Q}

results in the identification of A and B as top-level predicates. The user would be presented with these two attributes, each representing a subset of the current query result. Both can be used to extend the current query and to investigate one of the subsets further.

Now we will have a look on the extraction algorithm from the user's point of view. Current hierarchical file naming is unable to represent multi-hierarchical structures, which is a major burden to use it for naming real-world data. We assume an object set that uses the attributes *ProjectA, ProjectB, Papers, Sourcecode* in a number of files:

- file subset 1 {ProjectA, Papers},
- file subset 2 {ProjectA, Sourcecode},
- file subset 3 {ProjectB, Papers},
- file subset 4 {ProjectB, Sourcecode}.

If we query this object set for *ProjectA*, we are presented with the top-level predicates *Papers* and *Sourcecode* as choices to extend our query further. If we query the set for *Papers,* we get *ProjectA* and *ProjectB* as predicate choices. Each of the files is part of two hierarchies, the *ProjectA* hierarchy and the *Papers/Sourcecode* hierarchy. Depending on which entry point we chose, the algorithm allows to user to refine her search using the respective hierarchy.

4. BROWSING AN ATTRIBUTE-BASED NAMESPACE

A top-level predicate is a valid abstraction for all objects in its induced set, as all these object satisfy the constraint given by the TLP. Furthermore, candidate predicates that are sub-predicates of a TLP can be assumed as being included by the TLP. Thus, the TLP is a valid abstraction of its sub-predicates too.

We will now complete the basic notion of top-level predicates with two definitions to get a complete view of an explored object set.

4.1 Local objects

If we look at Figure 1, we observe that there are objects in the object set which are not covered by any predicate. We call these objects *local objects*, any refinement of the query using one of the proposed predicates would exclude them.

Detecting the local objects is done both easily and fast by subtracting the intersection of all predicates from the current object set. This can be done using boolean operations on the predicate maps.

4.2 Important top-level predicates

The number of top-level predicates is not inherently limited. In the case of each candidate predicate only intersecting with some

other candidates but not being completely included, the set of candidates is the set of TLPs itself.

Thus, if we have a large number of TLPs, we would like to reduce it. Often, we can do so by removing a TLP whose induced set is contained in the remaining set of TLPs.

We call a TLP whose induced set is not covered completely by other TLPs an *important top-level predicate*.

We can use this definition to build algorithms which reduce the set of TLPs to a smaller number. A simple example of an algorithm would be to start with a new empty set of predicates and add the TLP which extends the coverage of the new set the most. We keep on doing this until the new set covers the same objects as the original set of TLPs, i.e. all important TLPs are included. This algorithm is shown in Figure 4 in pseudo-code.

We can run this algorithm more than once to get groups of TLPs covering the same set each, but which have worse intersection properties.

```
1. T:= set of TLPs
2. C:= coverage of T
3. G:= new empty set of predicates
4. while o(G) ≠ C
5.   move predicate x from T to G with
6.     o(x) ∩ o(G) = min. and
7.     o(x) \ o(G) = max.
```

Figure 4. An example for a grouping algorithm

4.3 User interface for browsing

We have now structured our object set in a set of TLPs plus the remaining local objects. Furthermore, we grouped the TLPs in groups which provide a "good" coverage of the object set each.

We can use this structure to construct a user interface which completely hides the syntax and semantic of query formulation. It provides an interface to hierarchically browsing the object set. In each browsing step, the user is presented with a set of predicates (the TLPs) plus a set of local objects. Note that the partitioning of the current object set into top-level predicates and local objects is *complete* in the sense that every object in the current set is either a local object or included in one of the top-level predicates. Thus, all the objects are *reachable* through the query mechanism.

The user can select a TLP to zoom further into the object set. This TLP is added to the current query. The user can also zoom out by selecting an "Up" button, which removes the last predicate of the current query and thus returns to the previous object set.

The user does not have to be aware of him modifying a query by his actions. The clue he is given that he works on a attribute-based system is the format of the choices. They resemble attributes, which have formats and use terms that are familiar to the user.

This interaction can be made nearly indistinguishable from the accustomed interaction with a hierarchical file system. Figure 5 shows a prototype interface where the interaction with the attribute-based namespace is embedded in the Microsoft Windows Explorer as a Namespace Extension. The data stems from an attribute-based file system which accumulates normal file metadata along with file-type specific metadata like document titles and authorship. The extracted predicates are grouped by their attribute names.

