



Proceedings of the
Sixth WebDB Workshop

International Workshop on Web and Databases

San Diego, California
June 12-13, 2003

Editors: Vassilis Christophides and Juliana Freire

Foreword

This volume contains 17 papers that were presented at the Sixth International Workshop on the Web and Databases (WebDB), which was held June 11-12, 2003, in San Diego, California, in conjunction with the ACM SIGMOD Symposium on Principles of Database Systems (PODS), and the ACM Conference on Management of Data (SIGMOD), and the 2003 ACM Federated Computing Research Conference. The contributed papers were selected by the Program Committee out of 74 submissions. The papers present preliminary reports on continuing research. During the selection process, particular emphasis has been given to papers that promote novel research directions. In addition to the contributed papers, the WebDB 2003 program also included an invited talk by Joseph Hellerstein (UC Berkeley).

The organizers would like to thank the Program Committee for providing thorough valuations within a very short time; the Web Chairs, Vassilis Papadimos and Manos Pappagelis, for maintaining the Web site and putting together the proceedings; and the FCRC organizers for their help on local organization and other issues.

Program Committee

Bernd Amann, CNAM, Paris

Siheem Amer-Yahia, AT&T Research, USA

Michael Benedikt, Bell Labs, USA

Alin Deutsch, University of California at San Diego, USA

Mary Fernandez, AT&T Research, USA

Georg Gottlob, Vienna University of Science, Austria

Jayant Haritsa, Indian Institute of Science, India

Rick Hull, Bell Labs, USA

Lu Hong Jun, Hong Kong University of Science and Technology

Alberto Laender, Universidade Federal de Minas Gerais, Brazil

Laks Lakshmanan, University of British Columbia, Canada

Guido Moerkotte, University of Mannheim, Germany

David Maier, OGI School of Science and Engineering, USA

Giansalvatore Mecca, Universita' della Basilicata, Italy

Evaggelia Pitoura, University of Ioannina, Greece

Dimitris Plexousakis, University of Crete, Greece

Jayavel Shanmugasundaram, Cornell University, USA

Masatoshi Yoshikawa, Nagoya University, Japan

Organizers

Vassilis Christophides, ICS-FORTH & University of Crete, Greece

Juliana Freire, OGI School of Science & Engineering, OHSU, USA

Web Chairs

Vassilis Papadimos, OGI School of Science & Engineering, OHSU, USA

Manos Pappagelis, University of Crete, Greece

Invited Talk

The Marvelous Structure of Reality

Joseph Hellerstein, UC Berkeley

Web content is often described as "unstructured data" - an unfortunate phrase. In fact, web search engines work precisely by extracting and exploiting familiar structures in the content of the web. Viewed in this manner, the success of web search reinforces the notion that structure is ubiquitous. This perspective also highlights an important theme for further invention: seek out structure in naturally-occurring phenomena, and build systems to query that structure. Interesting variations on this theme abound, including systems to query physical phenomena (via sensors), systems to query systems (including themselves), and even systems to query ideas.

Joseph M. Hellerstein is an Associate Professor of Computer Science at the University of California, Berkeley. He is an Alfred P. Sloan Research Fellow, and a recipient of multiple awards, including NSF CAREER, NASA New Investigator, Okawa Foundation Fellowship, IBM's Best Paper in Computer Science, and ACM-SIGMOD's "Test of Time" award for his first published paper. In 1999, MIT's Technology Review named him one of the top 100 young technology innovators worldwide (TR100). Hellerstein's research focuses on data management and movement, including database systems, sensor networks, peer-to-peer and federated systems. He received his Ph.D. at the University of Wisconsin, Madison, a masters degree from UC Berkeley, and a bachelor's degree from Harvard College.

Contributed Papers

Semantic Web

- Exploiting Structure, Annotation, and Ontological Knowledge for Automatic Classification of XML Data** 1

Martin Theobald, Ralf Schenkel, and Gerhard Weikum

- Automatic annotation of data extracted from large Web sites** 7

Luigi Arlotta, Valter Crescenzi, Giansalvatore Mecca, and Paolo Merialdo

- Semantic Email: Adding Lightweight Data Manipulation Capabilities to the Email Habitat** 13

Oren Etzioni, Alon Halevy, Henry Levy, and Luke McDowell

Information Integration

- XML Interoperability** 19

Laks V. S. Lakshmanan, and Fereidoon Sadri

- Building Data Integration Systems: A Mass Collaboration Approach** 25

Robert McCann, AnHai Doan, Vanitha Varadaran, Alex Kramnik, and Chengxiang Zhai

- On the updatability of XML views over relational databases** 31

Vanessa Braganholo, Susan Davidson, and Carlos Heuser

XML Data Management and Query Processing

- Tree Automata to Verify XML Key Constraints** 37

Béatrice Bouchou, Mírian Halfeld-Ferrari-Alves, and Martin Musicante

- A Framework for Estimating XML Query Cardinality** 43

Carlo Sartiani

- Path-expression Queries over Multiversion XML Documents** 49

Zografoula Vagena, and Vassilis J. Tsotras

- TypEx: A Type Based Approach to XML Stream Querying** 55

George Russell, Mathias Neumuller, and Richard Connor

Peer-to-Peer Systems and Data Distribution

- A Comparison of Peer-to-Peer Search Methods** 61

Dimitrios Tsoumakos, and Nick Roussopoulos

- ODISSEA: A Peer-to-Peer Architecture for Scalable Web Search and Information Retrieval** 67

Torsten Suel, Chandan Mathur, Jo-wen Wu, Jiangong Zhang, Alex Delis, and Mehdi Kharrazi

On Distributing XML Repositories	<i>73</i>
<i>Jan-Marco Bremer, and Michael Gertz</i>	
Web Data Management and Services	
Index Structures for Querying the Deep Web	<i>79</i>
<i>Jian Qiu, Feng Shao, Misha Zatsman, and Jayavel Shanmugasundaram</i>	
Modeling Query-Based Access to Text Databases	<i>87</i>
<i>Eugene Agichtein, Panagiotis Ipeirotis, and Luis Gravano</i>	
Efficient Dissemination of Aggregate Data over the Wireless Web	<i>93</i>
<i>Mohamed Sharaf, Yannis Sismanis, Alexandros Labrinidis, Panos Chrysanthis, and Nick Roussopoulos</i>	
The Query Set Specification Language (QSSL)	<i>99</i>
<i>Michalis Petropoulos, Alin Deutsch, and Yannis Papakonstantinou</i>	

Exploiting Structure, Annotation, and Ontological Knowledge for Automatic Classification of XML Data

Martin Theobald
Saarland University
D-66041 Saarbruecken
Germany

mtheobald@cs.uni-sb.de

Ralf Schenkel
Saarland University
D-66041 Saarbruecken
Germany

schenkel@cs.uni-sb.de

Gerhard Weikum
Saarland University
D-66041 Saarbruecken
Germany

weikum@cs.uni-sb.de

ABSTRACT

This paper investigates how to automatically classify schemaless XML data into a user-defined topic directory. The main focus is on constructing appropriate feature spaces on which a classifier operates. In addition to the usual text-based term frequency vectors, we study XML twigs and tag paths as extended features that can be combined with text term occurrences in XML elements. Moreover, we show how to leverage ontological background information, more specifically, the WordNet thesaurus, for the construction of more expressive feature spaces. For efficiency our implementation computes features incrementally and caches ontology entries. Our experiments demonstrate the improved accuracy of automatic classification based on the enhanced feature spaces.

1. INTRODUCTION

Despite the great advances on XML data management and querying, the currently prevalent XPath- or XQuery-centric approaches face severe limitations when applied to XML documents in large intranets, digital libraries, federations of scientific data repositories, and ultimately the Web. In such environments, data has much more diverse structure and annotations than in a business-data setting and there is virtually no hope for a common schema or DTD that all the data complies with. Without a schema, however, database-style querying would often produce either empty result sets, namely, when queries are overly specific; or way too many results, namely, when search predicates are overly broad, the latter being the result of the user not knowing enough about the structure and annotations of the data. This calls for applying information retrieval (IR) technology, in particular, the ranked retrieval paradigm, but at the same time adding structural search conditions to the traditional IR repertoire of keyword queries [6, 7, 14, 8].

An important IR technique is automatic classification for organizing documents into topic directories based on statis-

tical learning techniques [10, 3, 12, 4]. Once data is labeled with topics, combinations of declarative search, browsing, and mining-style analysis is the most promising approach to find relevant information, for example, when a scientist searches for existing results on some rare and highly specific issue. This paper explores automatic classification for schemaless XML data. The anticipated benefit is a more explicit, topic-based organization of the information which in turn can be leveraged for more effective searching. The main problem that we address towards this goal is to understand which kinds of features of XML data can be used for high-accuracy classification and how these feature spaces should be managed by an XML search tool with user-acceptable responsiveness.

This paper systematically explores the design space outlined above by investigating features for XML classification that capture annotations (i.e., tag-term pairs), structure (i.e., twigs and tag paths), and ontological background information (i.e., mapping words onto word senses). Particularly challenging issues are how to deal with a very small training basis, on one hand, and how to handle documents that include multiple topics and a correspondingly heterogeneous vocabulary, on the other hand. For both issues exploiting structural and ontological features that go beyond the standard bag-of-words model (i.e., context-free terms and their frequencies) seems to be a promising direction.

2. FEATURE SPACES FOR XML DATA

Using only text terms (e.g., words, word stems, or even noun composites) and their frequencies (and other derived weighting schemes such as $tf \cdot idf$ measures [10]) as features for automatic classification of text documents poses inherent difficulties and often leads to unsatisfactory results because of the noise that is introduced by the idiosyncratic vocabulary and style of document authors. The key lies in a more precise document characterization for semistructured data such as XML by exploiting its structure and annotation. For XML data we postulate that tags (i.e., element names) will be chosen much more consciously and carefully than the words in the element contents and expect a certain awareness of authors for the need of meaningful and reasonably precise annotations and structuring.

So we view tags as high-quality features of XML documents. When we combine tags with text terms that appear in the corresponding element contents, we can interpret the resulting (tag, term) pairs almost as if they were (*concept*, *value*) tuples in the spirit of a database schema with at-

tribute names and attribute values. For example, pairs such as (programming_language, Java) or (lines_of_code, 15000) are much more informative than the mere co-occurrence of the corresponding words in a long text (e.g., describing a piece of software for an open source portal). Of course, we can go beyond simple tag-term pairs by considering entire *tag paths*, for example, a path “university/department/chair” in combination with a term “donation” in the corresponding XML element, or by considering *structural patterns* within some local context such as *twigs* of the form “homepage/teaching \wedge homepage/research” (the latter could be very helpful in identifying homepages of university professors).

An even more far-reaching option is to map element names onto an ontological knowledge base. This way, tags such as “university” and “school” or “car” and “automobile” could be mapped to the same semantic concept, thus augmenting mere words by their *word senses*. We can generalize this by mapping words to semantically related broader concepts (hypernyms) or more narrow concepts (hyponyms), if the synonymy relationship is not sufficient for constructing strong features. And of course, we could apply such mappings not just to the element names, but also to text terms that appear in element contents.

In doing this, we encounter a number of problems that need to be solved towards a viable and high-quality XML document classifier:

- *High dimensionality*: Combining tags with terms leads to a feature space of much higher dimensionality than the usual vector spaces used for document representation in IR and classification. Consequently, it will be populated much more sparsely than a simpler term vector space, and effectively training a classifier will be more challenging.
- *Ambiguity*: When mapping words onto ontological concepts the ambiguity of natural language often prevents a unique mapping. So we need some approach for disambiguating word senses based on the context of the word occurrence.
- *Noise*: We do not believe in semantically perfect annotations or perfect ontologies; terminology, even when chosen very carefully, is naturally diverse and evolves over time. So there will always be a fair amount of noise in the enhanced feature spaces.
- *Efficiency*: The high dimensionality of the richer feature spaces and also the mappings onto ontological concepts incur significant computational overhead; so an efficient implementation is all but straightforward.

2.1 Combining Terms and Tags

When parsing and analyzing an XML document we extract characteristic words from element contents by means of simple stopword filtering or noun recognition, and we combine these text terms with the corresponding element name; these pairs are encoded into a single feature of the form *tag\$term* (e.g., “car\$Passat”).

We include all tags in this combined feature space, but for tractability and also noise reduction only a subset of these features is selected in the input vectors of the classifier. For each topic and its competing topics in the classification (i.e., the sibling nodes in the topic directory), we compute, from

the training documents, the *Mutual Information* [1, 10] (MI, aka. relative entropy or Kullback-Leibler divergence) between the topic-specific distribution of the features and a distribution in which the features of documents are statistically independent of the topics:

relevance of feature X_i

for discriminating topic C_k :

$$MI(X_i, C_k) = \sum_{X \in \{X_i, \bar{X}_i\}, C \in \{C_k, \bar{C}_k\}} P[X \wedge C] \log \frac{P[X \wedge C]}{P[X]P[C]}$$

Sorting the features in descending order of this measure gives us a ranking in terms of the discriminative power of the features, and we select the top m features for the classifier (usually, with m being in the order of 500 up to 5000). Thus, we obtain a ranking for the complete set of tag-term pairs that occur in the training data, because different element types (i.e., with different tags) usually imply different sets of specifically characteristic terms. For example, in digital library documents terms such as “www” or “http” are more significant within elements tagged “content”, “section”, or “title” rather than elements tagged “address” where they may simply indicate the author’s homepage URL.

2.1.1 Feature Weighting

Term-based features are usually quantified in the form of $tf \cdot idf$ weights [1, 10] that are proportional to the frequency of the term (tf) in a given document and to (the logarithm of) the inverse document frequency (idf) in the entire corpus (i.e., the training data for one topic in our case). So the highest weighted features are those that are frequent in one document but infrequent across the corpus. Analogously to the arguments for feature selection, we compute tf and idf statistics for tag-term pairs. The weight $w_{ij}(f_i)$ of feature f_i in document j is computed as $w_{ij}(f_i) = tf_{ij} \cdot idf_i$, where idf_i is the logarithm of the *inverse element frequency* of term t_i in the corpus.

This way, the idf part in the weight of a term is implicitly computed for each element type separately without extra effort. For example, in a digital library with full-text publications, the pair (journal_title, transaction) would have low idf value (because of ACM Transactions on Database Systems, etc.), whereas the more significant pair (content, transaction) would have high idf value (given that there have been relatively fewer papers on transaction management in the recent past).

2.2 Exploiting Structure

Using tag paths as features gives rise to some combinatorial explosion. However, we believe that this is a fairly modest growth, for the number of different tags used even in a large data collection should much smaller than the number of text terms and we expect real-life XML data to exhibit typical context patterns along tag paths rather than combining tags in an arbitrarily free manner. These characteristic patterns should help us to classify data that comes from a large number of heterogeneous sources. Nevertheless, efficiency reasons may often dictate that we limit tag-path features to path length 2, just using (parent tag, tag) pairs or (parent tag, tag, term) triples.

Twigs are a specific way to split the graph structure of XML documents into a set of small characteristic units with respect to sibling elements. Twigs are encoded in the form

“left child tag \$ parent tag \$ right child tag”; examples are “research\$homepage\$teaching”, with “homepage” being the parent of the two siblings “research” and “teaching”, or “author\$journal_paper\$author” for publications with two or more authors. The twig encoding suggests that our features are sensitive to the order of sibling elements, but we can optionally map twigs with different orders to the same dimension of the feature space, thus interpreting XML data as unordered trees if this is desired.

For tag paths and twig patterns as features we apply feature selection to the complete structural unit. MI is computed on *distinct feature sets* each for tag-term pairs and twig features. The final proportion of both kinds of features in the resulting topic-specific feature spaces is scaled by empirical consideration of the given data and depending on their degree of structural diversity.

2.3 Exploiting Ontological Knowledge

Rather than using tags and terms directly as features of an XML document, an intriguing idea is to map these into an ontological concept space. In this paper we have used WordNet [5, 11] as underlying ontology, which is a fairly comprehensive common-sense thesaurus carefully handcrafted by cognitive scientists. WordNet distinguishes between *words* as literally appearing in texts and the actual *word senses*, the concepts behind words. Often one word has multiple senses, each of which is briefly described in a sentence or two and also characterized by a set of synonyms, words with the same sense, called *synsets* in WordNet. In addition, WordNet has captured hypernym (i.e., broader sense), hyponym (i.e., more narrow sense), and holonym (i.e., part of) relationships between word senses.

In the following we describe how we map tags onto word senses of the ontology; the same procedure can be applied to map text terms, too. The resulting feature vectors only refer to word sense ids that replace the original tags or terms in the structural features. The original terms are no longer important while the structure of each feature remains unchanged. Obviously this has a potential for boosting classification if we manage to map tags with the same meaning onto the same word sense.

2.3.1 Mapping

A tag usually consists of a single word or a composite word with some special delimiters (e.g., underscore) or a Java-style use of upper and lower case to distinguish the individual words. Consider the tag word set $\{w_1, \dots, w_k\}$. We look up each of the w_i in WordNet, or actually a special database constructed from WordNet’s contents (see Section 3.2), and identify possible word sense s_{i_1}, \dots, s_{i_m} . For example, for a tag “goal” we would find the word senses:

1. goal, end – (the state of affairs that a plan is intended to achieve and that (when achieved) terminates behavior to achieve it; “the ends justify the means”)
2. goal – (a successful attempt at scoring; “the winning goal came with less than a minute left to play”)

and two further senses. By looking up the synonyms of these word senses, we can construct the synsets {goal, end, content, cognitive content, mental object} and {goal, score} for the first and second meaning, respectively. For a composite tag such as “soccer_goal”, we look up the senses for both

“soccer” and “goal” and form the cross product of possible senses for the complete tag and represent each of these senses by the union of the corresponding two synsets. Obviously, this approach would quickly become intractable with growing number of words in a composite tag, but more than two words would seem to be extremely unusual.

Now the key question is: which of the possible senses of a tag is the right one? Our disambiguation approach is based on word statistics for some local context of both the tag that appears in some document and the candidate senses that are extracted from the ontology. For a tag t_i , we consider the full text content (including the tag name) of the corresponding element and all its subordinate elements as a *local context* $con(t_i)$ of tag t_i in document d_j . For a candidate word sense s_j , we extract synonyms, all immediate hyponyms, holonyms, and hypernyms, and optionally the hyponyms of the hypernyms (i.e., the siblings of s_j in the relationship graph). Each of these has a synset and also a short explanatory text. We form the union of the synsets and corresponding texts, thus constructing a *local context* $con(s_j)$ of sense s_j for $j \in \{1, \dots, p\}$. As an example, the context of sense 1 of the word “goal” (see above) corresponds to the bag of words {goal, end, state, affairs, plan, intend, achieve, ..., content, cognitive content, mental object, perceived, discovered, learned, ..., aim, object, objective, target, goal, intended, attained, ...}, whereas sense 2 would be expanded into {goal, successful, attempt, scoring, winning, goal, minute, play, ..., score, act, game, sport, ...}.

