Feedback-driven Result Ranking and Query Refinement for Exploring Semi-structured Data Collections [*]

Huiping Cao, Yan Qi, K. Selçuk Candan
Arizona State Univ.
Tempe, AZ 85283, USA
{hcao11,yan.qi,candan}@asu.edu

Maria Luisa Sapino
Univ. di Torino
Torino, Italy
mlsapino@di.unito.it

ABSTRACT

Feedback process has been used extensively in document-centric applications, such as text retrieval and multimedia retrieval. Recently, there have been efforts to apply feedback to semi-structured XML document collections as well. In this paper, we note that feedback can also be an effective tool for exploring (through result ranking and query refinement) large semi-structured data collections. In particular, in large scale data sharing and curation environments, where the user may not know the structure of the data, queries may initially be overly vague. Given a path query and a set of results identified by the system to this query over the data, we consider two types of feedback: *Soft feedback* captures the user's preference for some features over the others. *Hard feedback*, on the other hand, expresses users' assertions regarding whether certain features should be further enforced or, in contrast, are to be avoided. Both soft and hard feedback can be "positive" or "negative". For soft feedback, we develop a probabilistic feature significance measure and describe how to use this for ranking results in the presence of dependencies between the path features. To deal with the hard feedback efficiently (i.e., fast enough for interactive exploration), we present finite automata based query refinement solutions. In particular, we present a novel LazyDFA$^+$ algorithm for managing hard feedback. We also describe optimizations that leverage the inherently iterative nature of the feedback process. We bring together these techniques in *AXP*, a system for *adaptive and exploratory path retrieval*. The experimental results show the effectiveness of the proposed techniques.

Categories and Subject Descriptors

H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval: Relevance feedback.

General Terms

Algorithms, Measurement.

Keywords

Relevance feedback, inter-dependent structural feature, feature cover, data-centric XML.

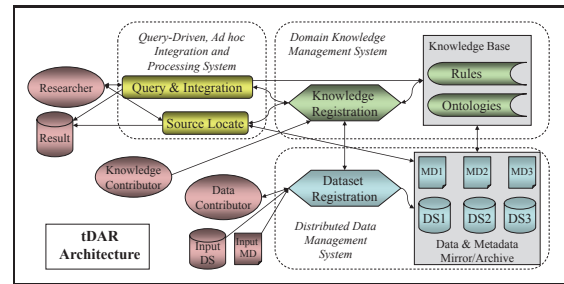**Figure 1: Overview of tDAR (it currently curates 48 projects, 71 datasets, and 100 coding sheets accessible through http://dev.tdar.org/tdar): provides search, integration, and querying functionalities to advance scientists to conduct synthetic and comparative research over data and metadata deposited by different researchers [24, 22, 23, 6]**

1. INTRODUCTION

Scientific data needed to address the most pressing research questions are almost never collected by a single research team. tDAR [24, 22, 23, 6] archives and provides integrated access to a multitude of data sets and metadata, collected by different researchers within the context of different projects and deposited to tDAR for sharing (Figure 1). Consider a scientist with a specific research problem accessing tDAR to identify relevant databases. When this scientist poses a query to tDAR, her query might match many relevant databases and data tables that are not immediately familiar to her. For this scientist to be able to leverage the available data sources in tDAR as effectively as possible, techniques that support effective data exploration are needed.

In the absence of precise knowledge about the data, users' queries may be initially vague. The results provided by the system in response to such imprecisely formulated queries, however, may contain hints to help users make their (initially vague) specifications iteratively more informed and precise. Especially when users are not sufficiently informed about the data (or sometimes of their interests) to formulate precise queries, feedback based exploration can play a critical role in helping find relevant information.

Feedback-based exploration has been shown to be very effective in non-structured document-centric domains, such as text retrieval [27, 25] and multimedia [18] where formulating precise queries is often impossible. This is sometimes referred to as bridging the *semantic gap* between the user and the database: given a query (say a sample data object provided for a "similarity" search), which features of the objects are relevant for the user's query may not be known in advance. In the context of information retrieval (IR) and multimedia applications, where the data and queries on text collections are often modeled as keyword vectors [28], user
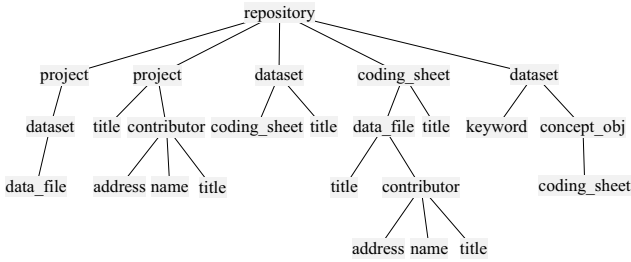
Figure 2: A small tree-structured data fragment

feedback about the relevance or irrelevance of the result documents is used for adjusting the weights in the query vector by increasing the weights of the more relevant keywords [28, 36].

Unlike the prior document-centric efforts [28, 36, 27, 25, 18], in this paper, we focus on the use of feedback in *data-centric* applications, such as large scientific database collections like tDAR, which houses heterogeneous and complex datasets. In data sharing and curation environments where the user may not know the underlying structure of the data which she is querying (thus is not able to form precisely specified statements of interest), queries over structured and semi-structured data (e.g., tree or graph) may also suffer from similar problems. In particular, an incompletely specified query may return too many results. Consider, for example, the data in Figure 2, which describes several metadata items in a scientific repository; A query "*I am interested in the 'titles' in the current repository*" (i.e. $//title$), would return six matches, each corresponding to a path from the root to a node labeled with "title". While there are some initial attempts, such as [21, 30, 31, 29, 15], to address the challenge of enabling feedback on structured data, this is still a largely unexplored area, with no effective strategies to support user feedback. One reason for this is that, while there are plenty of works on feedback on data that can be represented as vectors, not all data and queries are easy to map onto a vector space. This is especially true in semi-structured and structured data (e.g., data-centric XML, graphs), where the *structure* is often a critical component of the data. Path expressions – which express the desired characteristics of the paths on the data graph– combine requirements about values (such as the tags of an XML data object) with requirements about the structural organization of these values. This renders the traditional feedback techniques inapplicable.

1.1 Contributions of this Paper

In this paper, we focus on the problem of feedback-based exploration of data-centric XML collections in large data sharing and curation environments. Such queries are often represented in the forms of path or tree (twig) queries. For processing, query twigs are often partitioned into the constituent path queries and the results are joined during post-processing [9, 12]. Following the same approach, we focus on path expressions of type $P^{\{/,//,*\}}$. Extending these results to twig queries is future work.

When the user is exploring the data within the context of an initial query but does not have well-defined correctness criteria in mind yet, she may want the system to rank the results in the next iteration according to the positive or negative feedback she provides on the current results. To accommodate such declarations of preference, the system needs to support *soft feedback* and *ranking*. The soft feedback process is most suitable for users who are exploring the data and may change their minds about what they are looking for as they learn more about the data. In some other cases, we recognize that after observing the initial set of results returned by the system, the user may be able to identify certain structural aspects of the returned paths that are critical for the correctness, but not included in the original query. To accommodate these explicit as-
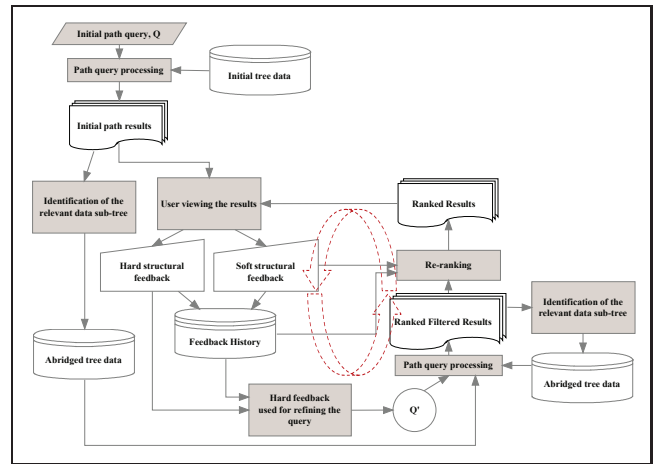

Figure 3: Overview of the *AXP* framework

sertions of desirability or un-desirability [34], we aim supporting *hard feedback* and *filtering*. Hard feedback process[1] is suitable for expert users who know what they are looking for but do not know the data to formulate "accurate" queries.

1.1.1 Contributions

More specifically, our contributions include the following:

- We propose and formalize four kinds of feedback (SHOULD, SHOULD-NOT, LIKE, and DISLIKE) to filter and reorder results to a path query. The proposed algorithms provide the users with the option of denoting which part of the results they want their feedback to focus on or to let the system discover this automatically based on the feedback.

- In order to support soft feedback (LIKE and DISLIKE), we propose a set of structural features suitable for feedback processing. We then propose probabilistic measures and algorithms for structural feature significance analysis to help reorder path results based on their relevance to the user. We also show how the structural dependencies between path features can be taken into account for semantically correct and effective feedback-based ranking. [2]

- We develop data structures and algorithms to refine user's query based on positive and negative hard feedback. In particular, we propose an efficient lazy automata based mechanism to capture both types of hard feedback. We propose a `LazyDFA` algorithm for minimizing the automaton construction time. Unlike the lazy-DFA schemes reported in the literature for message filtering [14, 13, 8], `LazyDFA` consists of two parts: `LazyDFA`$^+$ and `LazyDFA`$^-$ to deal with conjunctive and disjunctive filtering statements. We also present techniques to minimize DFA storage and execution costs.

We bring together these techniques in an *adaptive, and exploratory, path retrieval (*AXP*)* framework. The goal of *AXP* is to let the users explore the available data starting from an initial query context and by providing hard and soft feedback as appropriate.

1.1.2 Overview of the AXP Framework

An overview of the *AXP* framework is presented in Figure 3:

- Initially, the user poses a query.