5. EVALUATION

To evaluate the quality of the extracted hierarchy and to test the usability of the proposed browsing interface, we conducted a controlled user experiment which applied the extraction procedure of Section 3 to an attribute-named file system.

We compared the user's interaction with three user interfaces. The first one is a basic form-based query interface ("Traditional-UI", Figure 6) which lets the user choose from all available attribute names and values. The second one is a form-based interface which proposes query extensions with the algorithms of Section 3 ("Enhanced-UI", Figure 6). The third one is a file system browser like interface which mimics the interaction of current desktop file browsers ("Explorer-UI", Figure 7) and makes use of the techniques described in Section 3 and 4.

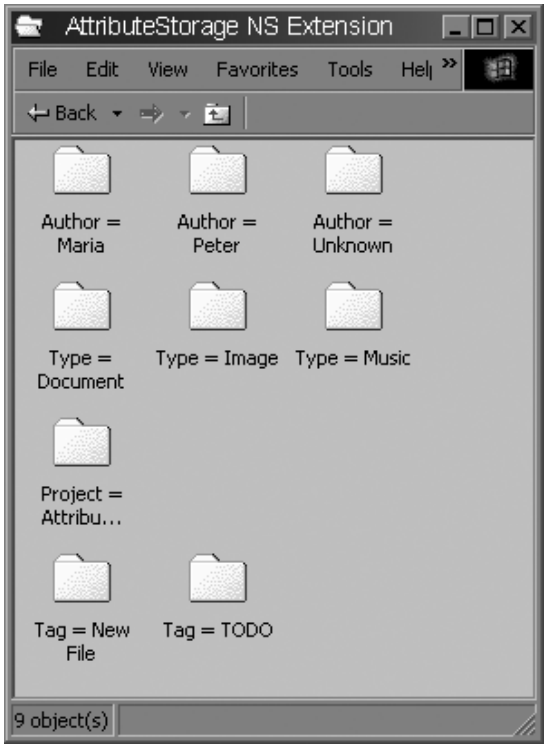


Figure 5. Hierarchical browsing of attribute-based data

The Traditional-UI and the Enhanced-UI differ only in the attribute names and values the user can choose from. Thus, differences in the user's performance can be accounted to the number and quality of the proposed attribute names and values.

The Enhanced-UI and the Explorer-UI provide the same choices for extending a query, but do so with a different look and interaction. Here, differences in the user's performance can be accounted to the differing presentation.

5.1 Interfaces

The basic form-based query interface ("Traditional-UI") allows the user to formulate queries dynamically. Each part of the query consists of the attribute name and a value to restrict the attribute to. With the respective button, the user can delete the whole query to start over or add an extension to the query. When selecting a value, the result set in the lower half of the window changes instantaneously. To simulate a large object set, only result sets of less than seven objects are displayed, otherwise a line saying "Too many files" is shown in order to force the user to refine the query.

In each query step, the user can choose from all known attribute names with the left pull-down menu. When the user has chosen an attribute name, the right menu lets the user choose from all known values of this attribute. Only the lowermost query predicate line is editable.

The enhanced form-based interface ("Enhanced-UI") works like the basic one, but restricts a query step's choice of attribute names and values to the top-level predicates of the current result set. When the user adds a new query restriction, or query line, the system extracts the top-level predicates of the current result set, divides them into attribute names and values and inserts them accordingly. The lower list shows the local files of the current query instead of its whole result set.

The third interface embeds the query extension extraction in a desktop file-browser-like interface ("Explorer-UI", Figure 7) as proposed in Section 4.3. The two buttons at the top allow the user to remove the last query predicate ("Up") and to clear the current query to start over ("New Query"). The "Query:"-line displays the current query. The list in the middle of the window shows the extracted top-level predicates of the current query's result set. A click on one if its items extends the query accordingly. The lower list shows the local files of the current query. Note that the Explorer-UI displays exactly the same information and gives the user the same choices as the Enhanced-UI, it only uses a different presentation.