The final step towards disambiguating the mapping of a tag onto a word sense is to compare the tag context $con(t_i)$ with the context of candidates $con(s_1)$ through $con(s_p)$ in terms of a similarity measure between bags of words (note that the context construction may add the same word multiple times, and this information is kept in the word bag). Our implementation uses the cosine similarity between the tf*idf vectors of $con(t_i)$ and $con(s_j)$. Finally, we map tag t_i onto that sense s_j whose context has the highest similarity to $con(t_i)$. Denote this sense as $sense(t_i)$ and the set of all senses of tags that appear in the training data as $senses_{train} = \{sense(t_i) | t_i \text{ appears in training data}\}$.

2.3.2 Quantified Relationships

The feature space constituted by the mapping of tags onto word senses is appropriate when training documents and the test documents to be classified have a major overlap in their word senses (i.e., the images of their tag-to-word-sense mappings). In practice, this is not self-guaranteed even for test documents that would indeed fall into one of the trained topics. Suppose, for example, that we have used training documents with tags such as “goal”, “soccer”, and “shot”, which are all mapped to corresponding word senses, and then a previously unseen test document contains tags such as “Champions_League”, “football”, and “dribbling”, which correspond to a disjoint set of word senses. The classifier would have no chance to accept the test document for topic “sports”; nevertheless we would intellectually rate the test document as a good candidate for sports.

To rectify this situation, we define a similarity metric between word senses of the ontological feature space, and then map the tags of a previously unseen test document to the word senses that actually appeared in the training data and are closest to the word senses onto which the test document’s tags would be mapped directly.

For defining a sense-to-sense similarity metric, we pursue a pragmatic approach that exploits term correlations in natural language usage. We consider the synsets $synset(s_1)$ and $synset(s_2)$, and estimate the statistical correlation between these two word sets in a very large text corpus. As the underlying corpus we use the Web itself, by performing a large crawl that starts with topic-specific training data (not only XML, but also HTML documents that are characteristic for the given topic). From the crawl result we compute the Dice coefficient [10]:

$$dice(s_1, s_2) = \frac{2 \cdot df(synset(s_1) \cup synset(s_2))}{df(synset(s_1)) \cdot df(synset(s_2))}$$

To avoid combinatorial explosion and the computational overhead in the critical path of the classifier, we have pre-computed these similarity measures for all neighboring word senses in a graph whose edges reflect hypernym/hyponym relationships. At run-time the word sense s' that is most similar to a given word sense s is determined by finding the shortest path between the two senses in the relationship graph and averaging the similarity measures of the corresponding edges.

Putting everything together, a tag t_i in a test document d_j is first mapped to a word sense $s := map(t_i)$, then we find the closest word sense $s' := argmax_{s'} \{sim(s', s) | s' \in senses_{train}\}$ that is included in the feature space of the training data, and scale the weight of the feature containing s in document d_j by $sim(s, s')$ based on concept similarities estimated from large-scale crawling. Fig. 1 depicts the architecture of our method.

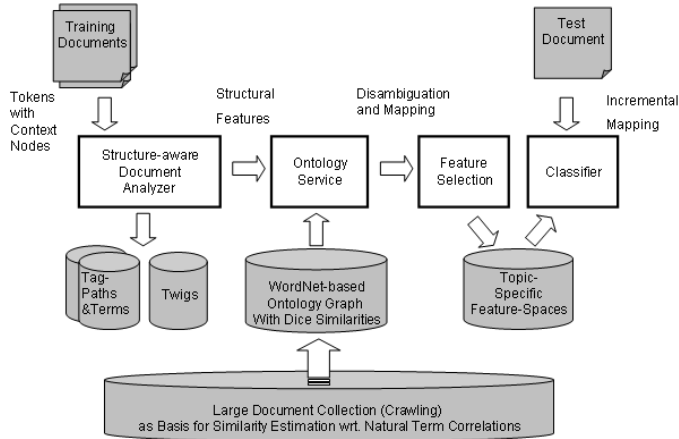


Figure 1: Exploiting Structure and Ontological Knowledge for Classification.

3. IMPLEMENTATION

3.1 Classifier

Document classification consists of a training phase for building a mathematical decision model based on intellectually preclassified documents, and a decision phase for classifying new, previously unseen documents fetched by the crawler. We chose *Support Vector Machines (SVM)* [2, 15, 9] as a leading-edge classification method. More specifically, we use the SVM-light V5.00 software, restricting ourselves to linear SVMs.

The *hierarchical multi-class* classification problem for a tree of topics is solved by training a number of binary SVMs, one for each topic in the tree. For each SVM the training documents for the given topic serve as positive samples, and we use the training data for the tree siblings as negative samples. SVM computes a maximum-margin separating hyperplane between positive and negative samples in the feature space; this hyperplane then serves as the decision function for previously unseen test documents (by computing a simple scalar product). A test document is recursively tested against all siblings of a tree level, starting with the root's children, and assigned to all topics for which the classifier yields a positive decision (or, alternatively, only to the one with the highest positive classification confidence).

3.2 Ontology Service

The internal structure of our ontology service is derived from the WordNet graph $G = (S, E_t)$ and stored in a set of database relations, i.e., a relation for the nodes of the ontology graph that yield all known synsets S , and a relation R_t for each of the supported edge types *hypernym*, *holonym*, and *hyponym* that connect these synsets nodes. The ontology graph provided by WordNet is enriched by edge-weights. For each edge type, all edges of the ontology graph are stored as triples $R_t = \{(id_s, id_t, dice(s, t))\}$, where $id_s, id_t \in E_t$, $dice(s, t) \in [0, 1]$ and $t \in \{hyponym, hypernym, holonym\}$, including the ids of the source and the target synset and the quantified similarity estimated by the Dice coefficient (see Section 2.3.2).

Our implementation of the ontology service provides the following basic functions (these functions can be invoked as a Java class API, via RMI to a Java bean, or as Web Service methods using SOAP):

- **findConceptByTerm**: For a document tag or term t , find all synsets S that contain t . Since a term may have different meanings depending on the context that it occurs in (*polysemy*), this method often returns more than one synset.
- **disambiguateConcepts**: For a given tag or term t_i along with its context $con(t_i)$ in a document, find the most suitable synset from a set of synsets S by comparing the cosine between $con(t_i)$ and all synset contexts $con(s_i)$ with $s_i \in S$. This method aims to eliminate the polysemy of terms that appear in multiple synsets.
- **findConceptSimilarity**: Query the ontology graph to detect the shortest path p between two concepts s_1 and s_2 and return the average edge weight $sim_p(s_1, s_2)$ between the source and the target node.
- **findBestMatch**: This method subsequently compares the similarity $sim_p(s, s_i)$ for $s_i \in senses_{train}$ and returns the training synset that maximizes $sim_p(s, s_i)$ (more details in Section 3.3).
- **getDiceCoeff**: This method computes the Dice coefficient for two arbitrary concepts s_1, s_2 by accessing the document frequencies $df(synset(s_1))$, $df(synset(s_2))$, and $df(synset(s_1) \cup synset(s_2))$ in the initial large-scale crawl directly.

To improve performance over frequently repeated queries, all result sets and retrieved similarities are cached. To implement the similarity search between two concepts s_i and

s_j , the ontology service searches for the shortest path between s_i and s_j . With regard to performance issues, we decided to avoid a breadth search over the complete ontology graph starting at each concept, and rather exploit the DAG-like structure of WordNet to use the transitive hypernym relations only. At run-time the sense's similarity $sim_p(s_i, s_j)$ is determined by finding the nearest common hypernym h_{s_i, s_j} that minimizes the sum of the lengths of the two paths $p_1 = \{s_i, \dots, h_{s_i, s_j}\}$ and $p_2 = \{s_j, \dots, h_{s_i, s_j}\}$ that connect s_i and s_j via h_{s_i, s_j} and determine the average edge weight of p_1 and p_2 .

$$sim_p(s_i, s_j) = \frac{1}{n + m} \left(\sum_{k=1}^{n-1} dice(s_{ik}, s_{i, k+1}) + \sum_{l=1}^{m-1} dice(s_{jl}, s_{j, l+1}) \right)$$

If there is more than one common hypernym that minimizes the path length, we choose the one with highest average edge similarity sim_p .

3.3 Incremental Mapping for Classification

In the *classification phase* we aim to extend the test vector by finding approximate matches between the concepts in feature space and the formerly unknown concepts of the test document. Like the training phase, the classification identifies synonyms and replaces all tags/terms with their disambiguated synset ids. 1) If there is a direct match between a test synset s and a synset in the training feature space (i.e., $s \in senses_{train}$), we are finished and put the respective concept-term-pair in the feature vector that is generated for the test document. 2) If there is no direct match between the test synset s and any synset derived from the training data (i.e., $s \notin senses_{train}$), we replace the actual test synset with its most similar match s' using the 'findBestMatch' method. Features which were removed in the training documents through feature selection are also skipped in the test documents. The weight $w_{ij}(f_i)$ of feature f_i in document j is now scaled by $sim_p(s, s')$, i.e., $w_{ij}(f_i) = sim_p(s, s') tf_{ij} idf_i$, to reflect the concept similarities of approximate matches in the feature weights.

To avoid topic drift, we limit the search for common hypernyms to a depth of 2 in the ontology graph. Concepts that are not connected by a common hypernym within this threshold are considered as dissimilar and obtain a similarity value of 0.

4. PRELIMINARY EXPERIMENTS

4.1 Setup

The following preliminary experiments were designed to shed initial light into the specific advantages and disadvantages of our approaches, using two different data sets in comparison to a standard term-based text classifier built from the element names and textual contents using standard $tf \cdot idf$ weights. All tests were run for two-topic directories, which is the simplest case from the classification point-of-view and directly corresponds to the binary SVM classification schema. To measure classification quality, we used the *F-measure* [1, 10] which is the harmonic mean of precision and recall, a standard quality measure in IR. Training and tests were performed on disjoint document sets.

4.2 The Internet Movie Database

The first data set consists of an XML version of the IMDB (Internet Movie Database) collection (www.imdb.org). The IMDB is available as a plain text database with structured records that contain movie titles, actors, genres, etc. We generated one XML document for each movie and automatically annotated the resulting XML elements according to the attributes of the corresponding records (e.g. title, actor, role, summary, goof, trivia, quotes, etc.). An XML tag was generated for each attribute. This resulted in a maximum nesting depth 4 (e.g., movie/casting/character/actor). To test the classifier we choose a training set with $n = 50$ movies for each of the two topics 'Action' and 'Western'. We used this genre label as the target of the classifier and removed the genre attribute from all test documents. The aim of this test is to study the precision and recall of tag-term pairs and twigs compared to traditional text classification. We varied the number m of features from $m = 10$ to $m = 10,000$ per topic using the MI criterion for both the pure textual and the structure-conscious classifiers. Since there are variations in tag ancestor/descendant/sibling occurrences, we dedicated 5 percent of the overall feature space dimensions to the most specific twig features, again using MI as selection criterion for the best twigs. So a large fraction of the feature space dimensions capture the best tag-term pairs, and ontology lookups were restricted to tags only. Before feature selection, the (maximum) structural feature space contained 25543 distinct tag-term pairs and twigs versus 12062 distinct text terms extracted from a total of 100 training documents.

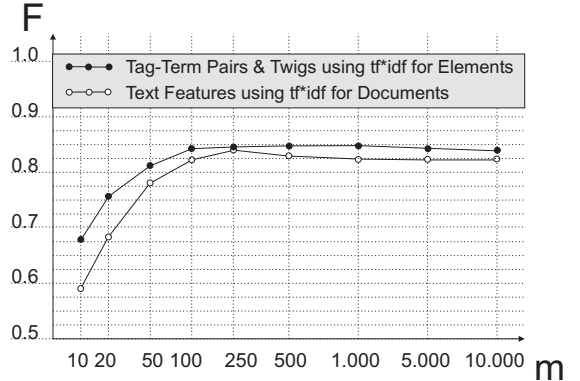


Figure 2: Classification of 'Action' versus 'Western' for the IMDB movie collection using very few features.

Figure 2 shows the improved accuracy (in terms of the F-measure) of tag-term pairs and twigs over mere text features for less than 250 features. For low-dimensional feature spaces the structure-aware features yield a significantly better representation of the training data than pure text feature spaces can achieve. Interestingly, the best structural features after MI turn out to be format descriptions as depicted in table 1 (with stemmed terms).

4.3 The Reuters-21578 Dataset

The second data set consists of the Reuters-21578 collection from the Reuters newswire which is part of the TREC standard benchmark collection (trec.nist.gov). The Reuters collection has a fairly restricted vocabulary (only 7783 distinct terms are contained in our largest training collection

Action	Western
release_country\$usa	sound_mix\$mono
video_standard\$ntsc	filmed_in\$white
sound_encoding\$analog	filmed_in\$black
disc_format\$clv	writer\$alphabet
master_format\$film	summary\$longhorn
sound_encoding\$cx	summary_by\$abilen
aspect_ratio\$close	summary_by\$le
aspect_ratio\$ld	summary\$ride
aspect_ratio\$teletext	summary_by\$adam
aspect_ratio\$caption	role\$henchman

Table 1: Top 10 features for 'Action' versus 'Western' using MI. Synset ids are replaced by original tags/terms for better readability.

with 1000 documents). The articles are authored by professional writers according to guidelines and so they are uniformly structured and hardly exhibit any annotations (merely line breaks, but no semantically valuable tagging which makes it analogous to processing simple HTML). So this experiment focused on studying the benefits of ontological features without exploiting the document structure. We applied the mapping of terms into ontological concepts only to *nouns* in the text contents for a very small set of training documents for the two Reuters topics 'Acquisition' and 'Earnings'. Beginning with just 1 example per topic, we stepwise increased the training basis to 500 samples per topic (starting with the longest documents and adding documents in descending order of length, i.e., their amount of terms).

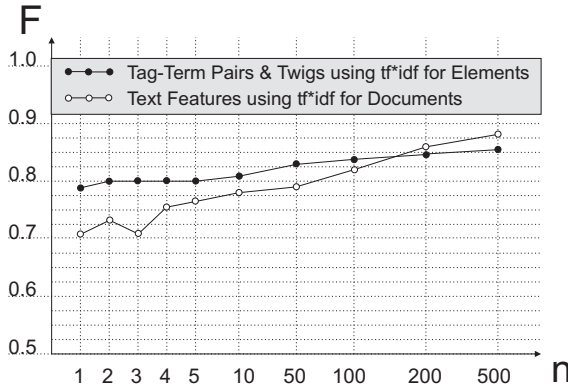


Figure 3: Classification of 'Acquisition' versus 'Earnings' for the Reuters-21578 collection using very few training documents.

Figure 3 shows that for less than 10 training samples, the classifier with tag-term pairs using ontology lookups on all known nouns clearly outperformed the standard text classifier. Moreover, the classifier with the ontology mapping seemed to be less sensitive to noise, whereas the pure text classifier showed some fluctuations created by noise. For more than 100 training samples we again observe a saturation effect in the text feature space; at this point both classifiers perform equally well and adding more samples does not lead to any richer feature spaces. The phenomenon that the ontology classifier is finally outperformed by the text classifier can be attributed to the fact that the disambiguation

of text terms does not always work correctly. At some point, a non-negligible number of terms is mapped to wrong concepts in the ontology (as we could verify intellectually), and this results in reduced accuracy.

5. CONCLUSIONS

This paper is a first step towards understanding the potential of exploiting structure, annotation, and ontological features of XML documents for automatic classification. Our experiments are fairly preliminary, but the results indicate that the direction is worthwhile to be explored in more depth. Our current and near-future work includes more comprehensive experimental studies and making the ontological mapping (namely, IR-style word sense disambiguation) more robust. In particular, we are working on incorporating other sources of ontological knowledge into our system to go beyond the information provided by WordNet. This ongoing work is part of the BINGO! project in which we are building a next-generation focused crawler and comprehensive toolkit for automatically organizing and searching semistructured Web and intranet data [13].

6. REFERENCES

- [1] R. Baeza-Yates, B. Ribeiro-Neto: Modern Information Retrieval. Addison Wesley, 1999.
- [2] C.J.C. Burges: A Tutorial on Support Vector Machines for Pattern Recognition. Data Mining and Knowledge Discovery 2(2), 1998.
- [3] S. Chakrabarti, M. Berg, B. van den and Dom: Focused Crawling: A New Approach to Topic-specific Web Resource Discovery. WWW Conference, 1999.
- [4] R. Duda, P. Hart, D. Stork: Pattern Classification, Wiley, 2000.
- [5] C. Fellbaum: WordNet: An Electronic Lexical Database. MIT Press, 1998.
- [6] N. Fuhr, K. Grossjohann: XIRQL: A Query Language for Information Retrieval in XML Documents. ACM SIGIR, 2001.
- [7] T. Grabs, H.-J. Schek: Generating Vector Spaces On-the-fly for Flexible XML Retrieval. XML and Information Retrieval Workshop - ACM SIGIR, 2002.
- [8] S. Guha, H.V. Jagadish, N. Koudas, D. Srivastava, T. Yu: Approximate XML Joins, ACM SIGMOD, 2002.
- [9] T. Joachims: The Maximum-Margin Approach to Learning Text Classifiers. Ausgezeichnete Informatikdissertationen 2001, D. Wagner et al. (Hrsg.), GI-Edition - Lecture Notes in Informatics (LNI), Koellen Verlag, Bonn, 2002.
- [10] C.D. Manning, H. Schuetze: Foundations of Statistical Natural Language Processing. MIT Press, 1999.
- [11] G. Miller: Wordnet: A Lexical Database for English. Communications of the ACM 38(11), 1995.
- [12] T. Mitchell: Machine Learning. McGraw Hill, 1996.
- [13] S. Sizov, G. Weikum, et al: The BINGO! System for Information Portal Generation and Expert Web Search. CIDR, 2003
- [14] A. Theobald, G. Weikum: The Index-Based XXL Search Engine for Querying XML Data with Relevance Ranking. EDBT, 2002.
- [15] V. Vapnik: Statistical Learning Theory. Wiley, 1998.

Automatic Annotation of Data Extracted from Large Web Sites

Luigi Arlotta¹ Valter Crescenzi¹ Giansalvatore Mecca² Paolo Merialdo¹

¹Università Roma Tre
Via della Vasca Navale, 79
I-00146 – Roma, Italy

{arlotta,crescenzi,meraldo}@dia.uniroma3.it

²Università della Basilicata
C.da Macchia Romana
I-85100 - Potenza, Italy
mecca@unibas.it

ABSTRACT

Data extraction from web pages is performed by software modules called wrappers. Recently, some systems for the automatic generation of wrappers have been proposed in the literature. These systems are based on unsupervised inference techniques: taking as input a small set of sample pages, they can produce a common wrapper to extract relevant data. However, due to the automatic nature of the approach, the data extracted by these wrappers have anonymous names. In the framework of our ongoing project ROADRUNNER, we have developed a prototype, called LABELLER, that automatically annotates data extracted by automatically generated wrappers. Although LABELLER has been developed as a companion system to our wrapper generator, its underlying approach has a general validity and therefore it can be applied together with other wrapper generator systems. We have experimented the prototype over several real-life web sites obtaining encouraging results.