---

[1]Hard feedback based exploration is supplemented by "breadcrumb" and "undo" functions: the user can *undo* recent feedback and go to an earlier state in the exploration following *breadcrumbs*.
[2]A preliminary version of the soft feedback process appeared as a poster at [7].

id	answer to query "//title"	feedback
1	repository/project/title	
2	repository/project/contributor/title	
2.1	repository/project	○ SHOULD ○ LIKE ◉ NEUTRAL ○ DISLIKE ○ SHOULD-NOT
2.2	repository//contributor	○ SHOULD ○ LIKE ◉ NEUTRAL ○ DISLIKE ○ SHOULD-NOT
2.3	repository//title	○ SHOULD ◉ LIKE ○ NEUTRAL ○ DISLIKE ○ SHOULD-NOT
2.4	project/contributor	◉ SHOULD ○ LIKE ○ NEUTRAL ○ DISLIKE ○ SHOULD-NOT
2.5	project//title	○ SHOULD ○ LIKE ◉ NEUTRAL ○ DISLIKE ○ SHOULD-NOT
2.6	contributor/title	◉ SHOULD ○ LIKE ○ NEUTRAL ○ DISLIKE ○ SHOULD-NOT
3	repository/dataset/title	
4	repository/coding_sheet/title	
5	repository/coding_sheet/data_file/title	
6	repository/coding_sheet/data_file/contributor/title	

Figure 4: The interface to help the user provide feedback: the user highlights multiple different feedbacks in one of the path results; feature "project/contributor" in path "repository/project/contributor/title" is marked as SHOULD-NOT feedback, "contributor/title" is marked as SHOULD feedback, and "repository//title" is marked as LIKE feedback

- The path results that match this initial query are retrieved (at this stage these results are used to form an abridged data representation to help efficient feedback processing).

- Since the number of the query results may be prohibitively large for the user to view, explore, and provide feedback on, a subset of the resulting paths are presented. Initially, the the system picks a random[3] K among the results to present to the user. (At each later iterations, the user is presented with subsets improved based on her feedback.)

- From the top-K results, the user marks parts of the presented results as preferred or not preferred (soft feedback). The user may also highlight some other parts as required or to be eliminated (hard feedback).

- Based on the hard feedback and the initial query, a new query is formulated and executed on an abridged tree, formed during the previous iteration. The results are written to a new abridged tree (to support exploration in later iterations).

- The results are re-ranked based on the soft feedback, and the top-K best results are presented to the user. The re-ranking can take into account both the feedback in the current iteration as well as previous iterations.

- Hard/soft feedback can be used to annotate the data and shared within the framework.

AXP maintains the feedback history to prevent conflicts in hard feedback as well as to minimize redundant work. In particular, hard and soft feedback can decay over time to give more emphasis on more recent feedback during exploratory data access[4]. To allow users highlight their feedback on their query results, tDAR prototype provides the feedback interface shown in Figure 4.

1.2 Related Work

Query refinement through relevance feedback has been well studied in the fields of information retrieval [27, 25] and multimedia [18]. Most of the existing algorithms rely on the *vector model* [28] of data to support feedback-based query refinement. Intuitively, the vector describes the composition of the data or the query in terms of its constituent features (such as color, edge, or keyword). The vector model is especially suitable for supporting feedback, because the user feedback can be used both (a) to move

---

[3]This initial set may not represent the entire result set well, but often contains sufficient information to bootstrap the feedback process. Alternatively, one can try to select and present an initial set of paths that will collectively help highlight the critical structural alternatives in the data. This is ongoing research.

[4]In this paper, we do not investigate the challenge of finding best way to implement feedback decay for path queries.

the initial query vector in the vector space in a way that better represents the user's intentions or (b) to re-assess the significance of the features so that the query better reflects user's feedback.

As we described in the introduction, relevance feedback may also be needed when dealing with structured and semi-structured data. While vector models are proven to be effective in supporting feedback, not all data can be easily mapped onto a feature space and this is especially true for data with complex structures. The increasingly common usage of graph- and tree-structured data, such as data-centric XML collections, poses new challenges in data retrieval [12] and feedback processing. A recent Dagstuhl report [4] on "Ranked XML Querying" highlights the significance of the "too-many-answers" problem to keyword-based XML retrieval. Observing that the traditional IR-based feedback processes cannot be directly applied to improve semi-structured data or XML retrieval, recent works, including [35, 20, 21, 30, 31, 29, 15], have tried to explore the role of relevance feedback for refining queries over XML documents. In [20, 21], Pan *et al.* decomposed the query into its *elementary* sub-queries. Given user feedback, each elementary sub-query is expanded with a weight showing its relative importance in the overall query. The total score of an XML path (or XML document) is then computed from the elementary scores of the sub-queries. Schenkel *et al.* [30, 31] also consider path-based feedback to expand queries. Given a result path, (e.g., $/article/fm/cr/p$), their method extracts path fragments (e.g., prefix $/article/\#$, infix $\#/fm/\#$, etc.). Then, these path fragments are used to identify significant tags or tag-pairs (of parent/child). In this approach, structural expansion is limited to only the tag-pairs. [15] and [29] both propose to enrich the initial query by adding significant terms and common sub-paths shared by most relevant result paths. Existing works in this area [20, 21, 30, 31, 29, 15] focus on the use of feedback for document-centric XML (e.g., INEX [1] efforts), but not data-centric XML collections. Also, different from the above works which only tackle soft-feedback processes, *AXP* provides an integrated both hard- and soft-feedback. In addition, *AXP* considers both positive and negative feedback.

In *AXP*, soft feedback is used for ranking data features from potentially most relevant to the user to the least relevant ones. Once the data features are associated with relevance scores and an appropriate scoring function is developed to rank query results, various ranked query processing techniques (such as, nearest neighbors [26], top-K ranked joins [32, 23, 17], and skylines [5, 16]) can be adopted to identify the set of results to be shown to the user. In [23] and [6], for example, we focused on ranked path and tree query processing on weighted graphs, respectively. Our focus, however, is not on the ranked query processing, but on learning the importance of structural features based on user feedback.

Also, we highlight that learning preferences for structured data (where the various structural features are structurally dependent) is not the same as the more traditional problem of learning preferences for independent attributes/dimensions of vector data [28]. This structural dependence is our focus in Section 3.3.1.

2. PRELIMINARIES

In this section, we introduce the data model, the notions of query and answer on the data model, and the concept of user feedback.

2.1 Data and Query Models

In this paper (and in our evaluation) we focus on data-centric XML collections (Figure 2). As mentioned earlier, we also focus on path expressions of type, $P^{\{/,//,*\}}$. Such path expressions are composed of query steps, each consisting of an axis (parent/child "/" or ancestor/descendant "//") test and a label test. Given a document, T, and a path query, Q, the result to the query is an *em-*

bedding of the query nodes onto the nodes of the data tree in such a way that all label and structural predicates are satisfied. This definition of a result is more general than the result definition in XPath, where only the right most query node is returned as a match. Notice that there can be zero, one, or more than one results to a given query in a given document. Reconsider for example the path query $//title$ and the tree document in Figure 2: the query has six matches, each corresponding to a different path from the root to a node labeled with "title". Note that the query model can accommodate predicates applied on elements (e.g., "get project titles containing 'Harp'"). However, we omit the corresponding details.

2.2 Structural Features of a Label Path

We define the structural features of a given label-path as follows:

DEFINITION 2.1 (FEATURES OF A LABEL PATH). *The set, $\mathcal{F}(P)$, of features of a label-path $P = l_0 \cdot \cdots \cdot l_j$ is the maximal set of path queries such that $\mathcal{F}(P) = \{Q_i \mid Q_i(P) = true\}$, where F_i is in the type of $P^{\{/,//\}}$.*

A feature of a label-path can be as short as a 1-label like $//l_1$, or as long as a whole result path.

EXAMPLE 2.1. *Given a label-path $P = repository \cdot project \cdot title$, where "repository" is the label of the root node, $\mathcal{F}(P)$ includes single label features, $/project$, $//project$, $//title$, as well as 2-label features, $/project/title$, $/project//title$, $//project/title$, and $//project//title$ (we ignore the root label).*

2.3 Feedback Model

As we mentioned in the Introduction (Section 1), *AXP* handles different forms of feedback:

DEFINITION 2.2 (USER FEEDBACK OVER A RESULT). *Let Q be a query over a document T, and r be a result for Q against T. Let $P(r)$ be a data path defined by r. The user feedback over r is a 4-tuple $F = \langle F_h^+, F_h^-, F_s^+, F_s^- \rangle$:*

- *F_h^+ is a set of features of $P(r)$, which the user marks as positive* hard *feedback: the intended meaning is that any future results SHOULD contain all the features listed in F_h^+.*
- *F_h^- is a set of features of $P(r)$, which the user marks as negative* hard *feedback: the intended meaning is that any future result SHOULD NOT contain any features in F_h^-.*
- *F_s^+ is a set of features of $P(r)$, which the user marks as positive* soft *feedback: the intended meaning is that the user LIKEs those results that contain these features better than others that do not contain these features.*
- *F_s^- is a set of features of $P(r)$, which the user marks as negative* soft *feedback: the intended meaning is that the user DISLIKEs any result containing the features included in F_s^-.*

EXAMPLE 2.2. *Consider again the path query $//title$ over the tree document in Figure 2, when presented with the set of results the user might want to express that (a) she is only interested in paths that contain the tag "contributor", while (b) she does not want to see any projects in the results, while she would (c) prefer to see datasets. Her feedback can be represented as the 4-tuple $\langle \{//contributor\}, \{//project\}, \{//dataset\}, \{\} \rangle$.*

Note that it is on the top K results that a user provides her feature feedback. In other words, these K results define the boundaries of the user's current exploration space. The user can express her preferences in terms of features.