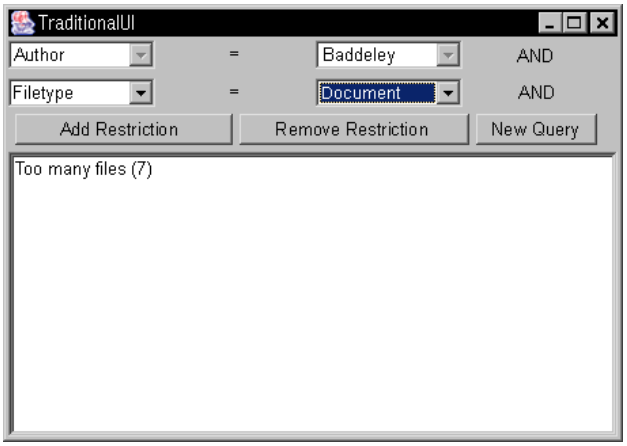


Figure 6. Form-based UI ("Traditional-UI", "Enhanced-UI")

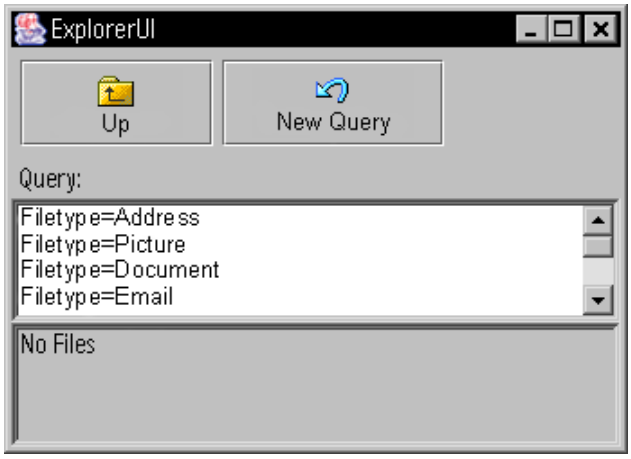


Figure 7. Hierarchical browser ("Explorer-UI")

5.2 Methodology

The experiments were conducted with a laptop equipped with an optical mouse on which the interfaces were running. An instructor was attending who quickly demonstrated each interface with one retrieval task, but didn't provide any further help or answers afterwards.

The subjects were told as a background that they were an assistant to a professor who asked them to find certain documents on his computer for him. They were given tasks which contained keywords that were similar to or matched object metadata (see Figure 8).

We prepared three task sets of eight tasks each. The tasks were structurally equivalent among the task sets but differed in the task's keywords and exact phrasing. To exclude any influence of the tasks' difficulty on the result, we kept the order of structurally equivalent tasks consistent between the tasks sets. For each interface, the users had to complete the tasks of one task set. This results in 24 tasks per subject on three interfaces. The mapping between task sets and interfaces was fully balanced which results in six different orders of interfaces. A task was counted as successfully solved when the subject clicked on the right document to open it. Unsuccessful tasks were those where the subject did not want to continue searching.

"Prof. X worked on project together with IBM in 1999. Find the project report."

"Look for a presentation on information visualization, which Prof. X did for his company in 2000."

"Prof. X wants to create a research poster for which he needs the university's logo."

"Prof. X has to give marks for the Java exercises. Find Peter's Java program of the winter term 2001.

Figure 8. Examples of tasks

After the experiments the subjects were given a questionnaire which asked them about their computer, internet and information retrieval experience. Then they were asked to order the user interfaces according to their personal preference, how much the