1. INTRODUCTION

The extraordinary amount of data that is currently available on the Web cannot be fully exploited because web pages are intended to be browsed by humans, and not computed over by applications. This drawback has motivated several researchers to study methods and techniques aimed at facilitating the creation of *wrappers*, i.e. web data extraction programs (see [12, 10] for surveys on wrappers).

Some recent proposals, based on machine learning approaches, are able to semi-automatically create wrappers. Since the major effort of the programmer is the production of the training set, several researchers have focused on the development of systems that offer sophisticated user interfaces to assist the programmer in building the training set for the target pages. These systems allow us to quickly build wrappers for pages that contain data we are interested in.

However, since the creation of wrappers as well as their

maintenance require a human intervention, the costs of a reliable application dramatically augment with the number of wrapped sources.

In order to reduce the wrapper production and maintenance costs, recent studies have concentrated on the automatic generation of wrappers. In the context of the ROADRUNNER project, we have developed a system that, based on a grammar inference approach [4, 3], automatically infers a wrapper from a collection of similar pages. Based on different techniques, systems for the same goal have been recently developed also by Arasu and Garcia-Molina [1] and by Wang and Lochovsky [14].

Automatic systems leverage on the observation that data published in the pages of very large sites usually come from a back-end database and are embedded within a common HTML template. Therefore many pages share a common structure, and differences correspond to the data coming from the database. The wrapper generation process aims at inferring a description of the common template, which is then used to extract the embedded data values.

These proposals reduce but do not eliminate the need for a human intervention. Since wrappers are built automatically, the values that they extract are anonymous and a human intervention is still required to associate a meaningful name to each data item. The automatic annotation of data extracted by automatically generated wrappers is a novel problem, and it represents a step towards the automatic extraction and manipulation of web data.

This paper reports our recent researches for automatically annotating data extracted from data-intensive web sites. Our proposal is based on the observation that web pages are designed to be consumed by human users, and therefore they usually contains text strings, i.e. labels, whose goal is to explicate to the final user the intentional meaning of the published data.

We have developed and experimented a working prototype, called LABELLER, that works as a companion system to our wrappers generator. Although LABELLER has been developed in the framework of the ROADRUNNER project, its underlying approach can be applied together with other wrapper generator systems.

The paper is organized as follows: Section 2 presents the problem we address; Section 3 overviews our approach and describes the algorithm we have implemented to annotate anonymous data. Experiments with the prototype system are reported in Section 4. Section 5 discusses limitations and opportunities of our approach. Finally Section 6 presents

some related work.

2. PROBLEM DESCRIPTION

We have analyzed about 50 automatically generated wrappers that work on pages from several web sites: a large majority of the data extracted by the wrappers are accompanied with a string representing a meaningful name of the value. This is not surprising: since web pages are intended to be browsed by humans, it is a common practice that the published data are accompanied by textual descriptions to help the user to correctly interpret the underlying information. Therefore, our approach for labelling data extracted from automatically generated wrappers relies on the observation that important information about the semantics of data is often available on the web pages themselves.

Let us introduce an example to present the overall context and the issues we address. Consider the HTML pages shown in Figure 1. Feeding our wrappers generator system with a set of sample pages like these, we obtain a grammar description of the common template, as follows:¹

```
<html>...(<b>Brand:</b>$J<hr><b>Model:</b>$K<hr>...
<div><b>Our Price:<br>$Q</b></div>...)+...</html>
```

This grammar works as wrapper by considering the non-terminals symbols ($\$J$, $\$K$, ...) as place-holders: during the parsing of the input pages, they match with the strings to extract. Data extracted from the two pages by this wrapper are shown in Figure 2.

It is worth noting that the extracted data are anonymous; nevertheless, looking at the source pages in Figure 1, we observe that useful labels are available on the pages themselves as part of the common template. For instance, the bold-face “Brand” compares nearby the extracted value “Seiko”, “Model” is close to “SXJZ32”, “SZZB14”, and so on. More interestingly, many of the anonymous fields extracted by the wrapper could be named after some terminal symbol of the grammar itself. For instance the terminal symbol **Brand** could be the label for all the values of $\$J$.

The goal of LABELLER is that of analyzing the wrapper and a set of sample pages in order to locate inside the common template meaningful labels for the otherwise anonymous data.

We fix the terminology in this settings by calling *variants* the non-terminal symbols and *invariants* the non-tag terminal symbols of the grammar.² We call *data-values* the strings that match with variants (e.g. “Seiko”, “SXJZ32”, “SZZB14”) and *labels* those matching with invariants (e.g. “Brand”, “Model”). More precisely, LABELLER’s goal is that of associating to each variant a meaningful label chosen among the invariants.

3. THE ROADRUNNER LABELLER

Our technique to annotate extracted data is inspired by the following observation: since web pages are designed to be presented on a browser to a human user, usually values and labels are *visually close* to each other.

Therefore, first LABELLER computes the coordinates of the bounding boxes of every data-value and every label in a

¹For the sake of presentation, we report a partial and simplified version of the real grammar.

²This terminology remarks whether they match or not different strings from one page to the other.

given sample page (as rendered by a graphical web browser in standard 800×600 resolution). Then it tries to find the optimal association label/data-value by analyzing their spatial relationships.

We have developed several heuristics to establish the correct associations: (i) values and labels are close to each other, (ii) usually a label is vertically, horizontally, or centrally aligned to its associated values, (iii) labels are usually placed either to the left or above values, (iv) it is not allowed that either a label or a value is between another value and its label.

Also, we impose the constraint that a variant cannot have more than one label.

Actually, the same variant/invariant could occur many times if the sample pages contain list of items, such as tables. In this cases, we simply consider only the first occurrences of each variant/invariant.

3.1 The Labelling Algorithm

The algorithm implemented by LABELLER is illustrated in Figure 3: it takes as input a wrapper and a set of sample pages; as output, it produces a set of label/variant associations for the given wrapper.

Initially it chooses an arbitrary order for the set of labels $L = \{l_i, i = 1 \dots N_l\}$ and for the set of variants $V = \{v_j, j = 1 \dots N_v\}$ of the wrapper. Then it calls the function LABELS to compute a set of label/variant associations (l_i/v_j) on each of the given samples.

LABELS tries to exploit the *spatial relationships* between data-values and labels in the graphical rendering of one web page as displayed by an ordinary browser. It computes an $N_l \times N_v$ matrix M of *scores*, that is positive real numbers such that $M[i, j]$ is a measure of the “goodness” of l_i as label for v_j ; as smaller are the scores as better are the associations.

Some associations are immediately discarded without further evaluation by setting the corresponding score in M to $+\infty$. Namely, we discard (l_i/v_j) if any of the following conditions holds: (i) l_i is below or to the right of v_j ; (ii) the distance between l_i and v_j is greater than D_{max} pixels; (iii) l_i is placed on the diagonal of v_j ; (iv) there exists another variant/invariant which is located between l_i and v_j . The first three conditions reflect the fact that a label is usually placed above or to the left of the corresponding data-value, never too far from it. The last condition considers that anything between a data-value and its label would keep the reader from intuitively associating them.

Given a pair (l/v) and the corresponding bounding-boxes on a page s , our score function is defined as follows:

$$SCORE(l, v, s) = DISTANCE(l, v, s) \cdot \sin(2 \cdot ALIGNMENT(l, v, s))$$

This definition captures two aspects: *distance* and *alignment*, because labels are usually close and aligned to the corresponding data-values.

The distance is simply defined as the minimum distance between the two bounding boxes. The ALIGNMENT factor is a measure of the alignment and is computed as the smallest angle amongst several possibilities. Namely, we consider the line crossing at the two midpoints, and all the lines crossing at the two farthest point on the same side (left, right, top, or bottom) of the two bounding-boxes; then we consider all possible (absolute values of) angles encompassed between one coordinate axis and these lines. Note that $\sin(2 \cdot ALIGNMENT(l, v, s))$ has a minimum when the boxes



Figure 1: Input HTML Pages from <http://www.watch.com>



I	J	K	L	M	N	O	P	Q
	Seiko	SXJZ32	Ladies	"Two-tone bracelet, Jewelry clasp"	"Silver-tone case, Pearl dial"	"Seiko Ladies bracelet, Two-tone, Gold colored hands and markers, Pearl dial, Movement Japan"	\$225.00	\$168.75
	Seiko	SZZB14	Ladies	Two-tone stainless steel bracelet	Two-tone case with a White dial	Seiko Ladies' Watches - Two-tone hands and markers	\$200.00	\$150.00

Figure 2: Data Extraction Output

are perfectly aligned, and a maximum when one is perfectly on the diagonal of the other.

Function LABELS terminates by producing the associations (l_i/v_j) according to a trivial *best first* strategy.

The results of every LABELS invocation are then collected and only the associations which have been established without contradictions in all input samples are considered part of the final output. That is, if the same label has been associated to different variants in different samples, it will not be part of the output.

While it is clearly possible that associations inferred according to the scores are due to random factors, i.e. a data-value and one label are incidentally close each other in one sample, it is unlikely that the two will be rendered so close in many samples if the web-designer did not “plan” this way. The experiments have confirmed this observation: the comparison of the results on several samples reduced the number of false positives.

4. EXPERIMENTS

Based on the above algorithm, we have developed LABELLER, a working prototype of the labelling system and used it to run a number of experiments on real web sites. We have generated wrappers for 19 collections of pages, each collection containing about 10 pages. We have then used these wrappers to extract data from the input pages. Finally, we have run LABELLER to annotate the extracted data with a label extracted from the input pages.

To evaluate the effectiveness of the system we have manually defined and stored in a database the correct associations between data-values and labels. The results produced by LABELLER have been compared against this collection.

The table in Figure 4 reports the results of our experiments and contains the following elements:³ (i) *samples*: the url of the site and a short description of the collection of pages considered for the experiment. (ii) *Wrapper*: some elements about the inferred wrapper; in particular we report the level of nesting of the inferred schema (*nesting*), the number of variants (*#variants*) and the number of invariants (*#invariants*) of the inferred wrapper. (iii) *Results*: the results obtained from the LABELLER; namely: number of correctly labelled data attributes (*#ok*); number of errors, i.e. number of variants the system has associated a wrong label with (*#error*); number of false positives (*#fp*), i.e. number of variants that although do not have a label in the source page, they have been associated with a label; number of false negative (*#fn*), i.e. number of variants for which the system was not able to associate a label, although a label was present in the source page.

As it can be seen from the table, for a large majority of variants (around 90%) the system is able to correctly identify the right label, even in the presence of nested structures. The system produces a small number of false positives (around 5%) and a very small number of errors (around 1.5%).

Finally, note that the system correctly associates a label with 51% of the wrapper variants. Here it is important to observe the association variant-label is at the schema level. Most of the labelled variants represent data-values which are repeated in the pages, e.g. data-values associated with

variants \$J, \$K, ... in Figure 2. As a consequence the labels cover a large majority of the extracted data-values (for our set of pages, around 82% of data-values).

5. LIMITATIONS AND OPPORTUNITIES

The encouraging results of our experiments show that our technique can be adopted as a starting point for more sophisticated approaches. The capability to label data can sensibly improve the degree of automation of related techniques [14, 13, 5, 7]. Our approach essentially requires only the distinction between variants and invariants and can be easily adapted to different automatic wrapper generation systems [1, 14].

The most restrictive hypothesis of our data annotation approach is rather obvious: we need that textual labels describing the meaning of the extracted fields are present in the page. However, many data are published on web pages leaving their meaning implicit.⁴ In fact, often pages do not include explicit labels for those data whose semantics can be clearly understood from the context (e.g. the title of a book in a page describing details about that book). On the other hand, we observe that it is unlikely, especially in a data-intensive web site, that a large number of relevant attributes are left unlabelled. Just to give an example, consider a bookstore site, such as `amazon.com`: usually the authors and the title of books are not explicitly labelled; on the contrary, detailed data, such as ISBN, publisher, prices and many other attributes, are presented with an associated label. More important, labels are usually associated with those data that can assume an ambiguous meaning; following the bookstore example, labels like “our price”, “saving”, “list price” are associated with data dealing with prices: these labels are indispensable to correctly interpret the published information.

We observe that the meaning of data implicitly described by the context can be interpreted by domain ontologies [6]; on the contrary, ontology based approaches cannot work with several ambiguous data, such as the prices of our example. Therefore we believe that it might be interesting to combine our approach with ontology-based annotation systems.

Another intriguing direction that we aim at investigating is that of searching web pages containing the unlabelled data-values in other web sites and looking for a label in the search results. This is an idea that somehow recalls the DIPRE technique proposed by Brin [2].

6. RELATED WORK

The problem we address is somehow new, since the development of automatic wrapper generator systems is a recent result.

A related problem is that of identifying instances of specific value domains, such as people names, phone numbers, prices, and so on. Software modules specialized to this issue are the so called *recognizers* [11]. However their usage in the context of web data extraction is limited and the number of domains they can work on is relatively small. Also they cannot associate different labels with data-values of the same domain; for example they can identify several variants representing a price in a page, but they cannot distinguish

³The parameters were experimentally chosen to be good: $D_{max} = 250$ and $\alpha = 0.02$.

⁴Also, it is worth saying that labels may be present as images and not as texts.

Algorithm LABELLER

Parameters : D_{max} , the maximum allowed distance from a variant to its label;
 α , a threshold for the SCORE function;

Input : a wrapper w , a set of samples $S = \{s_1, \dots, s_n\}$;

Output : a set $\{(l/v)\}$ of associations;

begin

Let $L = \{l_i, i = 1 \dots N_l\}$, $V = \{v_j, j = 1 \dots N_v\}$ **be** respectively the labels and the variants in w ;

Let A_k **be** LABELS(L, V, s_k), $k = 1 \dots n$;

Let A **be** the set of associations (l/v) such that l is associated only to v
 and v is associated only to l in $\cup_{k=1 \dots n} A_k$;

return A ;

end

a set $\{(l_i/v_j)\}$ LABELS(a set of labels L , a set of variants V , a sample s)

begin

Let M **be** an $N_l \times N_v$ matrix such that

$M[i, j] = \begin{cases} +\infty & , \text{ if VALIDPOSITION}(l_i, v_j, s) \text{ or INBETWEEN}(l_i, v_j, L, V, s) \text{ or SCORE}(l_i, v_j, s) \geq \alpha D_{max}; \\ \text{SCORE}(l_i, v_j, s), & \text{ otherwise;} \end{cases}$

Let A **be** the empty set;

do begin

 add $(l_{i_{min}}/v_{j_{min}})$ to A such that $M[i_{min}, j_{min}]$ is the minimum of M ;

 set $M[i_{min}, h]$ and $M[k, j_{min}]$ to $+\infty$, $h = 1 \dots N_l$, $k = 1 \dots N_v$;

while $(\exists M[i, j] \neq +\infty)$;

return A ;

end

boolean VALIDPOSITION(label l , variant v , sample s)

 { **return** (in the graphical rendering of s , l is placed either above or to the left of v
 and their distance is less than D_{max}); }

boolean INBETWEEN(label l , variant v , labels L , variants V , sample s)

 { **return** (there exists either a label or a value whose bounding-box
 in the graphical rendering of s is between those of l and of v); }

\mathbb{R}^+ DISTANCE(label l , variant v , sample s)

 { **return** (the minimum distance between bounding boxes of l and v in the rendering of s); }

\mathbb{R}^+ SCORE(label l , variant v , sample s) { **return** DISTANCE(l, v, s) $\cdot \sin(2 \cdot \text{ALIGNMENT}(l, v, s))$; }

Figure 3: The Labelling Algorithm

samples		wrappers			results			
site	description	nesting	#invariant	#variants	#ok	#error	#fp	#fn
www.fifaworldcup.com	players	1	93	30	22	0	1	0
www.fifaworldcup.com	teams	1	57	31	7	0	1	0
half.ebay.com	used books	2	26	29	5	2	0	1
it.finance.yahoo.com	stock quotes	1	52	26	16	1	1	0
match.com	mate finder	0	50	25	19	0	0	0
postershop.com	posters	1	29	16	6	0	0	0
www.allrecipes.com	recipes	1	42	21	9	0	2	1
www.allrecipes.com	recipe index	1	22	11	2	0	1	1
superstore.ubid.com	products	1	35	15	6	0	2	0
www.watchzone.com	watches	1	12	16	8	0	0	0
www.polo.com	dresses	1	11	11	1	0	0	0
autos.yahoo.com	cars	2	70	58	46	0	2	1
www.hotjobs.com	job listing	1	29	10	4	0	2	0
www.overstock.com	jewelries	1	35	13	3	0	0	0
ncdc.noaa.gov	storms	1	51	11	9	0	1	0
www.fbi.com	most wanted	0	17	18	12	0	0	0
www.vitacost.com	drugs	1	30	23	15	0	0	0
houseandhome.msn.com	house finder	1	169	104	45	0	2	3
cinema.com	movies	1	13	12	10	1	0	1
totals			843	480	245	4	15	8
totals (%)					90,07%	1,47%	5,51%	2,94%

Figure 4: Experimental Results

the different meanings (list price, price and saving) of the variants.

Knoblock *et al.* have addressed problem which is reminiscent to ours in the context of the wrapper maintenance issue [13]. They have developed a wrapper generator system that needs to be trained by a set of labelled samples. Whenever a wrapper fails they have to regenerate the wrapper; to this end their system tries to automatically build a set of labelled examples by using the data extracted before the wrapper failure [9].

Some wrapper generation approaches are based on domain ontologies [6, 8]: they derive the data extraction rules from knowledge about the domain. Therefore, in principle they do not need to face the labelling problem, since the extracted data represent instances of known concepts. The strong dependence on domain is the major limitation of these approaches.

7. REFERENCES

- [1] A. Arasu and H. Garcia-Molina. Extracting structured data from web pages. In *ACM SIGMOD 2003*, 2003.
- [2] D. Brin. Extracting patterns and relations from the World Wide Web. In *Proceedings of the First Workshop on the Web and Databases (WebDB'98) (in conjunction with EDBT'98)*, pages 102–108, 1998.
- [3] V. Crescenzi and G. Mecca. On automatic information extraction from large web sites. Technical Report rt-dia-76-2003, Università di Roma "Roma Tre", 2003. <http://web.dia.uniroma3.it/research/2003-76.pdf>.
- [4] V. Crescenzi, G. Mecca, and P. Merialdo. ROADRUNNER: Towards automatic data extraction from large Web sites. In *International Conf. on Very Large Data Bases (VLDB 2001), Roma, Italy, September 11-14, 2001*.
- [5] A. Doan, P. Domingos, and A. Y. Halevy. Reconciling schemas of disparate data sources: A machine-learning approach. In *SIGMOD Conference*, 2001.
- [6] D. W. Embley, D. M. Campbell, J. Y. S., S. W. Liddle, N. Y., D. Quass, and S. R. D. A conceptual-modeling approach to extracting data from the web. In *Proceedings of the 17th International Conference on Conceptual Modeling (ER'98)*, pages 78–91, 1998.
- [7] D. W. Embley, D. Jackman, and L. Xu. Multifaceted exploitation of metadata for attribute match discovery in information integration. In *Workshop on Information Integration on the Web*, pages 110–117, 2001.
- [8] D. W. Embley and J. Y. S. N. Y. Record-boundary discovery in web documents. In *ACM SIGMOD International Conf. on Management of Data*, pages 467–478, 1999.
- [9] C. Knoblock, K. Lerman, S. Minton, and M. I. Accurately and reliably extracting data from the web: A machine learning approach. *IEEE Data Engineering Bulletin*, 23(4):33–41, 2000.
- [10] S. Kuhllins and R. Tredwell. Toolkits for generating wrappers – a survey of software toolkits for automated data extraction from web sites. In *NODE02*, volume 2591 of *LNCIS*, pages 184–198, 2003.
- [11] N. Kushmerick. Wrapper induction: Efficiency and expressiveness. *Artificial Intelligence*, 118:15–68, 2000.
- [12] A. Laender, B. Ribeiro-Neto, A. Da Silva, and T. J. A. brief survey of web data extraction tools. *ACM SIGMOD Record*, 31(2), June 2002.
- [13] I. Muslea, S. Minton, and C. Knoblock. Hierarchical wrapper induction for semistructured information sources. *Journal of Autonomous Agents and Multi-Agent Systems*, 4:93–114, 2001.
- [14] J. Wang and F. Lochovsky. Data-rich section extraction from html pages. In *Proceedings of the 3rd International Conference on Web Information Systems Engineering (WISE 2002), 12-14 December 2002, Singapore, Proceedings*, pages 313–322. IEEE Computer Society, 2002.