3. SOFT FEEDBACK PROCESS

The soft feedback is used for ranking the alternative results so that user can explore relevant parts of the data more readily.

3.1 Structural Feature Elements

AXP splits any complex feedback statement into its constituent 1- and 2-label feature elements for further analysis. In particular, given a feedback statement $l_0, \cdots, l_x$, the 1-label features are $\{//l_i\}$ where $0 \leq i \leq x$, and 2-label features are $\{//l_i/l_{i+1}\}$ for $0 \leq i < x$ and $\{//l_i//l_j\}$ for $0 \leq i, j < x$ and $j - i > 1$. These features capture the two key structural building blocks of label paths: 1-label features of the form "$//l_n$" capture existence of nodes with a certain label and 2-label features of the form "$//l_n/l_m$", and "$//l_n//l_m$" capture the parent/child and ancestor/descendant relationships between these labels (Figure 4). Note that, while they are simpler than more complex structural features, even these are not fully independent from each other; e.g., "$//l_n/l_m$" implies "$//l_n//l_m$". If a set of features are not mutually independent (e.g., one path feature includes the other one as a sub-feature), this might negatively affect the relevance feedback process: feature independence assumption is often necessary for simplifying the probabilistic formulas [27]. In Section 3.3.1, we describe how *AXP* addresses inter-dependences between the features.

3.2 Structural Feature Significance

Path features used during the soft feedback process need to distinguish those paths that are aligned with the user's relevance judgment from those that are against it. Thus, features can be weighted based on how well they contribute to this relevance judgment.

DEFINITION 3.1 (FEATURE SIGNIFICANCE). *Let R_K be the set of K results presented to the user for feedback, and F_s be the set of 1- or 2-label features extracted from the user's (LIKE or DISLIKE) relevance feedback statements. The significance of feature $f \in F_s$, relative to the result set, R_K denoted as $p(f|R_K, F_s)$ is defined as $p(f|R_K, F_s) = \frac{|R_K(f)|}{|R_K(F_s)|}$, where $R_K(f)$ is the set of results that satisfy the feature f, whereas $R_K(F_s)$ is the set of results that satisfy any of the feedback statements in F_s.*

Note that the definition of feature significance is probabilistic in nature: it measures the likelihood of a result that is relevant to any of the feedback containing the given feature.

EXAMPLE 3.1. *Let us consider two results to query, '$//title$': $r_1 = repository \cdot coding\_sheet \cdot title$, and $r_2 = repository \cdot coding\_sheet \cdot data\_file \cdot title$. Let us assume that the user marks '$/coding\_sheet$' in r_1 and '$//coding\_sheet/data\_file$' in r_2 as feedback. Thus, $F_s = \{/coding\_sheet, //coding\_sheet/data\_file\}$ and $R_K = R_K(F_s) = \{r_1, r_2\}$. The 1-label feature '$f_1 = /coding\_sheet$' appears in both r_1 and r_2; thus, $p(f_1|R_K, F_s) = 1.0$. The 2-label feature '$f_2 = //coding\_sheet/data\_file$' appears only in r_2; thus, $p(f_2|R_K, F_s) = 0.5$.*

3.3 Combined Result Score

We define the score of a result in terms of the significances of its features as they relate to the user feedback: If a result contains features that are significant relative to the positive feedback, intuitively, its score needs to be high; on the other hand, if the result path contains features that are significant relative to the negative feedback, its overall score needs to be low. Let $\mathcal{F}_s^+$ and $\mathcal{F}_s^-$ represent positive and negative feedback, respectively.

DEFINITION 3.2 (RESULT SCORE). *The combined score of a result r_i is defined as follows:*

$$score(r_i) = \frac{p(Like|r_i, \mathcal{F}_s^+) \cdot p(Like|r_i, \mathcal{F}_s^-)}{p(Dislike|r_i, \mathcal{F}_s^-) \cdot p(Dislike|r_i, \mathcal{F}_s^+)}.$$

Here, $p(Like|r_i, \mathcal{F}_s^+)$ denotes the probability that result r_i will be judged relevant by the system based on the feedback in $\mathcal{F}_s^+$. Let

R_K be the set of K results shown to the user for feedback and R_K^+ denote those that satisfy at least one of the positive feedback statements in $\mathcal{F}_s^+$. Then,

$$p(Like|r_i, \mathcal{F}_s^+) = \frac{p(r_i|R_K^+, \mathcal{F}_s^+)p(R_K^+, \mathcal{F}_s^+)}{p(r_i, \mathcal{F}_s^+)}.$$

On the other hand, $p(Dislike|r_i, \mathcal{F}_s^+)$ denotes the probability that result r_i is judged *not relevant* based on $\mathcal{F}_s^+$. Let $R_K^{\neg(+)} = R_K - R_K^+$ denote those results that do not satisfy any of the positive feedback statements in $\mathcal{F}_s^+$. Then,

$$p(Dislike|r_i, \mathcal{F}_s^+) = \frac{p(r_i|R_K^{\neg(+)}, \mathcal{F}_s^+)p(R_K^{\neg(+)}, \mathcal{F}_s^+)}{p(r_i, \mathcal{F}_s^+)}.$$

The terms, $p(Dislike|r_i, \mathcal{F}_s^-)$ and $p(Like|r_i, \mathcal{F}_s^-)$ are defined similarly based on $\mathcal{F}_s^-$. Note that, since they are computed based on different sets of feedback, $p(Dislike|r_i, \mathcal{F}_s^+)$ is not necessarily equivalent to $p(Dislike|r_i, \mathcal{F}_s^-)$. Similarly, for $p(Like|r_i, \mathcal{F}_s^-)$ and $p(Like|r_i, \mathcal{F}_s^+)$. Given these, and using the Bayes' theorem, we can rewrite $score(r_i)$ as follows:

$$\frac{p(r_i|R_K^+, \mathcal{F}_s^+) \cdot p(r_i|R_K^{\neg(-)}, \mathcal{F}_s^-)}{p(r_i|R_K^-, \mathcal{F}_s^-) \cdot p(r_i|R_K^{\neg(+)}, \mathcal{F}_s^+)} \times \frac{p(R_K^+, \mathcal{F}_s^+) \cdot p(R_K^{\neg(-)}, \mathcal{F}_s^-)}{p(R_K^-, \mathcal{F}_s^-) \cdot p(R_K^{\neg(+)}, \mathcal{F}_s^+)}.$$

(1)

Since the term at the right is constant, we only need to compute the term on the left. Given a feature set F' and a corresponding $R' \subseteq R_K$, we need to compute $p(r_i|R', F')$. Let $F'(r_i)$ denote the set of corresponding features of the path r_i. Since r_i is the conjunction of all of its features, we can write $p(r_i|R', F')$ as $p(\bigwedge_{f_j \in F'(r_i)} f_j|R', F')$. If all the features of r_i were independent, we could easily rewrite this as

$$p(r_i|R', F') = \prod_{f_j \in F'(r_i)} p(f_j|R', F') = \prod_{f_j \in F'(r_i)} \frac{|R'(f_j)|}{|R'|};$$

however, the 1- or 2-label features of the form "$//l_n$", "$//l_n/l_m$", and "$//l_n//l_m$" are not always independent. For example, feature "$//l_n//l_m$" implies the feature "$//l_n$". Similarly, "$//l_n/l_m$" implies "$//l_n//l_m$". Thus, such dependencies between features have to be taken into account in assessing their contributions.

3.3.1 Feature Cover

Given two inter-dependent features of a path, we refer to the one that always implies the other as the more specific feature:

DEFINITION 3.3 (SPECIFICITY). *Given two features f_g, f_s, the feature f_s is more* specific *than f_g, (denoted as $f_s \ll f_g$) iff for any label path P, f_g is a feature of P if f_s is a feature of P.*

Whenever the feature $//project/title$ is in a path, the feature $//project//title$ is also satisfied; thus $//project/title$ is a more specific feature than $//project//title$. Given a set of features, its *feature cover* consists of the most specific features in the set:

DEFINITION 3.4 (FEATURE COVER). *A feature cover F_c corresponding to a feature set F is the set of all features in F such that $\forall f_i \in F_c, \nexists f_j \in F$ s.t. $f_j \ll f_i$.*

EXAMPLE 3.2. *Given the label path $P = repository \cdot project \cdot contributor \cdot title$ (where $_repository$ is the root), its complete feature set, $F(P)$, is*

$\{$ $//project, /project, //contributor, //title,$
$//project/contributor, //project//contributor,$
$/project/contributor, /project//contributor,$
$//project//title, /project//title,$
$//contributor//title, /contributor/title\}.$

The feature cover for this path, however, is much smaller:

$F(P) = \{$ $/project, //project/contributor,$
$//project//title, //contributor/title\}.$

THEOREM 3.1. *Let F be a set of features and F_c be the corresponding feature cover. Then, $p\left(\bigwedge_{f \in F} f\right) = p\left(\bigwedge_{f \in F_c} f\right)$.*
That is, the probability that any given path P contains all features in F is equal to the probability that the path contains all features in the cover, F_c. Let $p(F)$ be a shorthand for $p(\bigwedge_{f \in F} f)$; then we can prove the above theorem as follows:

PROOF SKETCH 3.1. *First of all, if $f_s \ll f_g$, then $p(f_s \wedge f_g) = p(f_s)$: from the Bayes' theorem, we can get: $p(f_s \wedge f_g) = p(f_g|f_s) \cdot p(f_s)$. Since $f_s \rightarrow f_g$, $p(f_g|f_s) = 1$. Thus, $p(f_s \wedge f_g) = p(f_g|f_s) \cdot p(f_s) = p(f_s)$. Let $f_i \in F$ be a feature such that there is a more specific feature in F. Using the Bayes' theorem, we can see that*

$$p(F) = p(F|F_{/\{f_i\}}) \cdot p(F_{/\{f_i\}}),$$

where $F_{/\{f_i\}} = F - \{f_i\}$. But, since $F_{/\{f_i\}}$ is collectively more specific than F, $p(F|F_{/\{f_i\}}) = 1$ and, consequently,

$$p(F) = 1 \cdot p(F_{/\{f_i\}}) = p(F_{/\{f_i\}}).$$

This process can be applied to all features in F which have a more specific feature, eliminating them from the set. Finally, all the features that are in the set are those that are the most specific ones; i.e, $p(F) = p(F_c)$.