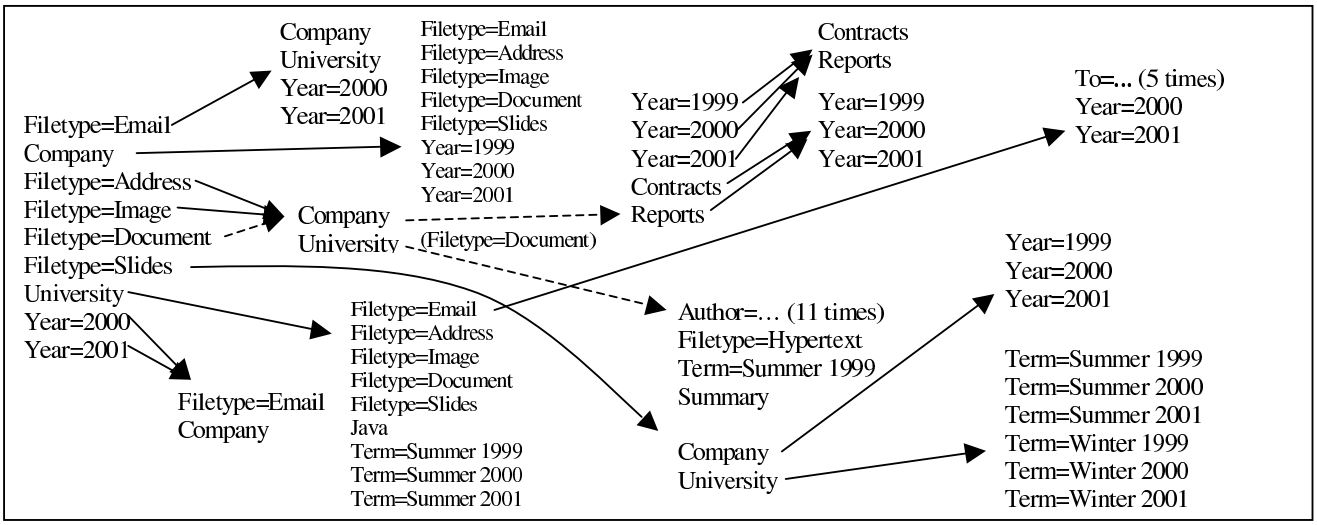


Figure 8. A part of the extracted hierarchy of the experiment's namespace

reduction of choices in the Enhanced-UI helped them when compared to the Traditional-UI, and how much the query extension proposals of the Enhanced-UI and the Explorer-UI met their expectations on how to continue with their query.

We recruited 12 subjects for our experiments, all being students of a wide range of university programs. All had experience with using Microsoft Windows GUIs, had been using computers on the usual level for web browsing and word processing, and were familiar with form-based queries from libraries' web interfaces.

During the experiment, we logged the user interactions (mouse clicks, list choices and list selections) along with exact timestamps.

5.3 Data set

The file set consisted of 231 files named with 15 attribute names, which had between zero ("tag attributes") and 14 values. These appeared in the choices of the Traditional-UI. For the empty result set, there were nine top-level predicates of four distinct attribute names (these were displayed initially in the upper pane of the Explorer-UI and in the attribute choice of the Enhanced-UI). The size of the metadata set was chosen so that it is not too big for list selection, which is needed in the traditional interface and not too small for mining. Figure 8 shows a part of the extracted hierarchy. Each block of predicates shows the TLPs of the respective subset of object set, with the TLPs for the whole object set on the left. A link symbolizes the query extension with the respective predicate and leads to the TLPs of the result set of the query. For each result set, the complete list of TLPs is shown, but only a subset of the possible query extensions are followed.

In all three systems, the query processing and mining times were negligibly short.

5.4 Results

From our log files, we extracted the ratio of successfully solved tasks (Table 1). The differences between the means are statistically significant ($F(2, 22) = 4.661$, $p < 0.021$).

Table 1. Successful tasks (out of 8)

Number of solved Tasks	Mean	Standard Deviation (SD)	Standard Error (SE)
Traditional	7.17	1.34	0.39
Enhanced	8.00	0.00	0.00
Explorer	8.00	0.00	0.00

Table 2 shows the mean times for the task completion. These numbers include tasks that could not be solved, they were counted with 200 seconds, an artificial time limit which is among the highest times for successfully solved tasks. The differences between mean times are statistically significant ($F(2,22) = 17.385$, $p < 0.0005$).

Table 2. Retrieval Times

Retrieval Time (sec.)	Mean	SD	SE
Traditional	58.88	50.40	5.14
Enhanced	39.02	35.52	3.63
Explorer	17.27	9.22	0.94

The logging of interactions were used to calculate the number of query refinement steps, which includes the steps backwards (Table 3). The differences between the means are not statistically significant.

Table 3. Number of Query Refinements

Refinement Steps	Mean	SD	SE
Traditional	3.59	2.23	0.23
Enhanced	3.88	2.11	0.22
Explorer	3.52	1.17	0.12

We continue with the results of the post-experiment questionnaire. The first question asked for the personal preferences (Table 4).

Table 4. Personal Preference

Explorer > Enhanced > Traditional	10 out of 12
Explorer > Traditional > Enhanced	1 out of 12
Enhanced > Explorer > Traditional	1 out of 12

The next question asked how helpful the user considered the preselection of attributes for the form-based interface (Traditional vs. Enhanced-UI, Table 5) on a scale of 1 (not helpful) to 6 (very helpful).

Table 5. Helpfulness of preselection

	Mean	SD	SE
Form, Helpfulness	4.33	1.30	0.38

The third and fourth question asked how well the proposed query extensions met the user's expectation on how to continue with their further search for both the Enhanced-UI and the Explorer-UI (Table 6) on scale from 1 (not representative for expectation) to 6 (very representative). The differences between the means are statistically significant ($F(1, 11) = 2.67$, $p < 0.013$).