Semantic Email: Adding Lightweight Data Manipulation Capabilities to the Email Habitat

Oren Etzioni, Alon Halevy, Henry Levy, and Luke McDowell
Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195 U.S.A
{etzioni,alon,levy,lucasm}@cs.washington.edu

ABSTRACT

The Semantic Web envisions a portion of the World Wide Web in which the underlying data is machine understandable and applications can exploit this data for improved querying, aggregation, and interaction. This paper investigates whether the same vision can be carried over to the realm of email, the adjacent information space in which we spend significant amounts of time.

We introduce a general notion of semantic email, in which email messages consist of a database query or update coupled with corresponding explanatory text. Semantic email opens the door to a wide range of automated, email-mediated applications. In particular, this paper introduces a class of *semantic email processes*. For example, consider the process of sending an email to a program committee, asking who will attend the PC dinner, automatically collecting the responses, and tallying them up. We describe a formal model where an email process is modeled as a set of updates to a data set, on which we specify certain constraints. We then describe a set of inference problems that arise in this context, and provide initial results on some of them. In particular, we show that it is possible to automatically infer which email responses are acceptable w.r.t. a set of ultimately desired constraints. Finally, we describe a first implementation of semantic email, and outline several research challenges in this realm.

Categories and Subject Descriptors

H.2.8 [Database Applications]: Miscellaneous;
F.2 [Analysis of Algorithms and Problem Complexity]: Miscellaneous;
D.4.3 [Communications Applications]: Electronic mail

Keywords

Semantic web, email, formal model, inference, ultimate satisfiability, lightweight data manipulation

1. INTRODUCTION

There is currently significant interest in making portions of the WWW machine understandable as part of the broad vision known as the “Semantic Web” [1]. While the WWW is a rich information space in which we spend significant amounts of time, many of us spend even more time on email. In contrast to the WWW, where most of our interactions involve consuming data, with email we are both creating and consuming data. With the exception of the generic header fields associated with each email message, the email information space has no semantic features whatsoever. While the majority of email will remain this way, this paper argues that adding semantic features to email offers tremendous opportunities for payoff in productivity while performing some very common tasks. To illustrate the promise, consider several examples.

- In the simplest case, suppose you send an email with a talk announcement. With appropriate semantics attached to the email, sending the announcement can also result in automatically (1) posting the announcement to a talks web site, and (2) sending a reminder the day before the talk.
- Suppose you are organizing a PC meeting, and you want to know which PC members will stay for dinner after the meeting. Currently, you need to send out the question, and compile the answers *manually*, leafing through emails one by one. Furthermore, you need to do so every few days in order to find out who has answered and who has not. With semantic email, the PC members can provide the answer in a way that can be interpreted by a program and compiled properly. In addition, after a few days, certain PC members can be reminded to answer, and those who have said they’re not coming to the PC meeting need not be bothered with this query at all.
- As a variant of the above example, suppose you are organizing a *balanced potluck*, where people should bring either an appetizer, entree or dessert, and you want to ensure that the meal is balanced. In addition to the features of the previous example, here semantic email can help ensure that the potluck is indeed balanced by examining the answers and requesting changes where necessary.
- As a final example, suppose you want to give away tickets to a concert that you cannot use. You would like to send out an announcement, and have the semantic email system give out the tickets to the first respon-

dents. When the tickets are gone, the system should respond politely to subsequent requests. Alternatively, you may want to sell the tickets to the highest bidder and have the system help you with that task.

These examples are of course illustrative rather than exhaustive. However, the examples suggest a general point: we often use email for tasks that are reminiscent of lightweight data collection, manipulation, and analysis tasks. Because email is not set up to handle these tasks effectively, accomplishing them manually can be tedious, time-consuming, and error-prone.

In general, there are at least three ways in which semantics can be used to streamline aspects of our email habitat:

1. **Update:** we can use an email message to add data to some source (e.g., a web page, as in our first example).
2. **Query:** email messages can be used to *query* other users for information. Semantics associated with such queries can then be used to automatically answer common questions (e.g., asking for my phone number or directions to my office).
3. **Process:** we can use semantic email to manage simple but time-consuming processes that we currently handle manually.

The techniques needed to support the first two uses of semantic email depend on whether the message is written in text by the user or formally generated by a program on the sender's end. In the user-generated case, we would need sophisticated methods for extracting the precise update or query from the text. In both cases, we require some methods to ensure that the sender and receiver share terminologies in a consistent fashion.

This paper focuses on the third use of semantic email to streamline processes, as we believe it has the greatest promise for increasing productivity and is where the most pain is currently being felt by users. Some hardcoded email processes, such as the meeting request feature in Outlook, invitation management via *Evite*, and contact management via *GoodContacts*, have made it into popular use already. Each of these commercial applications is limited in its scope, but validates our claim about user pain. Our goal in this paper is to sketch a *general* infrastructure for semantic email processes. Feature rich email systems such as Microsoft's Outlook/Exchange offer forms and scripting capabilities that could be used to implement some email processes. However, it is much harder for casual users to create processes using arbitrary scripts, and furthermore, the results would not have the formal properties that our model provides. We describe these properties in Section 3.

To the best of our knowledge, this paper is the first to articulate and implement a general model of *semantic email processes* (SEPs).¹ Our technical contributions are the following. We first describe a formal model for semantic email processes. The formal model specifies the meaning of semantic email processes and exposes the key implementation challenges. We then pose several fundamental inference problems that can be used to boost semantic email creation and processing, and describe some initial results on these problems. We discuss implementation issues that arise for semantic email and how we have addressed these in our first

¹ Associating semantic content with mail has been proposed before [7, 3], but such proposals have focused only on using semantics to improve mail search, sorting, and filtering.

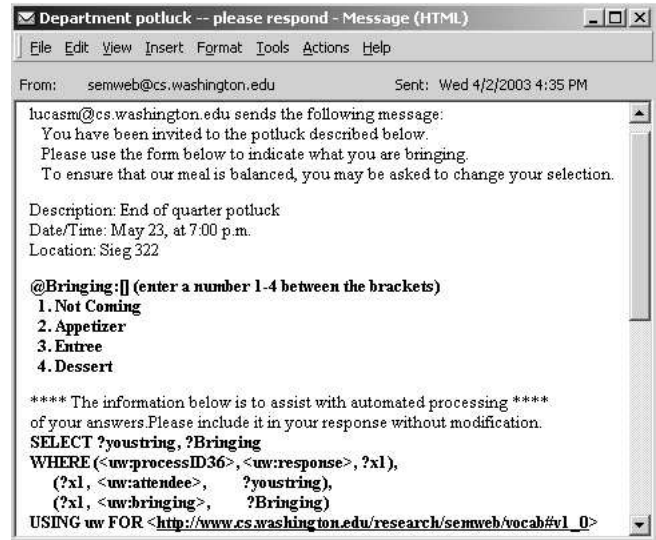


Figure 1: A message sent to recipients in a “Balanced potluck” process. The bold text at the top is a form used for human recipients to respond, while the bold text at the bottom is a query that maps their textual response to a formal language.

semantic email prototype. Finally, we outline several research challenges related to semantic email.

2. FORMAL MODEL OF SEMANTIC EMAIL PROCESSES

Our formal model of SEPs serves several goals. First, the model captures the exact meaning of semantic email and the processes that it defines. Second, the model clarifies the limitations of SEPs, thereby providing the basis for the study of different variations with varying expressive powers. Finally, given the model, we can pose several formal inference problems that can help guide the creation of semantic email processes as well as manage their life cycle. We emphasize that the users of SEPs are not expected to understand the formal model or write specifications using it.

In our model, we assume that email addresses uniquely determine individuals or sets of potential participants in the process. A SEP is initiated by an individual participant, called the *originator*. Informally speaking, the coordination of a process is achieved by a *supporting data set*, and a set of updates that the recipients of the email can make to this data set as they respond to the original email.

EXAMPLE 2.1. *We first illustrate our formal model with the example of setting up a balanced potluck. The originator of the process will initially send out a message announcing the potluck and asking everyone to say whether they are coming and whether they are bringing an appetizer, entree, or dessert (see Figure 1). As a result of this email, a supporting data set is automatically created. The data set includes a single table with a row for every recipient of the email. The table will have two columns, email and bringing. The first column will be an email address of one of the recipients, and the second column will have a constraint specifying that it can only have one of four values: { not-coming, appetizer, entree, dessert }. Initially, the second column will be set to NULL.*

The process will specify that the result of a response from a recipient p is an update to the data set, where p is allowed to update only the row where the first column is p , and the update can only modify the second column. Hence, the only thing p can do is modify NULL to one of the four allowable values.

To guarantee that the potluck is balanced, the originator will also impose a constraint on the contents of the data set. For example, the originator may specify that the difference in the number of appetizers, entrees, or desserts can be at most two.

There are still several choices that the originator needs to make. The first is whether the constraints are imposed as the responses come in (e.g., after two desserts have arrived, the system will not accept another dessert until at least one entree and one appetizer have been volunteered). Alternatively, the constraints may be imposed at the end with another series of balancing emails (and here we need to assume some cooperation on behalf of the participants). Another choice is whether to allow participants to modify their selection, and how to handle such modifications. Finally, the originator needs to decide whether and which followup messages will be sent during the process. Section 3 elaborates on some of these choices and demonstrates how inference can be used to automatically direct constraint application and message generation. \square

We now define the components of the formal model: participants, supporting data set, messages, responses, and the constraint language.

Participants: a process has an originator, p_0 , and a set of recipients, \mathcal{P} . Note that p_0 could be in \mathcal{P} .

Supporting data set: we assume that the supporting data set, D , is a set of relations. The initial contents of the relations are specified by the originator in the beginning of the process (usually to be a set of default values for the columns). With each relation in D we associate a schema that includes:

- A relation name and names, data types, and range constraints for the attributes. A special data type is `emailAddress`, whose values are the set \mathcal{P} . Attributes may also be given default values.
- For every relation in D , there may be a single attribute of type `emailAddress` that is distinguished as a *from* attribute, which means that rows in the relation whose value is p can only result from messages from the participant p . The *from* attribute may be declared unique, in which case every recipient can only affect a single row in the table.
- A set of constraints on D , denoted by C_D , specified in the constraint language we describe shortly.

Messages: Processes proceed via a set of messages M . M includes the originator's first email, asking the queries for the process. Other messages in M can either be to the recipients (or subset thereof) or to the originator. A message to the recipients is specified by:

- a time point or a triggering condition,
- a set of recipients specified by a query on D , and
- a message text, e.g., a prompt for the questions being asked by the process or a reminder to a recipient to respond to an earlier request.

A message to the originator has the first and third components above.

Responses: the set of responses to the originator's email is specified as follows:

- **Attributes:** the set of attributes in D that are affected by responses from recipients. This set of attributes *cannot* include any from attributes.
- **Insert or Update:** recipients can only add tuples, only modify tuples, or both. Recall that if there is a *from* field then all changes from p pertain only to a particular set of tuples.
- **Single or Many:** can recipients send a single response or more than one? As we explain in Section 3.1, some responses may be *rejected* by the system. By *Single*, we mean one non-rejected message.

Constraint language: The constraints C_D are specified in a language that includes conjunction and disjunction of atomic predicates. Atomic predicates compare two terms, or a term with a set. Terms have the following form:

- an attribute variable (referring to the value of a particular attribute in a row)
- a constant
- an aggregate applied to a column of a relation, or to a subset of the rows that satisfy an equality predicate (thereby obtaining the aggregate value of a single group in the relation).

We allow comparison predicates ($=$, \neq , $<$, \leq), `LIKE`, and \in , \notin between a constant and an enumerated finite set.

EXAMPLE 2.2. As described earlier, in the balanced potluck example, we will have a single table named `Potluck` with two columns: `email`, of type `emailAddress` and declared to be unique, and `bringing`, with the constraint `Potluck.bringing \in {not-coming, appetizer, entree, dessert}`. The default value for the `bringing` attribute is `NULL`.

In addition to the single column constraint, we will also specify several constraint formulas similar to the one below, specifying that the potluck should be balanced (please note that the below is not a proposal for a syntax of the constraint language so, for now, it should be taken with kindness):

$$(\text{Count} * \text{where bringing} = \text{'dessert'}) \leq (\text{Count} * \text{where bringing} = \text{'appetizer'}) + 2$$

Let us assume for now that except for the range constraint on the value of the `bringing` attribute, there are no additional constraints on responses. Furthermore, assume that recipients are allowed to send a single (acceptable) response, and hence cannot change their mind after they've committed to a dish.

Finally, the set of messages in our example includes (1) the initial message announcing the potluck and asking what each person is bringing, (2) messages informing each responder whether their response was accepted or not, (3) a reminder to those who have not responded 2 days before the potluck, (4) regular messages to the originator reporting the status of the RSVPs, and (5) a message to the originator in the event that everyone has responded. \square

3. INFERENCE FOR SEMANTIC EMAIL

Given the formal model for a SEP we can now pose a wide variety of inference problems, whose results can serve to assist in the creation and management of SEPs. The data-centric flavor of our model will enable us to bring various techniques from data management to bear on these inference problems.

The core problem we want to address using inference is whether a SEP will terminate in a *legal* state, i.e., a state that satisfies C_D . The input to the inference problem includes the constraints C_D and possibly the *current* state of D along with a response r from a recipient. The output of the inference problem is a condition that we will check on r to determine whether to *accept* r . In our discussion, we assume that r is a legal response, i.e., the values it inserts into D satisfy the range constraints on the columns of D .

The space of possible inference problems is defined by several dimensions:

- **Necessity vs. possibility:** as in modal logics for reasoning about future states of a system [14, 8, 4], one can either look for conditions that guarantee that *any* sequence of responses ends in a desired state (the \Box operator), or that it is possible that some sequence of responses brings us to a desired state (the \Diamond operator).
- **Assumptions on the recipients:** in addition to assuming that all responses are legal, we can consider other assumptions, such as: (1) *all* the recipients will respond to the message or (2) the recipients are flexible, i.e., if asked to change their response, they will cooperate.
- **The type of output condition:** at one extreme, we may want a constraint C_r that can be checked on D when a response r arrives, where C_r is specified in the same language used to specify C_D . At another extreme, we may apply an arbitrary procedure to D and r to determine whether r should be accepted. We note that a constraint C_r will inevitably be weaker than an arbitrary algorithm, because it can only inspect the state of D in very particular ways. As intermediate points we may consider constraints C_r in more expressive constraint languages. Note that in cases where we can successfully derive C_r , we can use database triggers to implement modifications to D or to indicate that r should be rejected.
- **Generation of helpful intermediate messages:** in addition to finding a condition for accepting responses, we may infer helpful messages when a response is rejected (e.g., to suggest that while a dessert could not be accepted an entree would be welcome).

As a very simple example, suppose we consider the case where we want *all* response sequences to end in a legal state, we make no assumptions on the recipients, and we are interested in deriving a constraint C_r that will be checked when a response arrives. If the initial state of D is a legal state, then simply setting C_r to be C_D provides a sufficient condition; we only let the data set D be in states that satisfy C_D . In the example of the balanced potluck, we will not accept a response with a dessert if that would lead to having 3 more desserts than entrees or appetizers.

In some cases, such a conservative strategy will be too restrictive. For example, we may want to continue accepting desserts so long as it is still *possible* to achieve a balanced potluck. This leads us to the following inference problem.

3.1 Ultimate Satisfiability

We now describe our central result concerning inference for SEPs. Our goal is to find the *weakest* condition for accepting a response from a recipient. To do that, we cut across the above dimensions as follows. Suppose we are given the data set D after 0 or more responses have been accepted, and a new response r . Note that D does not nec-

essarily satisfy C_D , either before or after accepting r . We will accept r if it is *possible* that it will lead to a state satisfying C_D (i.e., considering the \Diamond temporal operator). We do not require that the condition on r be expressed in our constraint language, but we are concerned about whether it can be efficiently verified on D and r . Furthermore, we assume that recipients can only update their (single) row, and only do so once. Hence, the columns that can be affected by the recipients start out with the NULL value, and can get assigned values in a_1, \dots, a_n .²

DEFINITION 3.1. (ultimate satisfiability) *given a data set D , a set of constraints C_D on D , and a response r , we say that D is ultimately satisfiable w.r.t. r if there exists a sequence of responses from the recipients, beginning with r , that will put D in a state that satisfies C_D .* \square

In what follows, let \mathcal{C}^\wedge be our constraint language restricted to conjunctions of atomic predicates (i.e., disjunction is not allowed, and negation can only be applied on atomic predicates). A term in a predicate of C_D may select a group of rows in an attribute A , and aggregate the value of the corresponding values in an attribute B . We say that the aggregation predicates in C_D are *separable* if whenever there is a non-COUNT aggregate over an attribute B , then B is not a grouping attribute in any term. (All of the examples given in this paper can be expressed with separable constraints.) We consider the aggregation functions MIN, MAX, COUNT, SUM, and AVERAGE.

THEOREM 3.1. *Let \mathcal{S} be a semantic email process where C_D is in the language \mathcal{C}^\wedge . Then,*

- *If the predicates in C_D are separable, then ultimate satisfiability is in polynomial time in the size of D and C_D .*
- *If the predicates in C_D are not separable, then ultimate satisfiability is NP-hard in the size of D .* \square

As an example of applying this theorem, in the balanced potluck example, suppose a new dessert response arrives. At that point, the inference procedure will (1) determine the maximal number of people who *may* come to the potluck (i.e., the number of recipients minus the number of people who replied *not-coming*), (2) check that even if the dessert response is accepted, then there are still enough people who have not answered such that the ultimate set of dishes could be balanced.