3.3.2 Computing $p(r_i|R', F')$ based on the Cover

The following corollary provides us with a way to deal with inter-dependent features of a path when computing $p(r_i|R', F')$.

COROLLARY 3.1. *Let r_i be a result path, $F'(r_i)$ be the set of features of r_i, and $F_c'(r_i)$ be the corresponding feature cover. Then,*

$$p(r_i|R', F') = p(F'(r_i)|R', F') = p(F_c'(r_i)|R', F')$$
$$= \prod_{f \in F_c'(r_i)} p(f|R', F') = \prod_{f \in F_c'(r_i)} \frac{|R'(f)|}{|R'|}$$

In other words, the features in the cover are sufficient when computing the score of a path.

However, one more challenge remains in computing $p(r_i|R', F')$ based on the feature cover. While the above corollary is sufficient for computing $p(r_i|R', F')$ for paths that consist solely of features that occur in R', if any of the features does not occur in R', then we have $p(r_i|R', F') = 0$. This means that if any one of r_i's features does not occur in R', it is not distinguishable from other such paths, irrespective of how significant all its other features are. Naturally, this is not appropriate when ranking paths that contain a feature for which no user feedback is applicable. To prevent 0-ing out of $p(r_i|R', F')$ for such paths, a lower-bound, α, is used: i.e., if a feature f does not occur in R' or when $R' = \emptyset$, then $p(f|R', F') = \alpha$. Here, α is a positive real number, such that $\alpha \ll 1/|R'|$; i.e., smaller than the $p(f|R')$ value of any single feature. In the experiments we use $\alpha = 1/|R'|^2$.

3.4 Complexity

Let the number of results presented to the user for feedback be K and let the average length of the result path be l. Let the size of the feedback cover be $|F_c|$. Thus, computing feature scores takes $O(Kl|F_c|)$ time, where each result is scanned and the features are counted. Once the features are counted, all results obtained in the previous iteration have to be re-scored. If the total number of results is $|R|$, then this is done in $O(|R||F_c|)$ time.

4. HARD FEEDBACK

As described in Sections 1.1 and 2.3, hard feedback is used for refining the user's initial imprecise query based on structural features that the user expects to see in the results as well as with features that the user expects not to see in the results. Let $\mathcal{F}_h^+$ and $\mathcal{F}_h^-$ be the sets of positive and negative hard feedback features. Intuitively, the initial query Q can be rewritten into the query $Q(\mathcal{F}_h^+, \mathcal{F}_h^-)$ as

$$Q(\mathcal{F}_h^+, \mathcal{F}_h^-) = Q \wedge \left(\bigwedge_{Q_i \in \mathcal{F}_h^+} Q_i \right) \wedge \neg \left(\bigvee_{Q_i \in \mathcal{F}_h^-} Q_i \right). \quad (2)$$

One way to process this refined query would be to simply go over the initial results to the query Q and eliminate those that satisfy any feature in $\mathcal{F}_h^-$ or do not satisfy all features in $\mathcal{F}_h^+$. Finite automata (FA) have been shown to be effective in message filtering tasks [9, 13, 19], where a set of incoming messages are filtered against a set of pre-registered filter conditions. For example, Diao *et al.* use nondeterministic finite automata (NFA) to filter incoming XML messages based on pre-specified filtering criteria [9]. In [13], Green *et al.* propose to use deterministic finite automata (DFA) to represent the filtering conditions in an integrated manner. The difference between these two approaches is that in the nondeterministic finite automata based schemes, the initial filter representation is easy to construct and small, while the run-time state space can be arbitrarily large, whereas in DFA-based schemes, the initial filter representation itself may be large, but the system has more control over how to expand and explore the state space needed for filtering. The lazy-DFA based schemes, for example, leverage this to reduce the filtering costs significantly [14, 13, 8].

The existing DFA- and lazy-DFA based message filtering schemes however cannot be directly applied to query refinement: in message filtering, filter conditions are combined using a disjunction ("$\vee$") operator (i.e., a path is selected if it satisfies any of the conditions), whereas $Q(\mathcal{F}_h^+, \mathcal{F}_h^-)$ consists of conjunctions ("$\wedge$") of path expressions. Moreover, the naive use of path filtering strategy is likely to be unnecessarily expensive: the inherently iterative nature of the feedback process gives rise to optimization opportunities that a naive filtering scheme may not be able to leverage effectively. As in the message filtering schemes, *AXP* relies on a finite automata-based representation which has the right expressive power to express $Q(\mathcal{F}_h^+, \mathcal{F}_h^-)$. However, the construction and execution of the *AXP* DFA differ from the DFAs for message filtering.

4.1 DFA-based Path Query Refinement

A deterministic finite automaton (DFA) is a 5-tuple $D = (S, \Sigma, \delta, s_0, S_f)$, where S is the set of all states, Σ is the alphabet of symbols, δ is the set of all transitions between the states, $s_0 \in S$ is the starting state, and S_f is the set of accepting states. A string, u, is accepted by a deterministic finite automaton if, starting from state s_0 and following the state transitions implied by δ for each symbol of u, the automaton reaches an accepting state in S_f after the final symbol of u is processed. It is known that any regular language can be recognized by a DFA. Moreover, regular languages are known to be closed under union, intersection, difference, and complement operations. Let $L(Q)$ denote the regular language expressed by the path expression Q (since path expressions are a subset of the regular expressions which describe regular languages, such a regular language exists). Let $D(Q) = D(L(Q))$ denote the DFA that recognizes $L(Q)$. Then, one can obtain a DFA, $D(Q(\mathcal{F}_h^+, \mathcal{F}_h^-))$, recognizing $Q(\mathcal{F}_h^+, \mathcal{F}_h^-)$ as
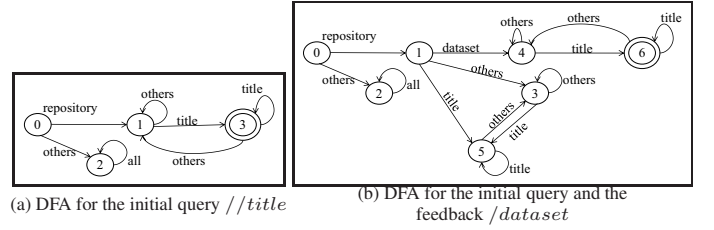

(a) DFA for the initial query $//title$ (b) DFA for the initial query and the feedback $/dataset$

Figure 5: DFA samples to represent user feedback; the path query is implicitly with respect to the data root

$$D(L(Q)) \otimes \left(\bigotimes_{Q_i \in \mathcal{F}_h^+} D(L(Q_i)) \right) \otimes \left(\bigotimes_{Q_i \in \mathcal{F}_h^-} D(\overline{L(Q_i)}) \right), \quad (3)$$

where $\otimes$ denotes the DFA intersection operation, and $\overline{L(Q_i)}$ is the complement of the language $L(Q_i)$.

THEOREM 4.1. *The language that $D(Q(\mathcal{F}_h^+, \mathcal{F}_h^-))$ recognizes is the set of results to the query Q which also satisfy the "SHOULD" feedback, but do not satisfy any of the "SHOULD-NOT" feedback.*

PROOF SKETCH 4.1. *The proof relies on using Equation 2, closure properties of the regular languages, and the relationship between regular languages and finite automata.*

The above theorem does not specify how to construct an efficient automaton to represent the feedback. A well-known result in automata theory is that for any regular language, there is a unique DFA having the smallest number of states that accepts it and this DFA can be constructed by clustering similar states and removing unreachable ones. The time complexity of the DFA minimization task is polynomial in the size of the input DFA. In *AXP*, we construct a *base* DFA directly from Equation 3.

EXAMPLE 4.1. *Let Q be $//title$ and let the user positive feedback be of the form $/dataset$ (i.e., "the child of the root must be 'dataset' "). Figures 5(a) and 5(b) show the DFAs corresponding to the original query and the refined query, respectively.*

4.1.1 DFA-based Query Refinement

DFA is constructed for reflecting the hard feedback on the query results. After *AXP* constructs a DFA, the algorithm runs the XML data/document over the DFA to get results incorporating feedback. The algorithm assumes that the document nodes are visited in pre-order. To leverage the tree-structured nature of the input to minimize the number of state transitions, the algorithm maintains a $Stack$ to keep track of the DFA states. The algorithm also maintains a $contextPath$ to remember the visited document element sequences so that result paths can be extracted. When seeing a start tag, $\langle e \rangle$, the process finds the next state, ns, and updates the context path and the running stack appropriately. If this state corresponds to an accepting state, the algorithm extracts and reports the result path. When seeing an end tag, $\langle /e \rangle$, the top elements of context path and the running stack are popped out. Since the refined DFA of the query incorporates feedback in each iteration, its states and transitions may grow with the number of feedback iterations. We improve this by getting rid of useless states and transitions.

4.1.2 Abridged Data Tree

The above discussion and the algorithm assume that in each iteration, *AXP* applies the DFA on the initial document, D. This is not necessary and would involve significant amount of redundant

work. In fact, in iteration j, *AXP* could run the feedback refined DFA only over the part of the tree that contributed to the results in iteration $j-1$. We call this the *abridged* tree, $D' \subseteq D$.