Table 6. Representativeness of proposed query extensions

	Mean	SD	SE
Enhanced-UI	4.67	0.78	0.22
Explorer-UI	5.33	0.65	0.19

5.5 Discussion

The experiment was conducted to get answers to two research questions. Our first thesis is that the extracted query extension proposals are a good abstraction of the underlying object set and aid a user in refining a query.

The results of the experiment show that the user is able to solve more tasks (Table 1), and needs less time per task (Table 2) and per refinement step (Table 2 and 3) when restricting the choices in the form-based interface to the top-level predicates of the current result set. The post-experiment questionnaire reveals that the users considered the restriction of possible query extensions helpful when compared to the presented set of query extensions in the Traditional-UI (Table 5). Furthermore, they considered the presented query extensions as representative for their further query (slightly dependent of their presentation, Table 6).

The general problem here is the one of menu selection. For each query refinement, the user has to choose from a list of refinement alternatives. The time needed for this grows with the length of the list and can be optimized with the use of hierarchical menus [7]. Thus it seems that the relative lower number of choices between the Traditional-UI and the Enhanced-UI could be the cause for the better user performance, independent of the quality of the proposed quality. This, however, only takes the selection times of single refinement steps into account. When the quality of the proposed alternatives was bad, it would result in more refinement steps, more errors and a longer overall time, of which none is the case. Thus we can conclude by our measurements that top-level predicates are a good abstraction of the underlying object set. This thesis is supported by the user's personal observations (Table 4, 5, 6).

Our second thesis is that a file-browser like interface (Section 4.3) is preferable over a form-based interface for retrieving objects from an attribute-based namespace. While the success rate is not improved further, the Explorer-UI enables the user to solve the tasks in less time per task (Table 2) and per extension step (Table 2 and 3). Furthermore, most users prefer the usage of the file-browser like Explorer-UI over the form-based Enhanced-UI (Table 4) and even have the impression that the quality of query extension proposals is better (Table 6) while they are actually the same.

The research of menu selection can also be applied to explain the differences between the Enhanced-UI and the Explorer-UI, which both present the same choices in a different way. While the Enhanced-UI introduces one artificial hierarchy layer (choice of attribute, then choice of value), the Explorer-UI displays these choices as a flat list. When applied to a short list, the introduction of an additional menu layer lengthens the time needed to make the overall selection. [7]. This is the case here, and explains the better performance of the file-browser like presentation in the experiment along with the facts that one additional mouse click is needed for opening the pull-down choice, and the two pull-down choices give less overview than one list with all alternatives. Combined with the users' preference of the Explorer-UI, our thesis that an file-browser like presentation is favorable over a form-based interface for presenting top-level predicates and local files to the user is supported.

6. RELATED WORK

6.1 Query formulation

Current user interfaces for querying attribute-based data sets are mostly variations of the standard model of forming constraints on attributes using boolean operators. While early systems like the Semantic File System [4] demanded the user to do this manually using a command line, newer systems use form-based methods or even interfaces for direct manipulation (Dynamic Queries [5], Filter-Flow [10], Presto [3]) to embed the syntax and semantics in a graphical interaction.

Form-based methods especially benefit from limiting the number of choices of attribute names and values or from a limited number of proposed alternatives.

Early systems let the user choose from all attribute names and possible values [4], newer systems restrict that choice to attribute names and values that are actually present in the queried object set [9] as those result in the only reasonable extensions to the query extensions. If textual entry of attribute names and values is possible, one can let the system try guessing the anticipated entry or use fuzzy search methods to find the real attribute name or value.

6.2 Hierarchical structures

Not every search for information starts with a clear goal in mind. If the user has a rather vague idea of what he's searching for or if he's not yet familiar with the system or the dataset, it is beneficial to be able to browse a data set without having to explicitly formulate exact queries or having to be familiar with query formulation.

6.2.1 Attribute-based data

KnownSpace [1] analyzes user access patterns and clusters the objects to be able to map the object set in a multi-dimensional space. However, [1] gives no details on the used cluster method.

6.2.2 Text corpora

Much work has been conducted in the field of extracting hierarchical structures from text corpora.