Discussion: The challenge in proving this theorem is in reasoning about the possible relationships between aggregate values (current and future), given a particular state of D . Reasoning about aggregation has received significant attention in the query optimization literature [15, 17, 9, 10, 2, 5]. This body of work considered the problem of optimizing queries with aggregation by moving predicates across query blocks, and reasoning about query containment and satisfiability for queries involving grouping and aggregation. In contrast, our result involves considering the current state of the database to determine whether it can be brought into a state that satisfies a set of constraints. Furthermore, since C_D may involve several grouping columns and aggregations, they cannot be translated into single-block SQL queries, and

²As it turns out, the case in which the recipients can add rows is actually easier, because there are less constraints on the system.

hence the containment algorithms will not carry over to our context.

To the best of our knowledge, formalisms for reasoning about workflow [13, 12] or about temporal properties of necessity and possibility have not considered reasoning about aggregation. For instance, workflow formalisms have generally been restricted to reasoning about temporal and causality constraints [16].³ One exception is the recent work of Senkul et al. [16], who expand workflows to include *resource constraints* based on aggregation. Each such constraint, however, is restricted to performing a single aggregation with no grouping (and thus could not express the potluck constraint given in Example 2.2). In addition, their solution is based upon general constraint solving and thus may take exponential time in the worst case. We’ve shown, however, that in our domain SEPs can easily express more complex aggregation constraints while maintaining polynomial inference complexity in many interesting cases.

Ultimately the problem of reasoning about semantic email will be related to the problem of reasoning about e-services; see [6] for a recent survey.

4. IMPLEMENTATION AND USABILITY ISSUES

There are several significant challenges involved in implementing semantic email and getting it adopted widely. Our formal model provides a framework for identifying these challenges. Below we discuss some of these challenges, and describe the particular choices we made in our prototype implementation.

Message handling: The first and most important challenge is how to manage the flow of messages. The problem can be divided into three: message creation, transport, and reply. For message creation, we envision a set of GUIs that guide the user through composing the message and selecting the appropriate options for the process. In addition, there will be an interface for monitoring the progress of the process. Under the covers, these interfaces can make use of the inference procedures described above.

Ideally, these GUIs would be integrated with the user’s mail client, which would then handle sending and receiving process-related mail on the user’s behalf. For replying, a recipient’s email client would present the recipient with an interface for constructing legal responses, or automatically respond to messages it knows how to handle (e.g. “Decline all tickets to the opera”). This approach, however, requires all participants in a process to install additional software (limiting its applicability) and is complicated by the variety of mail clients currently in use.

Consequently, we have adopted the following pragmatic strategy. Users originate a process by selecting a simple text form from a shared folder or a webpage, filling out the form, and then mailing that form to a special email address (e.g., `semweb@cs.washington.edu`). A central server reads this message, creates a new email process based on the parameters in the form, and sends out an initial message to the recipients. These messages also contain a simple text form that can be handled by any mail client. Recipients reply via

³Workflow formalisms could potentially convert aggregation constraints to temporal constraints by explicitly enumerating all possible data combinations, but this may result in an exponential number of states.

mail⁴ directly to the server, rather than to the originator, and the originator receives status and summary messages directly from the server when appropriate. The originator can query or alter the process via additional emails or a web interface. This approach is simple to implement, requires no software installation, and works for all email clients.⁵ We believe that divorcing the processing of semantic email (in the server) from the standard email flow (in the client) will facilitate adoption by ameliorating potential user concerns about privacy and about placing potentially buggy code in their email client.

Human/Machine Interoperability: An important requirement is that every message must contain both a human-understandable (e.g. “I’m giving away 4 opera tickets”) and an equivalent machine-understandable portion (e.g., in RDF or SQL). For messages sent to a recipient, this ensures that either a human or a machine may potentially handle and respond to the message. Thus, a process originator may send the same message to all recipients without any knowledge of their capabilities. For responses, a human-readable version provides a simple record of the response if needed for later review. In addition, a machine-understandable version enables the server to evaluate the message against the process constraints and take further action.

In our implementation, we meet this requirement by having the server attach RDF to each outgoing text message. A human responds by filling out an included text form, which is then converted into RDF at the server with a simple mapping from each field to an unbound variable in a RDQL query associated with the message.⁶ A machine can respond to the message simply by answering the query in RDF, then applying the inverse mapping in order to correctly fill out the human-readable text form.

Streamlining semantics with other aspects of email: Despite the advantages of semantic email, we do not want to create a strict dichotomy in our email habitat. In our potluck example, suppose that one of the recipients wants to know whether there is organized transportation to the potluck (and this information affects his decision on what to bring). What should he do? Compose a separate non-semantic email to the originator and respond to the semantic one only later? A better solution would be to treat both kinds of emails uniformly, and enable the recipient to ask the question in replying to the semantic email, ultimately providing the *semantic* response later on in the thread.

Our implementation supports this behavior by supplying a “remark” field in each response form, where a recipient may include a question or comment to be forwarded to the originator. For a question, the originator can reply, enabling

⁴Alternatively, we could send recipients a link to a suitable web-based form to use for their response. This mechanism is fully consistent with our formal model and has some advantages (e.g., forms are not restricted to text, immediate data validation). We chose instead to use email because we feel it fits more naturally with how recipients typically handle incoming messages.

⁵This approach can still enable the automated answering of common questions, but requires an extra step for recipients to forward such messages to the server

⁶Technically, this response does not contain a pure machine-understandable portion (e.g. in raw RDF), but does contain all the information necessary (fields, RDQL, and mapping) to easily produce this portion.

the recipient to respond to the original semantic question with the included form, or to pose another question.

Database and inference engine: There are several system issues to consider, such as where does the actual database and inference engine reside. Right now, we built a separate semantic email system implemented as a centralized server supported by a relational database.

Implementation Status: We have developed a prototype semantic email system and deployed it for public use.⁷ So far we have developed simple processes to perform functions like collecting RSVPs, giving tickets away, and organizing a balanced potluck; these can be customized for many other purposes (e.g. to collect N volunteers instead of give away N tickets). We are currently testing the system with real users to determine how well these processes generalize to everyday needs.

The prototype is integrated with our larger MANGROVE [11] semantic web system. This provides us with an RDF-based infrastructure for managing email data and integrating with web-based data sources and services. For instance, the MANGROVE calendar service accepts event information via email or from a web page. Future work will consider additional ways to synergistically leverage data from both the web and email worlds in this system.

5. CONCLUSIONS

We have introduced a paradigm for enriching our email habitat with the ability to perform lightweight data collection and manipulation tasks. We believe that this form of semantic email has the potential to offer significant productivity gains on email-mediated tasks that are currently performed manually in a tedious, time consuming, and error-prone manner. Moreover, semantic email opens the way to scaling similar tasks to large numbers of people in a manner that is not feasible with today's person-processed email. For example, large organizations could conduct surveys and voting via email with guarantees on the behavior of these processes. We motivated semantic email with several examples, presented a formal model that teases out the issues involved, and used this model to explore several important inference questions. Finally, we described our publicly accessible prototype implementation.

This short paper focused on a particular class of semantic email processes (SEPs) that can be described with a simple constraint language. However, our notion of semantic email is substantially broader, which suggests several research challenges:

- Our formal analysis considered a point in the space of semantic email processes, which led to our central theorem. It would be worthwhile to carefully analyze the remaining points in the space described in Section 3.
- In our analysis we adopted a limited constraint language — how does increasing its expressive power impact the tractability of inference?
- We considered a small set of illustrative examples. Future work will explore additional tasks and investigate any impediments to widespread adoption.
- SEPs are only one instantiation of semantic email, which far from exhausts its potential. New ways to leverage semantic email seem like a promising direction for future work.

Finally, we see semantic email as a first step in a tighter integration of the web (semantic or not) and email, the two information spaces in which many of us spend the bulk of our online time.

6. ACKNOWLEDGMENTS

This research was partially supported by NSF ITR Grant IIS-0205635, by NSF CAREER Grant IIS-9985114 for Alon Halevy, and by a NSF Graduate Research Fellowship for Luke McDowell.

7. REFERENCES

- [1] T. Berners-Lee, J. Hendler, and O. Lassila. The semantic web. *Scientific American*, May 2001.
- [2] S. Cohen, W. Nutt, and A. Serebrenik. Rewriting aggregate queries using views. In *Proc. of PODS*, pages 155–166, 1999.
- [3] D. Connolly. A knowledge base about internet mail. <http://www.w3.org/2000/04/mailllog2rdf/email.html>.
- [4] R. Fagin, J. Y. Halpern, Y. Moses, and M. Y. Vardi. *Reasoning About Knowledge*. M.I.T Press, 1995.
- [5] S. Grumbach and L. Tininini. On the content of materialized aggregate views. In *Proc. of PODS*, 2000.
- [6] R. Hull, M. Benedikt, V. Christophides, and J. Su. E-Services: A look behind the curtain. In *PODS*, 2003.
- [7] A. Kalyanpur, B. Parsia, J. Hendler, and J. Golbeck. SMORE - semantic markup, ontology, and RDF editor. <http://www.mindswap.org/papers/>.
- [8] R. E. Ladner. The computational complexity of provability in systems of model propositional logic. *SIAM Journal on Computing*, 6(3):467–480, 1977.
- [9] A. Y. Levy and I. S. Mumick. Reasoning with aggregation constraints. In *Proc. of EDBT*, Avignon, France, March 1996.
- [10] A. Y. Levy, I. S. Mumick, and Y. Sagiv. Query optimization by predicate move-around. In *Proc. of VLDB*, pages 96–107, Santiago, Chile, 1994.
- [11] L. McDowell, O. Etzioni, S. D. Gribble, A. Halevy, H. Levy, W. Pentney, D. Verma, and S. Vlasheva. Evolving the semantic web with Mangrove. Technical Report UW-CSE-03-02-01, February 2003.
- [12] C. Mohan. Workflow management in the internet age. www.almaden.ibm.com/u/mohan/workflow.pdf, 1999.
- [13] S. Mukherjee, H. Davulcu, M. Kifer, P. Senkul, and G. Yang. Logic based approaches to workflow modeling and verification. In *Logics for Emerging Applications of Databases*, 2003.
- [14] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual IEEE Symposium on Foundations of Computer Science*, pages 46–57, 1977.
- [15] K. Ross, D. Srivastava, P. Stuckey, and S. Sudarshan. Foundations of aggregation constraints. In A. Borning, editor, *Principles and Practice of Constraint Programming*. Lecture Notes in Computer Science, 874. Springer Verlag, 1994.
- [16] P. Senkul, M. Kifer, and I. H. Toroslu. A logical framework for scheduling workflows under resource allocation constraints. In *VLDB*, 2002.
- [17] D. Srivastava, S. Dar, H. V. Jagadish, and A. Y. Levy. Answering SQL queries using materialized views. In *Proc. of VLDB*, Bombay, India, 1996.

⁷ Accessible at www.cs.washington.edu/research/semweb/email

XML Interoperability

Laks V. S. Lakshmanan^{*}
Department of Computer Science
University of British Columbia
Vancouver, BC, Canada
laks@cs.ubc.ca

Fereidoon Sadri[†]
Department of Mathematical Sciences
University of North Carolina at Greensboro
Greensboro, NC, USA
sadri@uncg.edu

ABSTRACT

We study the problem of interoperability among XML data sources. We propose a lightweight infrastructure for this purpose that derives its inspiration from the recent semantic web initiative. Our approach does not require either *source-to-source* or *source-to-global* mappings. Instead, it is based on enriching local sources with semantic declarations so as to enable interoperability. These declarations expose the semantics of the information content of sources by mapping the concepts present therein to a common (application specific) vocabulary, in the spirit of RDF. In addition to this infrastructure, we discuss tools that may assist in generating semantic declarations, and formulation of global queries and address some interesting issues in query processing and optimization.

1. INTRODUCTION

Interoperability and data integration are long standing open problems with extensive research literature. Much of the work in the context of federated databases focused on integrating schemas by defining a global schema in an expressive data model and defining mappings from local schemas to the global one. More recently, in the context of integration of data sources on the internet, the so-called global-as-view and local-as-view paradigms have emerged out of projects such as TSIMMIS [20] and Information Manifold (IM) [12]. All of these have primarily concerned themselves with relational abstractions of data sources. Recently, the advent of XML as a standard for online data interchange has much promise toward promoting interoperability and data integration. But XML, being a syntactic model, in itself cannot make interoperability happen automatically. Two main challenges to overcome are: (i) data sources may model XML data in heterogeneous ways, e.g., using different nestings or groupings or interchanging elements and attributes, and (ii) sources

*Supported by NSERC.

[†]Supported by NSF grant IIS-0083312.

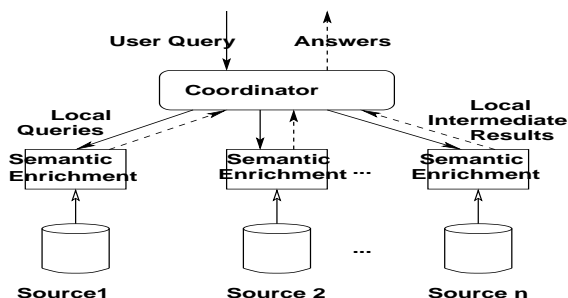


Figure 1: An architecture for semantic markup and query processing.

may employ different terminology, a classic problem even in multi-database interoperability. While earlier approaches to integration can be extended to handle XML, they suffer from the shortcoming of requiring a commonly agreed upon global schema. Can we interoperate without a global schema?

Indeed, there are a few recent proposals that do overcome the need for a global schema – Halevy et al. [7] and Miller et al. [16]. We will discuss them in detail in Section 5. In a nutshell, both of these approaches rely on source to source mappings. One problem here is that requiring such mappings for all pairs is too tedious and cumbersome. In order to mitigate this, one can merely insist, as [7] does, that the graph of pairs with available mappings be connected. A second problem is that when a source s_i is mapped to source s_j , if s_j does not have some of the concepts present in s_i , then they will be lost. E.g., s_i may include the ISBN for all its books while s_j may not.

Independently of all this, there has been a lot of recent excitement around the *semantic web* initiative, coming with its own host of technologies such as resource description framework (RDF) and ontology description languages such as DAML, OWL, and XTM [19]. The promise of semantic web is to expose the semantics of the information content of web resources (including text, audio, video, etc.) using common transparent vocabulary thus taking the web to a higher semantic level, enabling easy exchange of data and applications. Can we leverage these developments and create a lightweight scalable infrastructure for interoperability? We believe the answer is yes and take the first steps toward this creation here.

Our thesis is that each source(’s administrator) interested in participating in data and application sharing should “en-

rich” itself with adequate semantic declarations. These declarations would essentially expose the semantic concepts present in the source using common, but possibly application specific, vocabulary, in the spirit of ontologies, expressed in a framework such as RDF, as part of the semantic web initiative. Queries across data sources so enriched can be composed over this common vocabulary, being agnostic to the specific modeling constructs or terminologies employed in the local sources. A detailed example motivating these issues appears in Section 2. A suggested architecture for semantic enrichment and query processing is depicted in Figure 1. Other architectures such as peer to peer are also possible. We make the following contributions in this paper.

- We describe a lightweight infrastructure based on local semantic declarations for enabling interoperability across data sources (Section 3).
- These declarations are mapping rules that map data items in a source to a common vocabulary, inspired by RDF. Checking the validity of these mappings involves checking certain key constraints. We illustrate this issue and show how this checking can be done in general (Section 3).
- A user query, when translated against the data sources will in general lead to exponentially many inter-source queries and a linear number of intra-source queries. The former are more expensive and their elimination and optimization is an important problem. We show under what conditions inter-source queries can be eliminated altogether. We also develop a generic technique for optimizing inter-source queries (Section 4).

Proofs are given in the full paper [10]. Optimizing inter-source queries requires us to infer key and foreign key constraints that hold for the predicates defining the vocabulary, given the source schema (e.g., XML schema or DTD) and the mappings. While previous work has addressed the derivation of key constraints, we develop an algorithm for inferring foreign key constraints. Due to space restrictions we could not include this algorithm in this paper. Interested readers are referred to the full paper [10].

2. A MOTIVATING EXAMPLE

Consider a federation of catalog sales stores maintaining their merchandise information as shown in Figure 2.

Consider the query “For each state, list the state information and all (distinct) items available in warehouses in that state.” How can we express this query? There are two sources of complexity in writing this query (say in XQuery):

- For each source, the user has to write source specific “access code” involving XPath expression specific to it.
- The query, in general, is *not* the union of three intra-source queries, as we must consider combinations of state/warehouse pairs in one source with warehouse/item pairs in others. As a result, the resulting query is a complex union of a large number of “joins” across sources in addition to the three intra-source queries.

In addition to the obvious burden on the user for composing such queries, without further knowledge, they are hard to optimize. *Note that even if we were to employ source-source mapping techniques of [7] or [16], the second of these*

difficulties does not go away. The reason is that not every concept present in a source may have a counterpart in the source it is mapped to.

What we would ideally like is for the user to be able to write such queries easily, preferably by referring to some common vocabulary infrastructure created for the application on hand. The infrastructure should alleviate the burden on the user of source specific access code writing. Besides, we want the query expression to be simple to compose and to comprehend. Finally, we would like query optimization to be handled by the system.

The main idea behind our approach is based on the observation that the resource description framework (RDF) [18] is a mechanism for specifying metadata about a source that includes, not only metadata such as the author and creation date, but also the semantic concepts present therein. RDF normally employs predicates over subjects and values for describing such concepts. E.g., suppose we have the following common vocabulary (predicates) defined for describing catalog sales applications:

```
item-name, item-description, item-warehouse,
warehouse-city, warehouse-state
```

Each predicate takes two arguments where the arguments are required to play specific roles. E.g., `item-name` takes two arguments where the first argument should be an identifier of an item whereas the second should be an identifier of a name, or a literal name string itself. The roles of the (two) arguments of the other predicates is self-evident from the predicate names. This is in line with RDF convention where a URI is associated with each subject, property (predicate), and value (or object). For lack of space, we do not show the RDF specs. In fact, in the spirit of semantic web, we may expect that in addition to the semantic marking up of each source, there may be additional ontological mappings (expressed in languages such as OWL [15]) that relate concepts in a class hierarchy as well as specify additional constraints on them. In this paper, we do not consider ontologies. A final point is that an RDF-based semantic marking up for a large source can be tedious. Furthermore, storing the RDF marking up explicitly is redundant and wastes considerable space. To address this concern, we can: (i) write a transformation program in, say XSLT that transforms an XML document into the required RDF markup specs, and (ii) make use of tools that assist in the writing of the XSLT program.

We expect users (local source administrators) will create such transformations or mappings with the aid of tools. In the paper, for simplicity, in place of the complex syntax of XSLT, we use a fragment of XPath together with predicates to express the mappings. An example mapping program for the data sources of Figure 2 appears in Figure 3.

Global queries can now be formulated using these vocabulary predicates. No knowledge of local sources is needed for this task.

EXAMPLE 2.1 (A GLOBAL QUERY). Revisit the query “For each state, list the state information and all (distinct) items available in warehouses in that state.” For simplicity, we assume different sources are using the same domain for item ids (such as the manufacturer’s item id)¹. We can formulate this query as follows:

¹In general, local sources may use their own id domains.