The abridged tree, D', is constructed by combining distinct results. On seeing a new result, the current tree, D' is scanned from the root in a top-down manner to match the labels in this result path. A new branch is created in the tree when a non-matching tag is met. The construction time of the abridged tree (including the disk I/O) is linear in the size of the number of the results. This is also confirmed by experiments (Section 5).

4.1.3 Complexity

Each iteration, j, of the feedback process involves (a) computation of the refined DFA and (b) processing of the input data on this DFA. Since the DFA state transition is $O(1)$ for each data element observed in the abridged tree, the time complexity of processing the input data on the refined DFA is $O(|D_j|)$ where $|D_j|$ is the size of the abridged tree input to the iteration j.

The worst case time complexity of combining two DFAs, namely DFA_1 and DFA_2, using DFA intersection and/or union operations, is $O(|DFA_1| \cdot |DFA_2| \cdot |\Sigma|)$ where Σ is a set of the distinct labels, and $|DFA_1|$ and $|DFA_2|$ are the numbers of states in these DFAs. Thus, if applied naively, the worst case size of the refined DFA would grow exponentially with the number of iterations. The DFA compression process described in Section 4.1.1 prevents this exponential growth by eliminating after each iteration, j, those states that have not been visited in that iteration. Since the number of states that have been visited at iteration j is limited by the size, $|D_j|$, of the input abridged tree, the DFA construction cost for iteration j is bounded by $O(|D_j| \cdot |DFA_{feedback,j}| \cdot |\Sigma|)$, where $DFA_{feedback,j}$ is the DFA needed to express the feedback collected at iteration j.

4.2 Lazy-DFA Support

The main idea behind the lazy-DFA approach is that not all states of the DFA will be needed in run-time; thus, enumerating all the states of the underlying DFA can potentially waste significant amount of time and other resources [14, 13, 8]. Non-deterministic finite automata (NFA) solves this problem by encoding the automaton non-deterministically so that exponentially many states of the DFA can be combined into one. These states are unfolded in runtime as they are needed. In the lazy-DFA schemes, the query criteria are first captured using an NFA. During query processing, however, this is converted into a DFA on an on-demand basis, as particular states and transitions are needed.

Most existing works on lazy-DFA creation target the messaging filtering problem where multiple trigger criteria are combined using disjunctions [8]: if some document/path satisfies any of the given trigger query criteria, this document/path needs to be reported as a result. In contrast, the query refinement process discussed in this paper also requires *conjunctions* of the user's positive hard feedback statements: i.e., a path needs to be maintained only if it satisfies all positive feedback criteria. While this means that the acceptance criterion is more strict, verifying this acceptance criterion using a lazy-DFA approach may be more costly. This is because, when the traditional disjunctive lazy-DFA schemes [14, 13, 8] combine the NFA of the individual query statements into a single NFA, any of the original acceptance states can act as an acceptance state for the combined NFA. On the other hand, in the conjunctive case, the acceptance states of the original NFA are not individually sufficient: to accept a path, all the original NFA acceptance states should be reached simultaneously.

Since, according to Equation 2, the positive feedback and the

---

Function LazyDFA $(Q, D', \mathcal{F}_h^+, \mathcal{F}_h^-)$

- Q is the initial query, D' is the abridged document tree from the previous iteration, and $\mathcal{F}_h^+, \mathcal{F}_h^-$ are the feedback statements
- *Output*: R search results with context paths.

1. $NFAtemplate^+$ = combine the NFA-fragments of Q and $\mathcal{F}_h^+$;
   $NFAtemplate^-$ = combine the NFA-fragments of $\mathcal{F}_h^-$;
   LazyDFA$^+$ = initLazyDFAplus $(NFAtemplate^+)$;
   LazyDFA$^-$ = initLazyDFAminus $(NFAtemplate^-)$;
   map^+ = $\emptyset$; /* Correspondence between LazyDFA$^+$ states and $NFAtemplate^+$ state-sets*/
   map^- = $\emptyset$; /* Correspondence between LazyDFA$^-$ states and $NFAtemplate^-$ state-sets*/

   /* Process D' against the LazyDFA*/

2. R = null;
3. Treat D' as a sequence, E, of tag events of the form $\langle e1 \rangle \langle e2 \rangle ... \langle /e2 \rangle \langle /e1 \rangle$;
4. contextPath = null; /*Remember the path from the root to the current element*/
5. $Stack^+$ = null; $Stack^-$ = null;
   /* Remember the states of the LazyDFA$^+$ and LazyDFA$^-$ */
6. **while** ($e \in E$ is not the end of E)

   (a) if (e is start tag)

      i. **if** no LazyDFA$^-$ state is reachable from e, update LazyDFA$^-$ and map^- ;

      ii. **if** a terminal state in LazyDFA$^-$ can be reached from e skip all the descendent elements of e; **goto** Step 6;

      iii. $contextPath$.push(e);

      iv. Process e through LazyDFA$^+$

         A. $S_{dfa}^+ = Stack^+.top()$; /* top LazyDFA$^+$ state */
            $S_{dfa}^+\_next$ = nextLazyDFA$^+$State(LazyDFA$^+$, $NFAtemplate^+, map^+, S_{dfa}^+, e$);

         B. $Stack^+$.push($S_{dfa}^+\_next$);

         C. **if** ($S_{dfa}^+\_next$ is a terminal state), extract result from $contextPath$ and put it to R;

   (b) else
      $contextPath$.pop(); $Stack^+$.pop(); $Stack^-$.pop();
7. return R;

(a) LazyDFA algorithm

Function nextLazyDFA$^+$State (LazyDFA$^+$, $NFAtemplate^+, map^+, S_{dfa}, e$)

- Return: the LazyDFA$^+$ state reachable from S_{dfa} through e.
- Procedure:

1. $S_{dfa}' =$ LazyDFA$^+$.nextState(S_{dfa}, e);
2. if (S_{dfa}' is empty) /* this state need to be generated */

   (a) Get the set of $NFAtemplate^+$ states S_{nfa} corresponding to S_{dfa} from map^+;

   (b) Get the set of $NFAtemplate^+$ states S_{nfa}' reachable from S_{nfa} through e;

   (c) Create a new LazyDFA$^+$ state S_{dfa}' corresponding to S_{nfa}';

   (d) Update LazyDFA$^+$ by adding S_{dfa}' and related transitions;

   (e) Update map^+ by adding the correspondence between S_{dfa}' and S_{nfa}';

3. Return S_{dfa}';

(b) LazyDFA$^+$ runtime generation

Figure 6: LazyDFA **based feedback processing**

initial query are to be combined using conjunctions, whereas the negative feedback can be combined using disjunctions (as in the message filtering schemes), *AXP* separates the positive feedback and negative feedback into two different sets and creates a lazy-DFA for each (LazyDFA$^+$ and LazyDFA$^-$, respectively). Figure 6 shows the combined LazyDFA algorithm. In what follows, we describe the LazyDFA strategy for managing hard feedback.

4.2.1 Positive `LazyDFA` (`LazyDFA`$^+$)

To begin with, the path query and each of the positive feedback statements are converted into NFA-fragments. Let us consider a query or a positive feedback of the form $op_1\ vq_1\ op_2\ vq_2\ \cdots\ op_n\ vq_n$. As in [9], the algorithm,

1. creates a starting state for the NFA-fragment,

2. creates a transition, $Tr(vq_i)$, labeled vq_i for each vq_i (including $*$) and the ancestor/descendant operator "//",

3. creates a state for each vq_{i-1}/vq_i, such that $Tr(vq_{i-1}).end = Tr(vq_i).start$,

4. creates a state for each $vq_{i-1}//vq_i$, such that $Tr(vq_{i-1}).end = Tr(//).start$ and $Tr(//).end = Tr(vq_i).start$.

All NFA-fragments are combined into a single NFA-template as in [9] (Step 1 in Figure 6(a)). Figure 7(a) shows the NFA-template created for an initial query "//title" and feedback "/dataset".

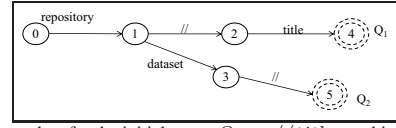One major difference of this process from [9] is that

- if the input is the original query, then the transition corresponding to the final label vq_n points to a *partial* terminal state (i.e., the state is not sufficient by itself to accept a path).

- if the input is a feedback statement, then a new partial terminal state is created and a "//" transition from the last state of this feedback statement to the new *partial* terminal state is added to the NFA-fragment. This extra transition is needed because feedback statements can be satisfied by sub-paths of a result path: for example, a result path, $a \cdot b \cdot c$, can satisfy both the query "//c" and the feedback "//b".

Note that the NFA-template created above is not a standard NFA. First of all, the "//" transitions in the NFA-templates do not correspond to real NFA transitions. More importantly, as described above, due to the conjunctive integration semantics, the *partial* terminal states of the NFA fragments are not real final states. For the automaton to be in an acceptance state, all its partial terminal states have to be reached together. Thus, to distinguish from a real terminal state, as in Figure 7, we use a double dotted circle to represent these partial terminal states.

The NFA-template is used by the `LazyDFA`$^+$ algorithm as a blueprint to create the necessary DFA states in run-time (Steps 2 to 6 in Figure 6(a)). In general, a `LazyDFA`$^+$ state corresponds to several NFA-template states. During the processing of the input data tree, when the `LazyDFA`$^+$ needs to move to a new state, if the required `LazyDFA`$^+$ state has already been created, then this state is used. Otherwise, a new `LazyDFA`$^+$ state needs to be created. This new state is inserted into an index structure, map^+, which keeps track of the correspondences between created `LazyDFA`$^+$ states and the associated NFA-template state-set. Whether it is reused or newly created, the state is also inserted into a stack to keep track of the states visited for the current data branch.