Sanderson and Croft [6, 8] present a method to extract subsumption hierarchies of terms. They use term relationships to find a hierarchy of concepts in a document set. After extraction of a set of terms, the system uses a heuristic to find hierarchical structures between them. The heuristic is based on the relative frequency of occurrence of the terms x and y in each other's context:

$$P(x|y) = 1 \text{ and } P(y|x) < 1.$$

However, the definition fails in the case when a few y do not co-occur with the term x. Therefore they relaxed the condition to be:

$$P(x|y) \geq 0.8 \text{ and } P(y|x) < P(x|y)$$

The value 0.8 was chosen through informal analysis of term pairs.

Note that our *sub-predicate* definition has the same structure as the one used by Sanderson and Croft [8] to characterize subsumptions between terms of documents in text corpora. Whereas they compare the co-occurrence of terms extracted directly from text corpora, we compare the relationships of metadata associated with objects. As the relationship of terms in natural language text corpora is not always clean, they relax their original definition to be able to recognize subsumption relationships that have a few violations in the text corpora (see related work section). We stay with the sharper definition as we work on metadata which is already an abstracted version of the object set. Note that this is no restriction of generality as our method works with a weaker sub-predicate definition as well.

Scatter/Gather [2] is a method to hierarchically browse data sets. In each retrieval step, the current data set is clustered and the cluster's contents are summarized. These summaries allow the user to restrict the current data set by choosing a cluster and then reiterate. Scatter/Gather has been applied to text corpora but there has been no published attempt yet to apply it to attribute-based data.

7. Conclusion

We identified properties of query predicates which allowed us to structure the attribute namespace hierarchically. We presented a process to extract this structure from an object set, which resulted in a set of so-called top-level predicates (TLPs).

Applied to a traditional form-based query interface, the top-level predicates allow the user to extend a query by choosing from a relatively small number of alternatives.

We introduced the notions of important top-level predicates and local files, which, in conjunction with the hierarchical "zooming property" of top-level predicates, allow for browsing of attribute-named object sets. This was applied to create an user interface which hides the syntax and semantics of query formulation and allows the user to browse an attribute-named data set hierarchically.

A controlled user experiment confirmed the quality of the set of top-level predicates as a valid abstraction from the underlying object set and showed that the proposed browsing interface is preferable to form-based interfaces for querying an attribute-named object set.

8. REFERENCES

[1] Baeza-Yates, R. Jones, T. Rawlins, G. New Approaches to Information Management: Attribute-Centric Data Systems. In *Proceedings of the 7th IEEE International Symposium on String Processing Information Retrieval (SPIRE'00)*, 2000.

[2] Cutting, D. Karger, D. Pedersen, J. Tukey, J. Scatter/Gather: A Cluster-based Approach to Browsing Large Document Collections. In *Proceedings of the 15th ACM SIGIR '92*. Denmark, 1992.

[3] Dourish, P., Edwards, K., Lamarca, A., Salisbury, M. Presto: An Experimental Architecture for Fluid Interactive Document Spaces. In *ACM Transactions on Computer-Human Interaction*. Vol.6, No.2, June 1999. 133-161.

[4] Gifford, D., Jouvelot, P., Sheldon, M., O'Toole, Jr J. Semantic File System. In *Proceedings of the 13th ACM Symposium on Operating Systems Prinici*, CA. October 1991. ACM Press. 16-25.

[5] Liao, H.S., Osada, M., Shneiderman, B. Browsing Unix Directories With Dynamic Queries: An Evaluation of Three Information Display Techniques. *Technical Report CS-TR-2841*. Dept. of Comp. Sci., U. of Maryland. February 1992.

[6] Lawrie, D.J. Croft, W.B. Discovering and comparing concept hierarchies. In *Proceedings of the 24nd ACM SIGIR '01*.

[7] Norman, K.L. The Psychology of Menu Selection: Designing Cognitive Control at the Human/Computer Interface. Ablex Publishing Corporation, 1991.

[8] Sanderson, M. Croft, W.B. Deriving concept hierarchies from text. In *Proceedings of the 22nd ACM SIGIR '99*.

[9] Wills, E., Giampaolo, D., Mackovitch, M. Experience with an Interactive Attribute-Based User Information Environment. *Technical Report WPI-CS-TR-94-2*. Worcester Polytechnic Institute, Worcester, MA. 1994.

[10] Young, D. Shneiderman, B. A graphical filter/flow model for Boolean queries: An implementation and experiment. *Journal of the American Society for Information Science*, 44(6):327-339, July 1993.