<pre> <!ELEMENT store (warehouse*)> <!ELEMENT warehouse (city, state, item*)> <!ELEMENT item (id, name, description)> <!--ATTLIST warehouse id ID #REQUIRED--> </pre>	<pre> <!ELEMENT store (items, warehouses)> <!ELEMENT items (item*)> <!--ELEMENT item (id, name, description)--> <!--ELEMENT warehouses (warehouse*)--> <!--ELEMENT warehouse (city, state)--> <!--ATTLIST item warehouse-id IDREFS #REQUIRED--> <!--ATTLIST warehouse id ID #REQUIRED--> </pre>	<pre> <!--ELEMENT store (items,warehouses,inventory)--> <!--ELEMENT items (i-tuple*)--> <!--ELEMENT i-tuple (id,name,description)--> <!--ELEMENT warehouses (w-tuple*)--> <!--ELEMENT w-tuple (id,city,state)--> <!--ELEMENT inventory (inv-tuple*)--> <!--ELEMENT inv-tuple (item-id,warehouse-id)--> </pre>
DTD: Source1	DTD: Source2	(Relational) DTD: Source3

Figure 2: Schema of three sources.

```

item-name($I,$N) <- source1/store/warehouse/item $X, $X/id $I, $X/name $N.
item-warehouse($I,$W) <- source1/store/warehouse $X, $X/item/id $I, $X/@id $W.
warehouse-state($W,$S) <- source1/store/warehouse $X, $X/@id $W, $X/state $S.

item-name($I,$N) <- source2/store/items/item $X, $X/id $I, $X/name $N.
item-warehouse($I,$W) <- source2/store/items/item $X, $X/id $I, $X/@warehouse-id $W.
warehouse-state($W,$S) <- source2/store/warehouses/warehouse $X, $X/@id $W, $X/state $S.

item-name($I,$N) <- source3/store/items/i-tuple $X, $X/id $I, $X/name $N.
item-warehouse($I,$W) <- source3/store/inventory/inv-tuple $X, $X/item-id $I, $X/warehouse-id $W.
warehouse-state($W,$S) <- source3/store/warehouses/w-tuple $X, $X/id $W, $X/state $S.

```

Figure 3: Mappings.

```

for $S in distinct-values(warehouse-state/tuple/state)
return <state> {$S}
  for $X in warehouse-state/tuple[state=$S]
    $Y in item-warehouse/tuple[warehouse=$X/warehouse]
  return
    <item> <id> {distinct($Y/item)} </id> <item>
    </state>

```

Note that the only information needed to formulate this query is the knowledge that the federation contains the vocabulary predicates **item-warehouse** listing items and the warehouses for each item, and **warehouse-state** listing warehouses and their respective states. Also note that, in general, each source in the federation may have none, some, or all of the predicates specified in the global query.

The rest, optimization and execution of query, is the responsibility of the system. We will discuss query optimization and processing in Section 4. ■

3. LOCAL SOURCE DECLARATIONS

The main idea behind our approach is to provide a simple yet powerful model for local information sources that makes interoperability possible. Any approach to interoperability and integration is ultimately dependent on standards and common vocabularies (or higher level declarations that relate vocabularies to each other). In our approach, we assume the availability of *application specific standard ontologies*. Such an ontology specifies a number of simple classes and predicates (often called *concepts* and *properties*) that are needed for the modeling of the application.² For example, in our simple catalog stores application, the relevant standard predicates were **item-name**,

In such cases, a mapping (for example between local item ids and manufacturer's item ids) should be available to relate different id domains in order to make interoperability possible.

²In addition, an ontology could state axioms involving relationships between classes and between predicates. For brevity, we do not discuss these in this paper.

item-description, **item-warehouse**, **warehouse-city**, and **warehouse-state**.

Where do ontologies come from? And aren't they a little like a global schema which we wanted not to have to assume? Firstly, currently there are several efforts underway toward developing just such ontologies [3] (also see the bibliography ontology in [13]). We envisage that groups of parties engaged in specific applications will converge to standard ontologies. Secondly, unlike the problem with global schema, we are *not* attempting to determine a commonly agreed upon global schema into which local source schemas can be mapped. Rather, the idea is that whenever a source/site wants to join an interoperability effort, it must map its data to the standard ontology that already exists. In other words, we get the global schema for free, as long as an application specific standard ontology has been developed for it.

A standard ontology lists:

- Predicate names and usages.
- Predicate arguments; type and *role* of each argument. For example, the predicate **item-warehouse** takes two arguments. The first argument should be an identifier for an item. The second argument should be an identifier for a warehouse.
- Information about identifiers. For example, for a predicate **person-name** where the first argument is an identifier for person, the ontology contains information on possible identifiers for people, such as, *ssn* (*social security number* in the US), *sin* (*social insurance number* in Canada), etc.
- Additional type information for arguments. For example, in the predicate **person-salary**, the ontology specifies that the type information for salary must specify additional information such as currency, and period (*e.g.* annual, monthly, bi-weekly).

To provide maximum flexibility, standard predicates should be the simplest possible. Each predicate defines a relation-

ship between a number of concepts, and is not divisible further into simpler predicates. Hence, following RDF, we expect the ontology will mainly employ simple binary predicates.

A local source exists in its native model (XML data, relational, spreadsheet, etc.). To make interoperability possible, the user (or local database administrator) should (1) choose the set of standard predicates that model the source contents from the relevant ontology, and (2) provide *mappings* (or transformation rules) that map source data to the standard predicates.

3.1 Mappings

Once the set of predicates for a source has been determined, mappings from XML data (or other source types) to predicates should be established. The language we use to specify XML-to-predicate mappings is based on a (subset of) XPath and is similar to mapping languages (also called *transformation rules*) in the literature (For example, [6]). The mapping for a binary predicate p has the following form

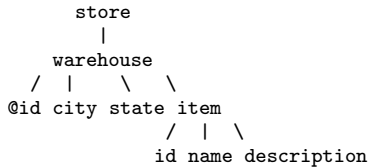
```
 $p(\$X, \$Y) \leftarrow \text{path1 } \$G, \$G/\text{path2 } \$X, \$G/\text{path3 } \$Y.$ 
```

where $\$X$ and $\$Y$ correspond to the arguments of p . $\$G$ is called the *glue* variable, and is used to restrict the $(\$X, \$Y)$ pairs to have the same $\$G$ ancestor element in the document.

We envision tools to assist users (or database administrators of local sources) in defining the mappings. The graphical interface displays the structure of the XML document as a graph. For a simple binary predicate, three nodes in the XML schema graph should be selected by the user to determine (1) and (2): Variables associated with each of the arguments, respectively, and (3): The glue variable. Normally, the node specifying the glue variable is the least common ancestor of the first two nodes that specify the arguments. Sometimes, gluing may be accomplished by equating values of elements related to the first two argument elements.

EXAMPLE 3.1 (MAPPING TOOL). In the schema graph of source 1 (below) user selects the three nodes `id`, `@id`, and `warehouse` for argument1, argument2, and glue, respectively. The following mapping is then generated by the tool.

```
item-warehouse($I,$W)<-source1/store/warehouse $X,  
                        $X/item/id $I, $X/@id $W
```



The mapping tool can also assist the user by performing certain *type* and *role* checking on the arguments of the predicate, and verifying or deriving integrity constraints for the predicates. The extent to which the tool can succeed in these tasks depends on the richness of local source (native) declarations of types and constraints. For example, a rich declaration for XML data would use a powerful schema declaration language (such as XML Schema) and contain elaborate integrity constraints satisfied by source data.

EXAMPLE 3.2. In the predicate `warehouse-state`, the ontology may specify that the first argument should be an identifier for warehouses, and should be the key for the predicate. These requirements can be verified by the tool if adequate declarations exist for the sources. For example, in the DTD of source 1 the `@id` attribute of `warehouse` is of type `ID`, hence it is an identifier for `warehouse`. Alternatively, if XML Schema was used, the requirements could be derived from the following constraint³

$$(\epsilon, (\text{source1/store/warehouse}, (\{\text{@id}\})))$$

using techniques of [6]. ■

We have studied the problem of deriving foreign key constraints (or referential integrity constraints) for the predicates from XML declarations and mapping rules [10]. These constraints play a very important role in query optimization in this approach. Davidson et al. [6] have studied the problem of deriving or verifying functional dependencies from XML key constraints. Their algorithms can be used in the mapping tool for the verification of mappings correctness by checking argument roles and constraints that should be satisfied by the predicates.

4. QUERY TRANSLATION, PROCESSING, AND OPTIMIZATION

Our simple model of (mostly binary) predicates significantly simplifies the task of query formulation. The federation consists of local sources, where the content of each source has been declared by a set of standard predicates and mappings to them. The global content is the collection of all local predicates. Given a predicate p , a source i may contain a fragment of p , which we denote by p_i . In general, fragments of the same predicate at different sources may contain overlapping data. We refer to the (conceptual) union of all fragments p_i of a predicate p as the *global* predicate p .

A query is formulated in terms of global predicates and predicate fragments. The query processor architecture is shown in Figure 1. A naive implementation approach is to materialize these predicates at the coordinator using the local mapping specifications, and then execute the query using materialized predicates. This approach will be very inefficient in general.

A better approach involves expanding the query by replacing each global predicate by the union of its fragments. The outcome, in general, consists of a collection of *local* (i.e., intra-source) and *inter-source* queries, the results of which should be combined to obtain the answer to the query. The coordinator is in charge of query expansion, query preparation and transformation (to local native schemata), and coordination of sub-query execution at different sources. It then collects results of the sub-queries and combines them to form the final answer.

A local sub-query is one where all the predicate fragments specified in the query belong to the same source. A local query can be rewritten in the native model of the source, and executed locally at that source. We will demonstrate this with an example below.

³Which says in the context of the whole document, attribute `@id` uniquely identifies a warehouse.

In an inter-source query, fragments from different sources are specified. We present various techniques to the translation, optimization, and processing of inter-source queries (Section 4.2). Processing these queries normally involves data transmission between sources so optimizing inter-source queries is important.

EXAMPLE 4.1 (A LOCAL QUERY). Let's consider the global query Q of Example 2.1. The local sub-query obtained from Q for source 1 is obtained by substituting predicate fragments `warehouse-state-1` and `item-warehouse-1` for the corresponding predicates in Q .

The local query is then transformed by substituting its variable declarations by variable declarations obtained from mappings for source 1 (see Section 2 for mappings):

```
for $S in distinct-values(source1/store/warehouse/state)
return <state> {$S}
  for $Z in source1/store/warehouse[state=$S],
    $W in source1/store/warehouse[@id=$Z/@id]
  return
    <item> <id> {distinct($W/item/id)} </id> <item>
    </state>
```

The last step for local queries is native query optimization, which may be carried out by the coordinator, the source, or both. In this example, the optimizer realizes $\$Z$ and $\$W$ are the same, and eliminates one to get the final local query to be executed by the local source. ■

4.1 When can Inter-source Processing be Eliminated?

A global query often involves more than one predicate. If there are n sources and the query involves m predicates, then the expansion of the query results in $O(n^m)$ queries. Only $O(n)$ of these queries are local queries. The rest are expensive inter-source queries.

We are interested in characterizing cases where inter-source queries can be eliminated. If we can eliminate all inter-source queries, then we have reduced the number of queries to $O(n)$, instead of $O(n^m)$. In the remainder of this section we will study a simple case where the global query involves the joining of two binary predicates. Results of this section can be extended to more general cases.

Let the global query involve the join of predicates $p(A, B)$ and $q(B, C)$ on argument B . Our Example 2.1 was of this type, joining `item-warehouse` and `warehouse-state` on the `warehouse` argument. In general, predicate fragments from different sources contain overlapping data. Often a **distinct** clause must be used in the query to eliminate multiplicities that result from this data duplication.

Consistency Condition: We say a predicate p satisfies the *consistency condition*, provided: (i) whenever a constraint $p(\alpha) \rightarrow p(\beta)$ holds in p , then it holds for every fragment p_i ; (ii) if for fragments p_i and p_j we have $t_i \in p_i$, $t_j \in p_j$, and $t_i(\alpha) = t_j(\alpha)$, then $t_i(\beta) = t_j(\beta)$.

Condition (i) states that when the semantics of a predicate p dictates that a constraint (key or functional dependency) should hold in p , then all fragments should satisfy this constraint. Further, by Condition (ii), data in different fragments should be consistent in the sense that two facts (or tuples) in different fragments with the same α component should also have the same β component when $p(\alpha) \rightarrow p(\beta)$ holds.

The following theorem characterizes the cases where inter-source joins are not needed. Let Q be a query involving the join of $p(A, B)$ and $q(B, C)$, and Q also has the **distinct** clause to eliminate duplicates. Let k be a source.

THEOREM 4.1. No inter-source processing involving p_k is needed if and only if the following conditions hold: (1) The key constraint $B \rightarrow C$, and the consistency condition hold for q , and (2) There is a foreign key constraint from p_k to q_k on the attribute B . ■

Proof is given in the full paper. Note that in this theorem we are assuming *only local information in the form of key and foreign key constraints, plus a consistency condition which is basically a statement of correctness of data across sources*. If other forms of constraints are permitted, the theorem can be extended by relaxing the conditions. An extension to this theorem is also given in the full paper.

4.2 Efficient Inter-source Processing (When Needed)

Let's consider a query involving the join of predicate p and q , and concentrate on the inter-source processing of $p_i \bowtie q_j$, $i \neq j$. We will discuss several approaches. The optimization process, in general, will need a *cost-based* approach which estimates the cost of different approaches and selects the best.

A Naive Algorithm. A naive approach for calculating $p_i \bowtie q_j$ is to ship one complete document to the other site, and then proceed as before by transforming variable declarations of the query to variable declarations on the documents (that are now locally available) and executing the query locally.

Algorithm 2. A more promising approach for calculating $p_i \bowtie q_j$, $i \neq j$ is to first materialize one of the predicates locally and send it to the other site. Then only transform query variable declarations on the second predicate to its corresponding document, and execute the query (using the first predicate and the second document). For example, p_i can be materialized at site i and shipped to site j . Then query is transformed to one that uses predicate p_i and the document at site j . This approach may be more efficient than the first approach since the volume of data transferred may be significantly lower (predicate p_i versus the whole document at site i). ■

If certain conditions hold, we are able to use the *semi anti-join* approach discussed below in the calculation of inter-source joins to avoid generation of duplicate answers, and thereby further optimize query processing.

Assume the global query involves join of predicates $p(A, B)$ and $q(B, C)$. Conditions under which semi anti-join approach can be used in the calculation of inter-source joins are: (1) The global query has a **distinct** qualifier, and (2) The key constraints $A \rightarrow B$ and $B \rightarrow C$ and the consistency condition hold for p and q .

The semi anti-join technique: When calculating inter-source join $p_i \bowtie q_j$ as a step towards calculating $p \bowtie q$, proceed as follows:

- (1) Let the partial result of $p \bowtie q$ calculated so far be r . Ship $r[A]$ to site i .
- (2) At site i , generate $p'_i = \{t \mid t \in p_i, t(A) \notin r[A]\}$.
- (3) Ship p'_i to site j .

- (4) Calculate $p'_i \bowtie q_j$.
- (5) Combine the result with r to obtain the updated partial result.

In the full paper [10], we present a variation on the semi anti-join algorithm that may be more efficient in some cases.

5. RELATED WORK

Data integration has received significant attention since the early days of databases. Most approaches to data integration involve deriving a global schema, which integrates local schemata. Then mappings are provided between the global schema and local schemata. There are two general approaches, *local as view* where local sources are considered as views of the global; and *global as view* where rules are provided to derive global data from local sources. There are a number of excellent surveys and discussions on data integration and related topics such as [4, 8, 9, 11, 17].

A more recent approach attempts at providing data sharing by mapping sources to each other. These approaches eliminate the need for deriving the global schema, and hence eliminate a substantial overhead in system design and creation. The approach of [7] is particularly attractive as it does not require schema mappings between every pairs of sources. Rather, schema mappings can be *composed* to derive new mappings. Hence, it suffices for a source to provide a mapping to one other source, preferably the one closest to its structure. Every source-pair need to be connected by a sequence of these peer-to-peer mappings. This approach works well when sources have more or less the same information content. Otherwise some concepts in a source may not have counterparts in another source, and the mapping will lose these concepts. The loss will be further propagated by mapping compositions. The Clio project has made significant contributions to the problems of data and schema integration from various source types (legacy, relational, object-oriented, semistructured) [2, 14, 16]. Their approach is based on providing tools to map any two schemas to each other, and hence it can be used for source-source as well as source-global approaches. Amnan et al. [1] propose mappings of XML sources into a global schema using the local as view approach and propose algorithms for query rewriting and optimization.

Our approach differs from most of the previous approaches in that we eliminate the need for source-source or source-global mappings. Instead, we provide simple but powerful declarations that are local to each source. Global interoperability is then obtained by using the standard vocabulary predicates declared for each source.

Effective query optimization in our approach depends on rich schema and consistency rules declaration for local sources, and the ability to derive key and foreign key (or referential integrity) constraints for the standard predicates. There has been a lot of recent interest in the declaration of constraints for XML data, and derivation of constraints when XML data is stored as relational. Buneman et al. study the question of constraints declaration for XML [5]. The problem of propagating XML key constraints to relational is studied in [6]. They provide algorithms to determine, given XML key constraints and XML to relational transformation rules, whether a given FD is implied by the XML constraints. We utilize these results, as well as our results

reported in the full paper [10], to derive key and foreign key constraints that are used in query optimization.

6. CONCLUSION

The vision behind our project is to create a lightweight scalable infrastructure for interoperating over XML data sources. Thereto, we presented an approach that derives its inspiration from the semantic web initiative, specifically RDF, to semantically mark up the information contents of data sources using application specific common vocabulary, by means of mapping rules in a language like XSLT. User queries are expressed over this vocabulary and are translated into source specific access code and optimized. We addressed issues of eliminating or optimizing the costly inter-source queries. The full paper addresses inference of integrity constraints over the vocabulary predicates, across the mapping rules. We plan to implement these ideas in a prototype system. Our ongoing work addresses various exciting technical challenges arising from this preliminary investigation.