When an input symbol necessitates a transition into a not-yet-created `LazyDFA`$^+$ state, all NFA-template states *reachable* using this symbol are combined into a new `LazyDFA`$^+$ state. Let S_{nfa} be the set of NFA-template states corresponding to the current `LazyDFA`$^+$ state. The set S'_{nfa} of NFA-template states reachable from S_{nfa} is computed as follows:
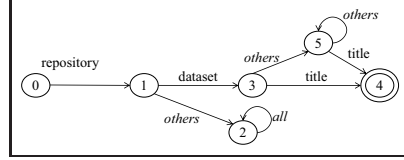
- **Rule 1 - All positive feedback need to be satisfied together**: If an NFA-template state has several outgoing transitions, they must be satisfied *together* to get a reachable NFA-template state.

(a) NFA-template for the initial query $Q_1 = //title$ combined with the feedback $Q_2 = /dataset$

`LazyDFA`$^+$ state	NFA-template state-set
0	$\{0\}$
1	$\{1\}$
2	$\emptyset$
3	$\{2^{//}, 3, \underline{5}^{//}\}$
4	$\{2^{//}, \underline{4}, \underline{5}^{//}\}$
5	$\{2^{//}, \underline{5}^{//}\}$

(b) `LazyDFA`$^+$ state and NFA state-set correspondence map (a state with superscript "//" means this state is reached through a "//" transition; the underlined states are terminal states)

(c) `LazyDFA`$^+$ after processing the document in Figure 2.

Figure 7: `LazyDFA`$^+$ **for positive hard feedback**

A `LazyDFA`$^+$ state is a terminal state only when it contains all the partial terminal states of the NFA-template. Thus, if, for no possible input string, the NFA-template state-set corresponding to a given `LazyDFA`$^+$ state cannot eventually reach to all partial terminal states of the NFA-template, then this `LazyDFA`$^+$ state can be marked as a failure state.

- **Rule 2 - "//" transitions on the NFA-template can correspond to empty sequences**: If an NFA-template state s_1 reaches another state s_2 through "//" transition, then any transition reaching s_1 automatically reaches s_2.

- **Rule 3 - "//" transitions on the NFA-template can correspond to sequences of variable length**: If an NFA-template state is reached through a "//" transition, it can be reached by any transitions. Therefore, the NFA-template state-set corresponding to a `LazyDFA`$^+$ state must include all past NFA-template "//" transitions used when processing the corresponding data branch.

Note that rule 1 is specific to the `LazyDFA`$^+$ due to its conjunctive nature, whereas rules 2 and 3 are similar to the treatment of "//" transitions in existing path filtering work, e.g. [9]. We illustrate these rules using the following example.

EXAMPLE 4.2. *We use the query in Example 4.1 to illustrate the* `LazyDFA`$^+$ *execution. The NFA-template is shown in Figure 7(a). In this figure, the NFA-template state 4 represents the partial terminal state for the initial query and state 5 denotes the partial terminal state for the feedback.*

Let us consider the document in Figure 2. Running this document over the NFA-template in Figure 7(a) results in the `LazyDFA`$^+$*, shown in Figure 7(c). The correspondence between the* `LazyDFA`$^+$ *states and the NFA-template state-sets are shown in Figure 7(b). In what follows, we go over some of the key steps to illustrate how the* `LazyDFA`$^+$ *mechanism works.*

Initially, `LazyDFA`$^+$ *has a single state 0, which corresponds to the NFA-template state-set $\{0\}$. When we see the root tag 'repository', we follow the only available NFA-template transition from $\{0\}$ to reach the NFA-template state-set $\{1\}$. Since this state-set did not exist before, a new* `LazyDFA`$^+$ *state, 1, corresponding to this state-set is created and recorded in map^+.*

From LazyDFA$^+$ *state* 1 *(i.e., the NFA-template state-set* {1}*),*

- *when we see either 'coding_sheet' or 'project' start tag, we can follow only one of the two available transitions on the corresponding NFA-template state-set,* {1}*. Since the semantics of the DFA merge operation is conjunctive, this means that the* LazyDFA$^+$ *cannot reach both of the partial acceptance states. This means that the current path cannot be recognized by the* LazyDFA$^+$ *(Rule 1). Thus, we generate a failure state,* 2*, for* LazyDFA$^+$*.*

- *if we see 'dataset' start tag, we could follow both of the available NFA-template branches to reach NFA-template states* 2 *and* 3*. In addition, since NFA-template state* 3 *and* 5 *are connected with a "//" transition, state* 5 *is also in the reachable set (Rule 2). Thus the set of reachable NFA-template states from the* LazyDFA$^+$ *state* 1 *is* {2, 3, 5}*. Thus, we generate a* LazyDFA$^+$ *state* 3 *to correspond to the NFA-template state-set* $\{2^{//}, 3, 5^{//}\}$ *and record it in* map^+*. The superscript "//" is used to remember that this NFA-template state is reached through a "//" transition: as long as we are on this data branch, we need to include this NFA-template state in newly created* LazyDFA$^+$ *states.*

When we are on the LazyDFA$^+$ *state* 3 *(i.e., the corresponding NFA-template state-set* $\{2^{//}, 3, 5^{//}\}$*), if we see the tag 'title', the NFA-template state*
- *4 is reachable from state* $2^{//}$
- $2^{//}$ *is reachable from itself* $2^{//}$ *(Rule 3), and*
- $5^{//}$ *is reachable from itself* $5^{//}$ *(Rule 3) and state* 3*.*

Thus, the reachable NFA-template state-set is $\{2^{//}, 4, 5^{//}\}$*. Since the state contains both NFA-template states* 4 *and* 5 *(i.e., the partial terminal states of the two NFA fragments), the corresponding* LazyDFA$^+$ *state,* 4*, is marked as a terminal state.*

4.2.2 *Negative* LazyDFA *(*LazyDFA$^-$*)*

Unlike for the LazyDFA$^+$, if a given path satisfies any of the negative hard feedback statements, we can decide that it *cannot* contain a result. Thus, the LazyDFA$^-$ has a disjunctive logic. Hence, LazyDFA$^-$ can be constructed using the standard lazy-DFA construction algorithms for message filtering systems [8].

When we see a new XML start tag $\langle e \rangle$ in the input, before passing it to the LazyDFA$^+$, LazyDFA first checks whether it reaches a terminal state in LazyDFA$^-$ (Step 6(a)ii in Figure 6(a)). If so, then the system skips this tag as well as all the descendant tags of $\langle e \rangle$ (i.e., the whole subtree rooted at e in D').

Since LazyDFA$^-$ can prune data sub-trees without further processing, in general, negative feedback is likely to reduce overall feedback processing time. In addition, especially if the overlaps between the result paths to the original query are large, then the negative feedback is likely to be more effective than the positive feedback in trimming the user's exploration space.

4.2.3 *Complexity*

The time complexity of the hard feedback management mostly depends on the time in creating the data structures, which include the NFA-templates, the LazyDFA, and the mapping table between the LazyDFA states and the NFA-template state-sets. Let $|\mathcal{F}_h|$ be the number of hard feedback statements. Let n be the maximum length of the query and the feedback statements. Given these, the size of the NFA-template is at most $O(|\mathcal{F}_h|n)$. As a consequence, the correspondence table size is bounded by $2n|\mathcal{F}_h|$. [13] has shown for disjunctive lazy-DFA that, given an XML document D, the lazy-DFA has at most $O(G)$ states, where G is the number of nodes in the data guide [11] of D. The sizes of both LazyDFA$^+$

and LazyDFA$^-$ are also similarly bounded (though due to its conjunctive nature, LazyDFA$^+$ explores a much smaller state space).

LazyDFA reduces the overall execution time by splitting the DFA into LazyDFA$^+$ and LazyDFA$^-$ and by letting LazyDFA$^-$ prune parts of the data tree without further processing. Thus, elements that are not guaranteed to lead to an acceptance state are not processed, saving processing time.

5. EXPERIMENTS

We implemented the *AXP* framework and the underlying techniques using Java. The experiments were performed on a Xeon(TM) 2.9GHz processor workstation with 2.00G RAM.

Data, queries, and the ground truth: The experiment results we report here are on the TreeBank data set [2]. We chose this dataset for its wide use in XML query processing and filtering literature [3]. This data set is a data-centric collection in that it contains the part-of-speech tags of Wall Street Journal articles, but not the textual content itself. Thus, it is a suitable set for evaluating feedback over data-centric XML. As an alternative, we have also considered the popular INEX data set [1] but could not use it because of its document-centric nature.

Except the tests on LazyDFA input and scalability (Figures 14 and 15), we used a portion of the Treebank XML tree, with $\sim 400K$ nodes and with a deep (the maximum depth is ~ 30 and the average depth is ~ 8) recursive structure. Thus, the data set also provides a suitable environment for testing the effect of feedback based exploration of path structured data. We randomly generated initial queries of the form "$//l_1//l_2$". Each data point presented in the feedback performance figures in this section is obtained by averaging the corresponding performance metric for multiple queries on the same data set.

Feedback. We have performed two types of feedback experiments: **(1)** In the first, larger, set of experiments, we used simulated user judgments (relying on an approach often applied in feedback research; e.g. [10] as well as INEX 2006 feedback track [33] [5]): for each query, one of its results is randomly chosen to represent its *ground truth*; i.e., the path that best represents user's preferences. Given $K(= 20)$ result paths, those features occurring in the ground truth are candidates to be provided as positive feedback; features conflicting with the ground truth are negative feedback candidates.

(2) The second set of experiments, reported in Section 5.3, were performed with the help of human subjects and they verify the results of the first set of experiments.

Efficiency and effectiveness measures. *For soft feedback* we are reporting execution times as the measure of efficiency. The effectiveness of the soft feedback process can be evaluated by observing whether the rankings obtained through the feedback process are correlated with the ground truth: given a result path with a system assigned score, s, and with m mismatches with the ground truth (each missed or extra feature counts as a mismatch), the score should be *negatively correlated* with the number of mismatches. The standard measure of correlation, *Pearson correlation coefficient* assumes that the relationship between the two variables is linear. In *AXP*, we do not expect the result score and the number of mismatches will necessarily have a (inversely) linear relationship. Thus, we report *Kendall-tau rank coefficient* which does not make the assumption of linearity.

---

[5]Personal communications with the authors confirmed that the feedbacks in [33] and their other works on the INEX data set are provided by simulated users based on the pre-specified ground truth as we do in this paper. A top-K result gets positive feedback iff it has been identified as relevant in this ground truth.
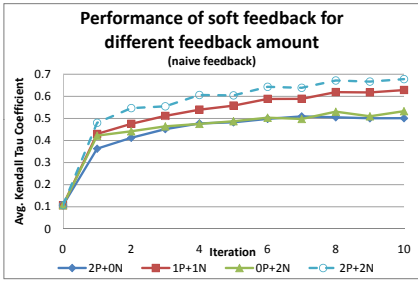
Figure 8: Effectiveness of soft feedback in matching ground truth; different curves denote different feedback amounts
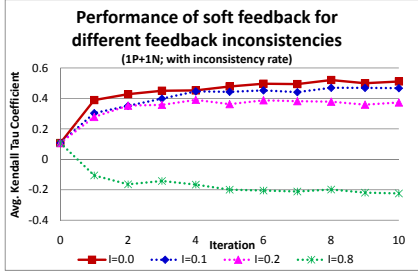

Figure 9: Effectiveness of soft feedback in matching ground truth; different curves denote different inconsistency rates

Figure 10: Scatter-plot depicting the processing times of soft feedback as a function of the number of results to be reranked

Figure 11: The speed with which hard feedback helps focus the user's exploration space; different curves correspond to different amounts of positive and negative feedback at each iteration

For *hard feedback* (DFA and `LazyDFA`), we report the number of states of the automata, number of transitions in the automata, and the execution time. Since the goal of the hard feedback process is to focus the exploration by eliminating the irrelevant paths from consideration, the effectiveness measure for the hard feedback is the number of paths in the result set. If the hard feedback process is effective, the number of paths matching the revised query should go to "1" (i.e., the ground truth) with only few iterations.

5.1 Evaluation of the Soft Feedback Process

Soft feedback results are collected using 10 different feedback sequences on 10 different queries; thus, each value presented in the results is the average of (10×10) 100 runs.

Effectiveness. The first set of soft feedback experiments shows the effectiveness of the positive and negative soft feedback in matching the rankings that would have been imposed by the ground truth (if the ground truth was available to the system in advance). As it can be seen in Figure 8, even when the feedback is selected naively without care at the discriminatory power of the features, both positive and negative feedback improve the quality of the ranking. Negative feedback is somewhat more effective than positive feedback; this is because there are more negative features in the data that can cause mismatches with the ground truth than the positive features that can be missed. Nevertheless, using both positive and negative feedback together is the most effective approach for improving rankings (compare the `1P+1N` curve (1 positive and 1 negative feedback per iteration) with the `2P+0N` (2 positive feedback, no negative feedback) and `0P+2N` curves (no positive feedback, 2 negative feedback)). Providing more feedback (`2P+2N` curve) consistently improves the quality of the ranking, indicating that *AXP* is effective in getting the most out of the feedback statements.

We also ran a set of experiments to study the effects of potential inconsistencies in user feedback. For this purpose, a portion of the feedback provided to the system were modified to be inconsistent with the ground truth: in other words, the user is providing counter-productive feedback to this search goal. As shown in Figure 9, when the inconsistency of the user feedback increases, this negatively affects the quality of the ranking with respect to the base
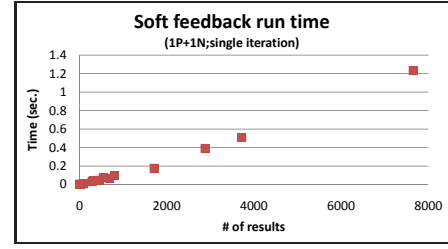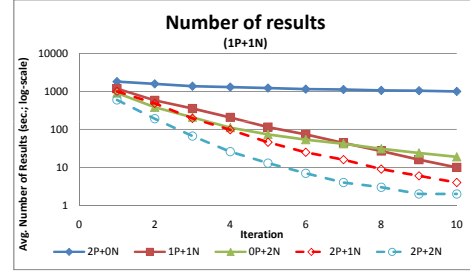
ground truth. In fact, when most of the feedback are inconsistent with the ground truth (see the "`I=0.8`" curve where the inconsistency rate in the feedback is 80%), the ranking coefficient becomes negative: in other words, if the user's feedback is inconsistent with what she really wishes to obtain (represented by the ground truth), the system learns, not what she wishes, but what she provides as feedback. This result is consistent with the expected behavior of feedback schemes and shows that our feedback process interprets the user's feedback correctly and can help the user when she is able to provide feedback that does not contain extreme inconsistency.

Execution time. As it can be seen in Figure 10, the execution time of the soft feedback generation process is linear in the number of results to be reranked. The cost of the statistical analysis phase for the soft feedback is negligible.

5.2 Evaluation of the Hard Feedback Process

Hard feedback results are also collected using 10 different feedback sequences for 10 random queries.

Effectiveness of hard feedback. The first set of experiments, presented in Figure 11, shows the effectiveness of the hard feedback in helping reduce the size of the user's exploration space. As it can be seen in this figure, both positive and negative feedback can help focus user's exploration. However, in general, negative feedback is more effective in helping eliminate irrelevant results. This can be shown by comparing the `2P+0N` curve with the `0P+2N` curve. Nevertheless, using both feedback together provides the best result (compare the `1P+1N` curve with `2P+0N` and `0P+2N` curves). Also, as expected, adding more feedback per iteration (`2P+1N` and `2P+2N` versus `1P+1N`) helps improve the speed with which the exploration space is trimmed.

This also impacts the execution time for processing feedback. Figure 12 shows the execution times for positive and negative feedback statements. Since it eliminates more results early on (thus reducing the abridged data tree size), negative feedback helps cut down the execution times. In general, however, both `LazyDFA`$^+$ for positive feedback and `LazyDFA`$^-$ for negative feedback work very efficiently and can support real-time interaction.

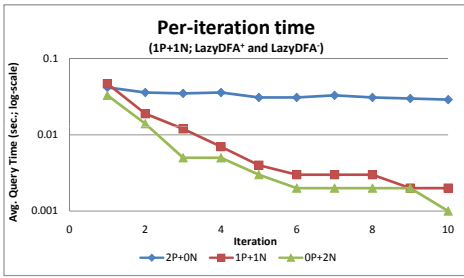The number of results obtained in each iteration also affects the

Figure 12: Per iteration runtime for `LazyDFA`$^+$ and `LazyDFA`$^-$
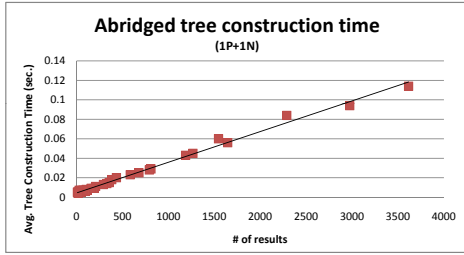

Figure 13: Scatter-plot depicting the construction time of abridged tree as a function of the number of distinct results

time taken to construct the abridged tree. As illustrated in Figure 13, as expected based on the discussion in Section 4.1.2, the abridged tree construction time is linear in the number of distinct results and is mostly negligible. Figure 14 illustrates how the abridged tree helps in the process. The execution time of the algorithm without the abridged tree is constant across multiple iterations because the input data for each iteration is the same (the whole dataset). When using the abridged tree, however, the running time of the algorithm drops with consecutive iterations: this is because the number of results (which are used to construct the abridged tree) decreases in each iteration as shown in Figure 14. Figure 15 further elaborates on the scalability of the `LazyDFA` method; `LazyDFA` is fast even for data sets as large as $\sim 750K$ nodes and the complexity increases linearly ($\sim 3\times$) in the number of nodes.

LazyDFA vs. DFA. As shown in Figure 16, the `LazyDFA` method, which generates the necessary states on demand, is more efficient than the pure DFA solution. This is because, the DFA method has to construct a potentially large automaton at each feedback iteration. Thus, while the `LazyDFA` method allows near real-time exploration, the naive DFA is not suitable for interactive use.

Space usage. The memory space used in each iteration is mainly for the FA states, transitions, and the results for this particular iteration. Abridged trees are written to the disk. Figure 17 shows the number of states and transitions created by the DFA and the `LazyDFA` schemes. As it can be seen here, the `LazyDFA` scheme significantly reduces the number of states and transitions generated (by creating these on demand, as necessary), thus providing both efficiency and scalability.

5.3 User Study Results

Using the same data set and 7 users who are not knowledgeable about the data, we also ran a set of user studies to verify the sim-

	No Feedback	1P+1N	2P+2N
Soft (Kendall-tau)	0.12	0.31	0.38
Hard (num results)	2290	1158	530

Table 1: Hard feedback helps users reduce the number of results; soft feedback improves the Kendall-tau coefficient
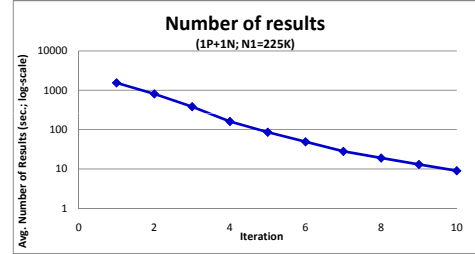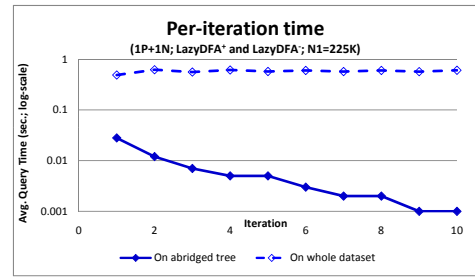

Figure 14: Per-iteration execution time and number of results; hard feedback processing with and without abridged tree
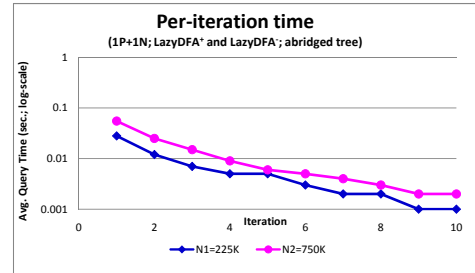

Figure 15: Per-iteration time with abridged tree; each curve corresponds to a different input data size

ulation results. We have provided these users 10 random queries and target results and asked them to try to reach the target result paths using hard and soft feedbacks. After each iteration, the user is presented 10 result paths to support feedback.