7. REFERENCES

- [1] B. Amnan et al. Querying XML sources using an ontology-based mediator. *CoopIS 2002*, pp. 429-448.
- [2] P. Andritsos et al. Schema management. *IEEE Data Engineering Bulletin*, 25(3), 2002.
- [3] T. Berners-Lee and E. Miller. The semantic web lifts off. *ERCIM News*, 51, pp. 9-11, 2002.
- [4] P. A. Bernstein. Applying model management to classical meta data problems. *CIDR 2003*.
- [5] P. Buneman et al. Keys for XML. *Computer Networks*, 39(5):473-487, 2002.
- [6] S. Davidson et al. Propagating XML constraints to relations. *ICDE 2003*.
- [7] A. Halevy et al. Crossing the structure chasm. *CIDR 2003*.
- [8] A. Y. Halevy. Answering queries using views. *The VLDB Journal*, 10(4):270-294, 2001.
- [9] *IEEE Data Engineering Bulletin*, 25(3), 2002. Special issue on Integration Management.
- [10] L. V. S. Lakshaman and F. Sadri. XML interoperability. <http://www.uncg.edu/~sadrif/papers/full-xmlinterop.pdf>.
- [11] M. Lenzerini. Data integration: A theoretical perspective. *PODS 2002*, pp. 233-246.
- [12] A. Y. Levy et al. Querying heterogeneous information sources using source descriptions. *VLDB 1996*, pp. 251-262.
- [13] E. Mena. <http://sol1.cps.unizar.es:5080/OBSERVER/ontologies.html>.
- [14] R. J. Miller et al. The Clio project: Managing heterogeneity. *SIGMOD Record*, 30(1), 2001.
- [15] Web Ontology Language (OWL), Reference Version 1.0, February 2003, <http://www.w3.org/TR/owl-ref/>.
- [16] L. Popa et al. Translating web data. *VLDB 2002*.
- [17] E. Rahm and P. A. Bernstein. A survey of approaches to automatic schema matching. *The VLDB Journal*, 10(4):334-350, 2001.
- [18] Resource Description Framework (RDF) <http://www.w3.org/rdf/>.
- [19] Semantic Web <http://www.w3.org/2001/sw/>.
- [20] The TSIMMIS project home page. <http://www-db.stanford.edu/tsimmis/tsimmis.html>.

Building Data Integration Systems via Mass Collaboration

Robert McCann, AnHai Doan, Vanitha Varadarajan, Alexander Kramnik

Department of Computer Science
University of Illinois, Urbana-Champaign, IL 61801, USA
{rlmccann, anhai, varadara, kramnik}@cs.uiuc.edu

ABSTRACT

Building data integration systems today is largely done by hand, in a very labor-intensive and error-prone process. In this paper we describe a conceptually new solution to this problem: that of *mass collaboration*. The basic idea is to think about a data integration system as having a finite set of parameters whose values must be set. To build such a system the system administrators construct and deploy a system “shell”, then ask the users to help the system “automatically converge” to the correct parameter values. This way the enormous burden of system development is lifted from the administrators and spread “thinly” over a multitude of users. We describe our current effort in applying this approach to the problem of schema matching in the context of data integration. We present experiments with both real and synthetic users that show the promise of the approach. Finally we discuss the future work, challenges, and the potential applications of the approach beyond the data integration context.

1. INTRODUCTION

The rapid growth of distributed data on the Web and at enterprises has generated much interest in building data integration systems. Figure 1 shows a data integration system over several sources that list books for sale. Given a user query that is formulated in the query interface (i.e., the *mediated schema*), the system uses a set of *semantic mappings* to translate the query into queries over *source schemas*. Those queries are then executed and the combined results are returned to the user.

Numerous works have been conducted on data integration, both in the database and AI communities (e.g., [6, 11, 7, 8, 3, 2, 9]). Substantial progress has been made in developing conceptual and algorithmic frameworks, query optimization, schema matching, wrapper construction, object matching, and fielding data integration systems on the Web.

Despite this progress, however, today building data integration systems is still largely done by hand in an extremely labor-intensive and error-prone process. The advent of languages and mediums for creating and exchanging semi-structured data, such as XML, OWL, and the Semantic Web, will further accelerate the needs for data integration systems and exacerbate the above problem. Thus it has now become critical to develop techniques that enable the

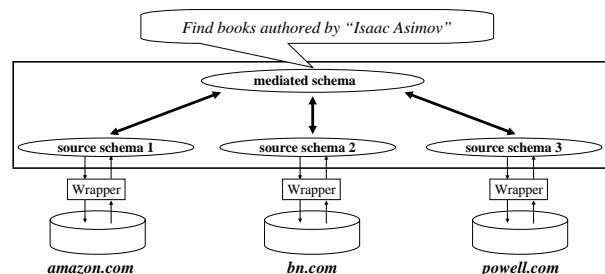


Figure 1: A data integration system in the book domain.

efficient construction and maintenance of data integration systems.

In this paper we describe the MOBS (Mass Collaboration to Build Systems) approach to efficiently building data integration systems. The basic idea underlying our approach is to ask the *users* of a system to “pay” for using it by answering relatively simple questions. Those answers are then used to further build the system and expand its capabilities. The following example illustrates our approach:

Example 1: Consider the data integration system in Figure 1. Today, to build such a system we must create the source schemas and the mediated schema and then specify the semantic mappings between these schemas. This is the bare minimum necessary to build a functioning system (with the help of a query processing engine such as the one described in [8]). Notice that any of the above three tasks is well-known to be difficult and time consuming. For example, even with the help of schema matching tools [12] it is still very labor-intensive to manually verify and correct *all* the semantic mappings that the tools suggest.

In the MOBS approach we also start by building the source schemas and the mediated schema. Next we treat the semantic mappings for the mediated-schema attributes as *system parameters*. We assign initial values to these parameters using random assignment or a schema matching tool. We then deploy this system “shell” on the Web and ask users to use it and provide feedback. We use the user feedback to readjust the system parameter values until these values converge.□

In the above example we have focused on treating semantic mappings as system parameters. However, we believe the approach can also be extended to “learn” other system features, such as the wrappers and the source schemas. Note

also from the above example that the mass collaboration approach would not replace, but rather *complement* well existing techniques to automate specific tasks in building data integration systems (e.g., schema matching and wrapper construction). In fact we believe it would *amplify* the effects of the current techniques.

Finally, the approach would be applicable to building systems in a broad variety of settings, including enterprise intranets, scientific domains (e.g., bioinformatics), and the Web. For example, within an organization, the employees can collaboratively build and expand a variety of systems that integrate organizational data. Bioinformaticists can collaboratively build a data integration system over the hundreds of online bioinformatics sources. Several volunteers in a particular Web domain (e.g., forest preservation) can deploy a system that integrate Web sources in that domain, by constructing an initial system shell, putting it on the Web, and asking the community to collectively maintain it. This way, the system can be constructed, maintained, and expanded at virtually no cost to any particular entity, but at great benefits for the entire community.

As described, the mass collaboration approach has the potential to dramatically reduce the cost of building data integration systems and spread their deployment in many domains. But the approach also raises numerous challenges. In the rest of the paper we address these challenges. Specifically, we make the following contributions:

- We propose mass collaboration, a conceptually novel approach to the problem of efficiently building data integration systems.
- We describe a solution that applies this approach to schema matching, a critical task that arises while constructing a data integration system.
- We present synthetic and real-world experiments that show the promise of our approach.
- We discuss future work, the challenges, and the potential applications of the approach beyond the context of data integration.

2. THE MOBS APPROACH

Our long-term goal is to use mass collaboration to automate as much as possible the *process* of building a data integration system. As a first step in this direction we study how to automate a major component of that process: finding semantic mappings between the source schemas and the mediated schema.

In this section we describe the MOBS solution to the above schema matching problem. As our running example we will use the simplified BookSeller data integration system in Figure 2. The system has a mediated schema with five attributes: *title*, *author*, *price*, *category*, and *year*, and four sources with schemas $S_1 - S_4$ (Figure 2).

Our task is to find the semantic mappings between the mediated schema and the source schemas $S_1 - S_4$. The MOBS solution to this problem consists of four major activities: *initialization*, *soliciting user feedback*, *computing user weights*, and *combining user feedback*. We now describe these activities in detail.

2.1 Initialization

We begin by building a *correct but partial* data integration system. We manually specify the correct mappings from *title* to each of the source schemas. Suppose these mappings are $title = a_1, title = b_1, title = c_1$, and $title = d_1$. The mappings allow us to build a system whose query interface consists of a *single* attribute: *title*. Users can immediately query this system to find books based on their titles.

We note that building an initial correct and partial system is crucial because even at the beginning we must have a functioning system. The system will have limited capabilities but it can already answer user queries *correctly*. If we simply initialize *all* mappings randomly we will probably create an initial system which produces *incorrect* query results. Users are unlikely to start using such a system.

Once we have the correct partial system we want to leverage user feedback to build the rest of the system: that is, to find the correct mappings for the remaining four mediated-schema attributes. To do this, we create *system parameters*, each of which maps a remaining mediated-schema attribute to a source-schema attribute. We consider *all* pairing of mediated-schema attributes with source-schema ones, and thus have the following parameters: $author = a_1, author = a_2, \dots, author = a_5, author = b_1, \dots, year = d_6$.

The correct value of a parameter such as $author = a_1$ is “yes” if *author* and a_1 are semantically equivalent (e.g., if a_1 is *writer*), and “no” otherwise.

Finally, we randomly set the initial values of the parameters. Section 4 discusses other initialization methods.

2.2 Soliciting User Feedback

We now deploy the above correct, but partial, system on the Web and ask users to begin using it and providing feedback. Our goal is to use the feedback to make the parameters “converge” to their correct values. When this happens we will have found the correct semantic mappings for the rest of the system.

Currently we solicit user feedback as follows. When the user submits a query to the system (e.g., “find all books whose titles contain ‘data integration’”), we make the user “jump through a hoop”. That is, the user must answer a question on the correct value of a parameter. Only after the user has answered this question do we display the results of his or her query.

Figure 3 shows a sample question that asks the user whether the attribute named *price* of a source (say, attribute d_3 of source S_4) matches attribute *year* of the mediated schema. This question amounts to soliciting the correct value of parameter $year = d_3$.

The user will answer “yes” or “no” (we are currently adding a third option, “not sure”), after examining the *name* of the attribute, several of its *data instances*, and the *context information* (i.e., the surrounding attributes in this case).

The frequency of “hoop jumping” can be adjusted, anywhere from one “hoop” per query to one “hoop” per several queries. In a related paper on this topic [5] we discuss other types of questions to ask the user as well as ways to entice users to answer questions.

2.3 Computing User Weights

In order to detect malicious or ignorant users we compute a *weight* for each user that measures the quality of his or her feedback. This weight is in the range $[0,1]$, with higher

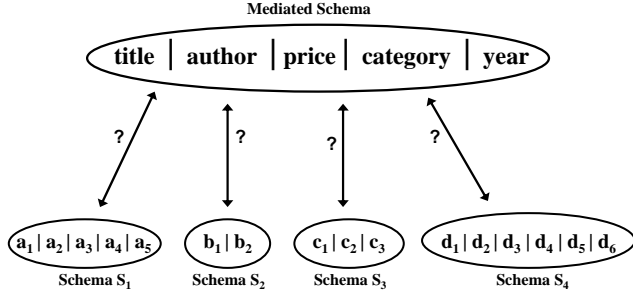


Figure 2: Using mass collaboration to find the semantic mappings between the attributes of the mediated schema and the source schemas.



Figure 3: A sample question that the system asks the user to answer.

weight indicating higher quality of feedback.

To compute such weights we set aside several sources for *user evaluation*. In the BookSeller example, suppose we set aside the first source, S_1 . We create all parameters that are related to this source: $title = a_1, title = a_2, \dots, title = a_5, author = a_1, \dots, year = a_5$. We then *manually* assign correct values to these parameters. Next, we solicit user answers on the correct values of these parameters as discussed in the previous subsection. Since we know the correct values of these parameters we can evaluate user answers and compute a user weight.

Consider a user u_1 . If the number of answers that u_1 has provided on the parameters coming from the evaluation sources is below a threshold k , then we say that user u_1 has not provided sufficient answers in order to be evaluated, and set $weight(u_1) = 0$. Otherwise we set $weight(u_1)$ to be the fraction of u_1 's answers that are correct.

To track users we require them to login to use the system. Note that a user needs to login only once; subsequent sessions can be handled automatically by cookies. For any single user we randomly mix the evaluation questions with teaching questions (i.e., questions used to get feedback on unlabeled sources) to ensure that the user does not know which ones are the evaluation questions.

Once a user weight has been computed we stop evaluating the user (we are exploring the option of continual evaluation, see further discussion in Section 3.1). We call a user *trustworthy* if his/her weight is above a threshold w (currently set at 0.65) and *untrustworthy* otherwise. If a user is untrustworthy we do not solicit additional feedback from him/her. In [5] we discuss methods to discourage users from intentionally providing incorrect feedback in an effort to be deemed untrustworthy and avoid answering questions.

2.4 Combining User Feedback

Before describing how to combine user feedback we discuss how we distribute user feedback. Recall that we have set aside source S_1 for evaluation and have manually identified the semantic mappings for this source. We have also identified semantic mappings from the first mediated-schema attribute *title* to all sources. Our job therefore is to solicit user feedback to find semantic mappings from the remaining mediated-schema attributes *author*, *price*, *category*, and *year* to sources $S_2 - S_4$.

We do this sequentially by first finding semantic mappings for *author*, then for *price*, and so on. As soon as we have found the semantic mappings for an attribute, say *author*, we immediately add *author* to the query interface and allow the user to use it to formulate his/her queries. This way user feedback can be shown to have an effect as soon as possible on the query interface, thus enticing users to provide feedback and making the system useful as early as possible.

To find mappings for a mediated-schema attribute, say *author*, we proceed sequentially by finding mappings from *author* to source S_2 , then S_3 , and then S_4 .

Consider source S_2 , which has only two attributes b_1 and b_2 (Figure 2). Finding mappings from *author* to S_2 reduces to obtaining the correct values for parameters $author = b_1$ and $author = b_2$. To do this, we solicit user answers in a round-robin fashion, using “hoop jumping” as described in Section 2.2. The first answer goes to $author = b_1$, the second goes to $author = b_2$, the third goes back to $author = b_1$, and so on.

Thus a parameter such as $author = b_1$ will receive a steady stream of “yes” and “no” user answers. Note that these answers are from trustworthy users. We monitor this stream of answers; as soon as a *convergence criterion* is satisfied we assign to parameter $author = b_1$ a value based on the answers it has received so far, and stop soliciting further answers for this parameter.

Suppose parameter $author = b_1$ has received a total of n predictions. The current convergence criterion is to stop if (a) the number of majority answers (either “yes” or “no”) reaches $n * 0.65$ and n exceeds 20, or (b) n exceeds 50. In either case we return the majority answer as the value for $author = b_1$. We say parameter $author = b_1$ has *converged* to that value.

We proceed similarly with parameter $author = b_2$. Once all parameters have converged we say the system has *converged* and stop soliciting user feedback.

3. EMPIRICAL EVALUATION

We now describe preliminary experiments that used both synthetic and real user populations to evaluate the MOBS approach.

3.1 Synthetic Experiments

Experimental Setting: We generated a variety of synthetic user populations. A population of 5000 users taken uniformly over $[0,1]$ means that we generated 5000 users and randomly assigned a reliability value from $[0,1]$ to each user. A reliability value of 0.6 means that on average the user answers 6 questions correctly out of 10.

Figure 4.a shows the results over 15 populations (see the figure legend). The populations belong to four classes. *Uniform $[0,1]$* was described above. *Uniform $\{0.6, 0.4\}$* means that half the users have reliability value 0.6 and the other

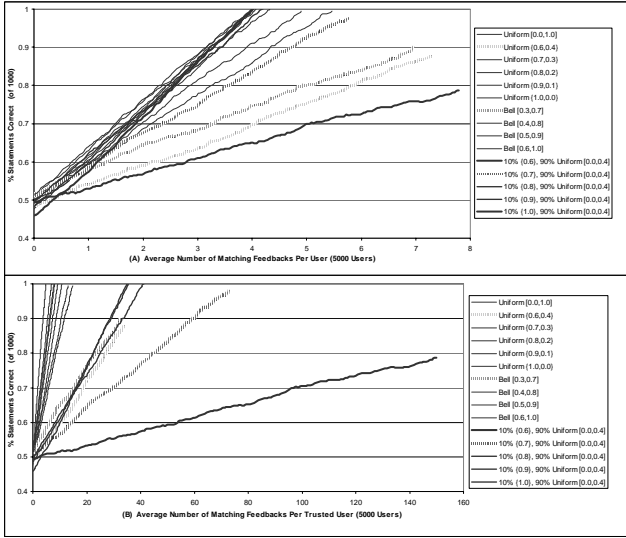


Figure 4: Matching accuracy as a function of average number of user answers, over a broad range of populations.

half 0.4. *Bell [0.3, 0.7]* means that the reliability values were generated according to a bell distribution over [0.3, 0.7]. Finally, *10% {0.6}, 90% uniform [0.0, 0.4]* means 10% of the users have reliability value 0.6 and the rest are assigned values uniformly over [0.0, 0.4]. The 15 populations thus represent a broad range of user populations that we expect to commonly occur in practice.

For each population we simulated its interaction with a data integration system that has 10 mediated-schema attributes and 12 sources, with 10 attributes per source. The simulation was carried out using the MOBS mechanism described in Section 2. We set aside 2 sources for evaluation. The system evaluated each user on 10 evaluation questions and deemed that user trustworthy only if he/she answered at least 7 questions correctly.

We solicited user feedback to match the schemas of the remaining 10 sources. Thus we had a total of $10 \times 10 \times 10 = 1000$ parameters whose values were to be set by the users. For each parameter we used the stopping criterion described in Section 2. At any timepoint each parameter has an assigned value (either the initial random assignment or the value obtained at convergence time). The *matching accuracy* at that timepoint is then computed as the number of parameters with correct value divided by the total number of parameters.

Matching Accuracy: First we want to know how matching accuracy changes over time and how high it can reach. Figure 4.a plots the *matching accuracy* as the average number of user answers increases, over all 15 populations. Each population is represented with a single line in the figure and the line ends when the system converges.

We use the term *evaluation answers* to refer to those answers that are used to evaluate the reliability of a user, and *teaching answers* to refer to those that are used to set the values of the system parameters. Figure 4.a considers on teaching answers, since the number of evaluation answers is the same for each user (10 in this case).

At the beginning the parameters were initialized randomly, hence matching accuracy around 50% (Figure 4.a). As users provide answers, the accuracy increases linearly and quickly reaches 100% for the first 9 populations in Figure 4.a, with only about 4 teaching answers per user.

Thus, for these populations each user has to provide only 14 answers on average (4 teaching and 10 evaluation) in order to correctly match schemas for the entire system. Note that if a system administrator were to do the schema matching he or she must examine *all* 1000 parameters. Thus, this result supports our argument that we can shift the significant labor (1000 answers) from the administrator to the mass of users, such that each user on average has to do only a small amount of work (14 answers in this case).

For the remaining 6 populations, Figure 4.a shows the matching accuracy in the range 79-99% when all parameters have converged. We note that these populations are quite “unreliable”, in the sense that they contain few “good” users and that the highest reliability values of these users are in the range [0.6, 0.7]. Since we used only 10 answers for evaluation and used a 0.65 threshold to filter out bad users, we admitted few good users and also admitted some bad users. Thus, for any parameter, when we cut off feedback to it at 50 answers and took the majority vote, there was a significant probability that we ended up with many answers from the bad users, and thus with the wrong parameter value. This explains the less than 100% accuracy at convergence.

Reaching Matching Accuracy of 100%: The above result suggests that if we increase the number of evaluation answers and tighten the convergence criterion (say, to consider cutoff threshold at 100 answers instead of 50), we should be able to increase the matching accuracy. This was indeed the case as we further experimented with these populations. We were able to prove (and confirm experimentally) that if we continuously (a) solicit both evaluation and teaching answers and (b) update user weights and parameter values accordingly, the matching accuracy will converge to 100% provided there are some trustworthy users in the population. The convergence speed obviously depends on the population characteristics. Figure 4.a demonstrates that this speed was quite fast (around 4 teaching answers per user) for a large variety of populations.