Results presented in Table 1 verifies the simulation results: soft feedback helps improve the Kendall-tau coefficient of the results presented to the user; similarly, users are able to reduce the number of results using hard feedback. As expected, more feedback helps users achieve higher improvements.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we presented an *adaptive, and exploratory, path retrieval framework, AXP*, to help the user explore the data-centric XML data, by providing positive and negative feedback that reflect her judgments and interests. Soft feedback are for ordering paths based on preference criteria, whereas hard feedback are used for eliminating irrelevant paths from consideration. We developed a probabilistic path ranking scheme to deal with soft feedback statements. We also introduced a novel `LazyDFA` data structure to process positive and negative feedback statements and leveraging the iterative nature of the feedback process. Experiments showed that feedback statements can be very effective in helping focus the exploration process. Similarly especially when used together, positive and negative soft feedback statements help improve the rankings of the results presented to the user during her exploration.

We focused on feedback processing over path query results. Our framework can be extended to more general twig queries for which the results are subtrees of the data. For twig queries, our soft feed-
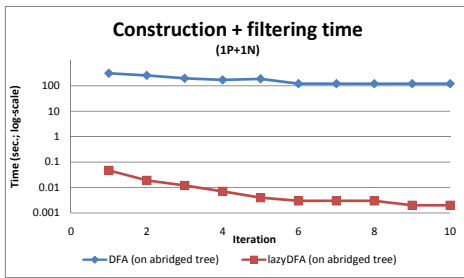
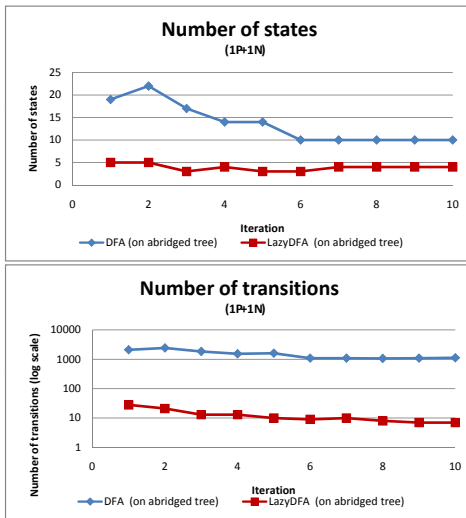**Figure 16: Running times of the DFA and `LazyDFA` methods**

Figure 17: Number of states and transitions

back process can be applied without any change since any complex feedback statement (even on twigs) can be split to its constituent 1- or 2-label features. Adapting the hard feedback process for twig query processing however is not as straightforward: our framework combines the initial query with positive feedback on its results to form the `LazyDFA`$^{+}$; on the other hand, the positive feedback on different branches in a result subtree cannot be combined into a single `LazyDFA`$^{+}$. This problem, however, can be dealt with by building a `LazyDFA`$^{+}$ for *each* twig branch and the positive feedback on that particular branch. The final result then will be calculated by combining the results obtained from all `LazyDFA`$^{+}$s in a post-processing step. We leave this as a future work. Note that, given a twig query, it may also be desirable to consider more complex features (e.g., feature in the form of twigs) in feedback assessment. This is also ongoing research.

7. REFERENCES

[1] Initiative for the evaluation of XML retrieval (INEX). http://www.inex.otago.ac.nz/.
[2] The penn treebank project, http://www.cis.upenn.edu/ treebank/.
[3] Treebank search tools in the tiger project, http://www.ims.uni-stuttgart.de/projekte/tiger/related/links.shtml#xml.
[4] S. Amer-Yahia, D. Hiemstra, T. Roelleke, D. Srivastava, and G. Weikum. Db&ir integration: Report on the dagstuhl seminar "ranked xml querying". *SIGMOD Record*, 37(3):46–49, 2008.
[5] S. Börzsönyi, D. Kossmann, and K. Stocker. The skyline operator. In *ICDE*, pages 421–430, 2001.
[6] K. S. Candan, H. Cao, Y. Qi, and M. L. Sapino. System support for exploration and expert feedback in resolving conflicts during integration of metadata. *VLDB J.*, 17(6):1407–1444, 2008.
[7] H. Cao, Y. Qi, K. S. Candan, and M. L. Sapino. Exploring path query

[8] D. Z. Chen and R. K. Wong. Optimizing the lazy dfa approach for xml stream processing. In *ADC: Proceedings of the 15th Australasian database conference*, pages 131–140, 2004.
[9] Y. Diao and M. J. Franklin. High-performance xml filtering: An overview of yfilter. *IEEE Data Eng. Bull.*, 26(1):41–48, 2003.
[10] M. Ferecatu, M. Crucianu, and N. Boujemaa. Improving performance of interactive categorization of images using relevance feedback. In *ICIP (1)*, pages 1197–1200, 2005.
[11] R. Goldman and J. Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In *VLDB*, 1997.
[12] G. Gou and R. Chirkova. Efficiently querying large xml data repositories: A survey. *TKDE*, 19(10):1381–1403, 2007.
[13] T. J. Green, A. Gupta, G. Miklau, M. Onizuka, and D. Suciu. Processing xml streams with deterministic automata and stream indexes. *ACM Trans. Database Syst.*, 29(4):752–788, 2004.
[14] A. K. Gupta and D. Suciu. Stream processing of xpath queries with predicates. In *SIGMOD '03*, pages 419–430, 2003.
[15] L. Hlaoua, M. Boughanem, and K. Pinel-Sauvagnat. Combination of evidences in relevance feedback for xml retrieval. In *CIKM '07*.
[16] M. E. Khalefa, M. F. Mokbel, and J. J. Levandoski. Skyline query processing for incomplete data. In *ICDE*, pages 556–565, 2008.
[17] J. W. Kim and K. S. Candan. Skip-and-prune: Cosine-based top-k query processing for efficient context-sensitive document retrieval. In *SIGMOD*, 2009.
[18] W.-S. Li, K. S. Candan, K. Hirata, and Y. Hara. Supporting efficient multimedia database exploration. *VLDB J.*, 9(4):312–326, 2001.
[19] M. Moro, P. Bakalov, and V. Tsotras. Early profile pruning on xml-aware publish-subscribe systems. In *VLDB*, 2007.
[20] H. Pan. Relevance feedback in xml retrieval. In *EDBT Workshops*, pages 187–196, 2004.
[21] H. Pan, R. Schenkel, and G. Weikum. Fine-grained relevance feedback for xml retrieval. In *SIGIR '08*, pages 887–887, 2008.
[22] Y. Qi, K. S. Candan, and M. L. Sapino. Ficsr: feedback-based inconsistency resolution and query processing on misaligned data sources. In *SIGMOD '07*, pages 151–162, 2007.
[23] Y. Qi, K. S. Candan, and M. L. Sapino. Sum-max monotonic ranked joins for evaluating top-k twig queries on weighted data graphs. In *VLDB*, pages 507–518, 2007.
[24] Y. Qi, K. S. Candan, M. L. Sapino, and K. W. Kintigh. Integrating and querying taxonomies with quest in the presence of conflicts. In *SIGMOD Conference*, pages 1153–1155, 2007.
[25] J. Rocchio. *Relevance Feedback in Information Retrieval*, pages 313–323. 1971.
[26] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. In *SIGMOD Conference*, pages 71–79, 1995.
[27] I. Ruthven and M. Lalmas. A survey on the use of relevance feedback for information access systems. *Knowl. Eng. Rev.*, 18(2), 2003.
[28] G. Salton, A. Wong, and C. S. Yang. A vector space model for automatic indexing. *Commun. ACM*, 18(11):613–620, 1975.
[29] K. Sauvagnat, L. Hlaoua, and M. Boughanem. Xfirm at inex 2005: Ad-hoc and relevance feedback tracks. In *INEX*, pages 88–103, 2005.
[30] R. Schenkel and M. Theobald. Feedback-driven structural query expansion for ranked retrieval of xml data. In *EDBT*, 2006.
[31] R. Schenkel and M. Theobald. Structural feedback for keyword-based xml retrieval. In *ECIR*, pages 326–337, 2006.
[32] Y. Tao, V. Hristidis, D. Papadias, and Y. Papakonstantinou. Branch-and-bound processing of ranked queries. *Inf. Syst.*, 32(3):424–445, 2007.
[33] M. Theobald, A. Broschart, R. Schenkel, S. Solomon, and G. Weikum. Topx: Adhoc track and feedback task. In *5th International Workshop of the Initiative for the Evaluation of XML Retrieval, INEX 2006*.
[34] X. Wang, H. Fang, and C. Zhai. A study of methods for negative relevance feedback. In *SIGIR '08*, 2008.
[35] R. Weber. Using relevance feedback in xml retrieval. In *Intelligent Search on XML Data*, pages 133–143, 2003.
[36] C. T. Yu, W. S. Luk, and T. Y. Cheung. A statistical model for relevance feedback in information retrieval. *J. ACM*, 23(2), 1976.

results through relevance feedback. In *CIKM*, pages 1959–1962, 2009.