Number of Answers per Trusted User: Figure 4.a shows that only a few teaching answers per user (4-8) are required until the system converges to high matching accuracy. However, if the system has decided that a user is not trustworthy it will not solicit answers from that user again. Thus the burden of providing answers will fall mostly to trustworthy users. Hence we want to know how many answers on average such a user must shoulder. Figure 4.b shows that this number of answers remains quite low: around 10 for many populations and under 41 for all populations, except the two outlier populations at 72 and 150. These two populations are the “unreliable” populations discussed above. This result suggests that the burden of feedback for both trustworthy and untrustworthy users remains relatively low.

Effects of Population Size: We have also experimented with population size varying from 50 to 100,000 users, over numerous population topologies. We observed that matching accuracy remains stable across varying sizes. In all cases the time it took to manage the mass collaboration mecha-

nism was negligible. We further observed that, as expected, the number of feedback required per user to reach convergence decreases linearly as the population size increases. This suggests that our approach can scale up to very large populations in all important performance aspects.

3.2 Experiments with Real Users

We have also conducted preliminary experiments with real users and real-world data to evaluate our approach. We built a small data integration system over four real-world book sources. The system is similar to the one shown in Figure 2, with five mediated-schema attributes, four sources, and five attributes per source schema.

We conducted two experiments. In the first one we set aside two sources for testing and built an initial partial system whose query interface had only two attributes: *title* and *author*. We then asked a set of volunteers to use the system and provide feedback. 11 people volunteered, about half of whom had not previously heard about the system.

We simply asked the volunteers to use the system and answer questions to the best of their knowledge. The system deemed seven users as trustworthy. The other four either did not finish evaluation, or provided noisy answers. The system used the trustworthy users’ feedback to find semantic mappings for the 3 mediated-schema attributes *price*, *category*, and *year*. It quickly converged and reached 100% matching accuracy. The system was able to expand its query interface to include all five attributes of the mediated schema, thus showing that it can leverage user feedback to expand its capabilities. The average numbers of answers per user and per trustworthy user are 27.45 and 42.71, respectively.

In the second experiment we set aside one source for evaluation and asked for user feedback to match the remaining three sources. We also changed the evaluation and convergence criteria. We asked 5 evaluation questions and admitted users who answered at least 3 correctly. For each parameter we asked for 5 teaching answers and took the majority vote. 8 users volunteered and the system trusted 5. Again the system quickly converged, with the correct values for all parameters except one, thus reaching an accuracy of $26/27 = 96\%$. The average numbers of answers per user and per trustworthy user are 22.25 and 35.2, respectively.

The real-user experiments therefore provide preliminary evidence that real users can handle the cognitive load of the questions and quickly answer most of them correctly. We are currently designing experiments with much larger data integration settings and many more users. We are also looking for experimental domains that are less well known in order to more thoroughly evaluate the ability of real users to handle relevant questions.

4. DISCUSSION & FUTURE WORK

We have described a basic mass collaboration framework for building data integration systems. We now discuss possible extensions to this framework, as well as additional issues that arise in employing mass collaboration.

Schema Matching: We consider extending our current work on schema matching in several ways. It is clear that the system parameters can be initialized using semi-automatic schema matching tools [12] and that these initial values can be combined with user feedback to achieve faster convergence.

Further, we have focused only on finding one-to-one mappings, such as “location maps to address”. We are currently extending our framework to handle more complex mappings, such as “location maps to the concatenation of city and state”.

Finally, in the current framework the values of parameters such as *author* = b_1 and *author* = b_2 are obtained *independently* of each other, from the user answers. In many settings, we may know that attribute *author* maps into at most *one* source attribute. Thus, if we already establish that the value of *author* = b_1 is “yes”, then we can immediately conclude that *author* = b_2 is “no”, without obtaining additional user answers. We are extending our framework to exploit such parameter correlations, in order to minimize the amount of user feedback.

Other Labor-Intensive Tasks: We believe that the current MOBS approach can be extended to handle other data integration issues besides schema matching. Any problem that can be recast as a sequence of “yes”/“no” questions can potentially benefit from this approach. The key will be to break down the problem in such a way that users can handle the cognitive load of answering each individual question. We are currently exploring applying the MOBS approach to the wrapper construction problem.

Mass Collaboration Methodologies: Mass collaboration techniques have been applied to a variety of problems, such as constructing knowledge bases, tech support websites, and word sense disambiguation (see the related work section). However, no systematic study of mass collaboration issues has been conducted. We are currently conducting such a study, which examines the key challenges of mass collaboration in the context of data integration and proposes solutions. Examples of key challenges include how to attract users, how to entice them to give feedback, what types of feedback to solicit, and how to combine their feedback. We provide preliminary discussion of these issues in [5].

Beyond Data Integration: The mass collaboration techniques that we are developing have potential applications beyond just the data integration context. In a sense, the techniques provide a “hammer” that we can use to handle a variety of “nails”.

One such “nail” that we are pursuing is marking up data on the Semantic Web. The Semantic Web is advocated as the next-generation World-Wide Web where data is marked up and software programs can exploit the marked-up data to better satisfy information needs of users. Virtually all works on marking up data on the Semantic Web have asked the *owner* of a Web page to mark up the page’s data. This approach however often leads to a catch-22 situation: owners are not willing to spend a significant amount of efforts to mark up their pages because they do not see applications that show them the benefits of marking up; on the other hand, applications are not developed because there is no marked-up data.

To break this catch 22, we are exploring a conceptually opposite solution to marking up data: instead of asking the page owner (i.e., the *producer*), we ask the people who visit the page (i.e., the *consumers*) to help mark up the data. Our solution to this problem builds upon the mass collaboration techniques that we are currently developing in the data integration context.

5. RELATED WORK

Our work draws from several related areas, which we discuss below.

Knowledge Base Construction: Our work was inspired by several recent works that attempt to leverage the large volume of Web users to build knowledge bases, tech support websites, and word sense disambiguation ([14, 13], *quiq.com*, *openmind.org*). The basic idea of these works is to have users contribute facts and rules in some specified language. Our work differs from these in several important aspects. First, in building a knowledge base, potentially *any* fact or rule being contributed constitutes a parameter whose validity must be checked. Thus, the number of parameters can be very high (potentially in the millions) and checking them poses a serious problem. In contrast, the number of (system) parameters in our case is comparatively much smaller and thus potentially much more manageable. Second, such knowledge bases must provide some mechanisms to allow users to immediately leverage the contributed information (to gain some instant gratification effect). Providing such mechanisms in the context of knowledge bases can be quite difficult, because it requires performing inference over a large number of possibly inconsistent or varying-quality facts. Such mechanisms are considerably much simpler in our case, because feedback on the system parameters can immediately affect the query results.

Building Data Integration Systems: The manual construction and maintenance of data integration systems is very labor intensive and error prone. There have been many works on reducing the labor costs of *specific* tasks during the construction process, such as schema matching [12] and wrapper construction (e.g., [10, 1]), but few works on a systematic effort to address cost reduction for the *whole process*, with the exception of [15]. Our work on mass collaboration can be seen as providing a systematic solution to this problem.

Schema Matching: Numerous works have been conducted on schema matching, a fundamental problem in integrating data from heterogeneous sources (see [12] for a survey of recent works). These works employ manually crafted rules and machine learning techniques, with some limited human interaction, to discover semantic mappings. In contrast, our current work leverages the feedback of a multitude of users to find the mappings. To our knowledge, this is the first work on schema matching in this direction.

Autonomic Systems: Our work here is also related to autonomic systems in that data integration systems in the mass collaboration scheme can also exhibit autonomic properties such self-healing and self-improving. The key difference is that autonomic systems have traditionally been thought of as achieving these properties by observing the external environment and adjusting themselves appropriately. In contrast, our systems are observed by the external environments (i.e., the multitude of users) and then are adjusted by them accordingly.

6. CONCLUSION

The current cost of ownership of data integration systems is extremely high due to the need to manually build and maintain such systems. In this paper we have proposed a

mass collaboration approach to efficiently build data integration systems. The basic idea is to shift this enormous cost from the producers of the system to the consumers, but spreading it “thinly” over a large number of consumers. We have discussed the challenges of this approach and outlined preliminary solutions. We have also described the current status of our research in this direction and discussed the relationship between this work and several other areas. This research is conducted within the context of the AIDA (Automatically Integrating Data) project at the University of Illinois, whose goal is to build autonomic data integration systems.

7. REFERENCES

- [1] N. Ashish and C. Knoblock. Wrapper generation for semi-structured internet sources. *SIGMOD Record*, 26(4):8–15, 1997.
- [2] R. Avnur and J. Hellerstein. Continuous query optimization. In *Proc. of SIGMOD '00*, 2000.
- [3] J. Chen, D. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A scalable continuous query system for internet databases. In *Proc. of SIGMOD '00*, 2000.
- [4] A. Doan, P. Domingos, and A. Halevy. Reconciling Schemas of Disparate Data Sources: A Machine Learning Approach. In *Proc. of SIGMOD '01*, 2001.
- [5] A. Doan, R. McCann Building Data Integration System: A Mass Collaboration Approach. In *Proc. of the IJCAI-03 Workshop on Information Integration on the Web*, 2003.
- [6] H. Garcia-Molina, Y. Papakonstantinou, D. Quass, A. Rajaraman, Y. Sagiv, J. Ullman, and J. Widom. The TSIMMIS project: Integration of heterogeneous information sources. *Journal of Intelligent Inf. Systems*, 8(2), 1997.
- [7] L. M. Haas, D. Kossmann, E. L. Wimmers, and J. Yang. Optimizing queries across diverse data sources. In *Proc. of VLDB '97*, 1997.
- [8] Z. Ives, D. Florescu, M. Friedman, A. Levy, and D. Weld. An adaptive query execution system for data integration. In *Proc. of SIGMOD '99*, 1999.
- [9] C. Knoblock, S. Minton, J. Ambite, N. Ashish, P. Modi, I. Muslea, A. Philpot, S. Tejada. Modeling Web sources for information integration. in *Proc. of the Nat. Conf. on AI (AAAI) '98*, 1998.
- [10] N. Kushmerick, D. Weld, and R. Doorenbos. Wrapper Induction for Information Extraction. In *Proc. of the Int. Joint Conf. on AI (IJCAI) '97*, 1997.
- [11] A. Y. Levy, A. Rajaraman, and J. Ordille. Querying heterogeneous information sources using source descriptions. In *Proc. of VLDB '96*, 1996.
- [12] E. Rahm and P. Bernstein. On matching schemas automatically. *VLDB Journal*, 10(4), 2001.
- [13] M. Richardson, R. Aggrawal, and P. Domingos. Building the Semantic Web by mass collaboration. Technical Report UW-TR-03-02-05, Dept. of CSE, Univ. of Washington, 2003.
- [14] M. Richardson and P. Domingos. Building large knowledge bases by mass collaboration. Technical Report UW-TR-03-02-04, Dept. of CSE, Univ. of Washington, 2003.
- [15] A. Rosenthal, S. Renner, L. Seligman, and F. Manola. Data integration needs an industrial revolution. In *Proc. of the Workshop on Foundations of Data Integration*, 2001.

On the updatability of XML views over relational databases

Vanessa P. Braganholo
Universidade Federal do Rio
Grande do Sul - UFRGS
Instituto de Informática
vanessa@inf.ufrgs.br

Susan B. Davidson
University of Pennsylvania
Department of Computer and
Information Science
susan@cis.upenn.edu

Carlos A. Heuser
Universidade Federal do Rio
Grande do Sul - UFRGS
Instituto de Informática
heuser@inf.ufrgs.br

ABSTRACT

XML has become an important medium for data exchange, and is also used as an interface to – i.e. a view of – a relational database. While previous work has considered XML views for the purpose of querying relational databases (e.g. Silkroute), in this paper we consider the problem of updating a relational database through an XML view. Using the nested relational algebra as the formalism for an XML view of a relational database, we study the problem of when such views are updatable. Our results rely on the observation that in many XML views of relational databases, the nest operator occurs last and the unnest operator does not occur at all. Since in this case the nest operator is invertible, we can consider this important class of XML views as if they were flat relational views.

1. INTRODUCTION

XML is frequently used for publishing as well as exchanging relational data. Due to the highly unintuitive representation of data in the relational model, it is also increasingly being used as a mechanism through which to query and update legacy relational databases. For example, interfaces for gene expression data frequently represent the data to be annotated as an XML view of a relational database (e.g. AGAVE and GAME [1]).

One reason for this use of XML is that it naturally captures many-one relationships between data through the nesting of elements. In contrast, in the relational model nested data becomes fragmented over many relations, with many-one relationships captured in key and foreign key constraints. Thus one of the advantages of XML for conceptualizing information is its connection to nested relations.

As a simple example, consider the nested table of figure 2(a), which represents information about conferences and their location by year. This same information when represented in the relational model would be split over two tables (see figure 1). The nested table of figure 2(a) can also be understood as the XML instance in figure 3.

While other work has considered the problem of querying relational databases through XML views (e.g. Silkroute [12]), in this paper we focus on the problem of updating a relational database through an XML view. More precisely, we wish to be able to trans-

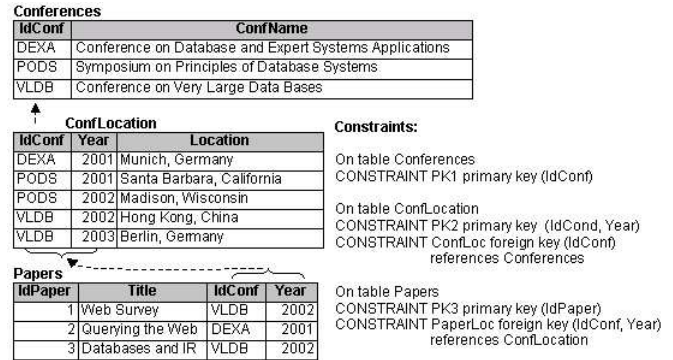


Figure 1: Sample database

late an update on an XML view to a set of updates on the underlying relations without introducing additional updates to the XML view.

To simplify the problem yet capitalize on the use of XML to capture nested relations, we consider XML views as defined by the nested relational algebra [23, 14]. The nested relational algebra contains the classical relational algebra operators (σ , π , \cup , \times , \bowtie , $-$) as well as the *nest* (ν) and *unnest* (μ) operators. There are several reasons for considering this algebra: First, there is a straightforward mapping between nested relations and XML views in this language [2, 8]. Second, it represents the core of languages such as XQuery when order and aggregate operators are ignored. Third, certain subclasses of expressions in this algebra have good properties regarding updatability. In this paper we will define such a subclass, drawing on classical relational view updatability results [9, 15]. Surprisingly, except for work on normal forms for nested relations [19, 20, 13] which focuses on removing ambiguity in nested relations, we could find no work related to updates through nested relations.

Using the results of this paper, we will show that the view of figure 2(a) is updatable for all insertions, deletions and modifications. That is, there is a unique, side effect free translation from any update on this view to the underlying relations of figure 1. The view is produced by the following query:

VIEW 1

$$\nu_{YearLocation} = (\pi_{(IdConf, ConfName, Year, Location)} (Conferences \bowtie ConfLocation))$$

This query is an example of a class which we call *well-nested project-select-join* queries. Views of this class are always updatable.

By relaxing restrictions on nesting and the form of the relational algebra expression, we can obtain a more general class of queries

IdConf	ConfName	YearLocation	
		Year	Location
DEXA	Conference on Database and Expert Systems Applications	2001	Munich, Germany
PODS	Symposium on Principles of Database Systems	2001	Santa Barbara, California
		2002	Madison, Wisconsin
VLDB	Conference on Very Large Data Bases	2002	Hong Kong, China
		2003	Berlin, Germany

(a)

IdConf	Details		
	Year	Location	Title
DEXA	2001	Munich, Germany	Querying the Web
VLDB	2002	Hong Kong, China	Databases and IR
	2002	Hong Kong, China	Web Survey

(b)

Figure 2: (a) View 1 (b) View 2

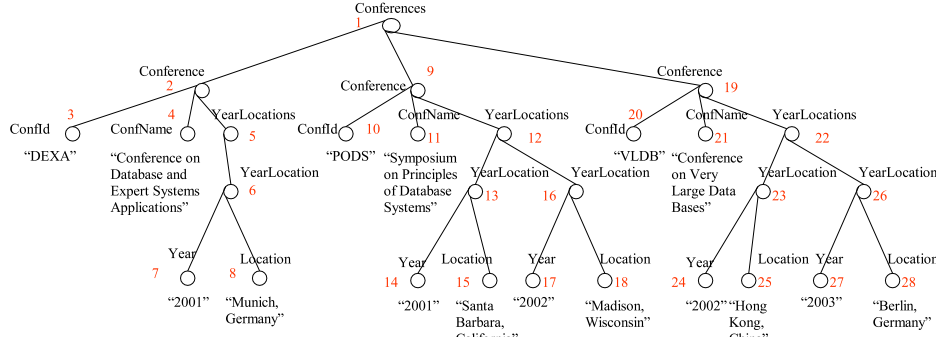


Figure 3: View 1 in XML

of form $\nu \dots \nu R$, where R is any relational algebra expression. We will call this class *nest-last relational queries*. An example of this class of views is as follows, and the corresponding nested relation is shown in figure 2(b).

VIEW 2

$$\begin{aligned} \nu_{Details} = & (Year, Location, Title) \left(\pi (IdConf, Year, Location, Title) \right. \\ & \left. \left(\sigma (ConfLocation.IdConf = Papers.IdConf \text{ AND } ConfLocation.Year = Papers.Year) \right. \right. \\ & \left. \left. (ConfLocation \times Papers) \right) \right) \end{aligned}$$

Although views defined by queries in this class are not in general updatable, we can determine whether or not a given update can be allowed using results from [9]. For example, in the above view we can always delete tuples but the same is not true for insertions or modifications. That is, we would not be able to insert the tuple $\langle \text{"NEW"}, 2003, \text{"LocName"}, \text{"Title"} \rangle$ on this view because we do not have the primary key of the Papers relation.

However, for XML views defined by general nested relational algebra (NRA) queries little can be said about updatability. Some general NRA queries can be rewritten to nest-last relational form using the rewrite techniques of [23]. However, there are others that cannot be rewritten and these remain an open problem. We will therefore focus in this paper on views produced by nest-last relational queries, and well-nested project-select-join queries.

The rest of this paper is organized as follows. In section 2 we define what it means for a view to be updatable, and summarize results from the relational case. Section 3 formalizes the notion of an update to an XML view, and discusses results on updates to views defined by nest-last relational queries as well as the special case of well-nested project-select-join queries. Section 4 concludes and presents future directions.

2. UPDATING RELATIONAL VIEWS

A large amount of work has been done on updates through relational views, and several different techniques have been proposed. We summarize them below:

1. View as an abstract data type: In this approach [21, 24], the DBA defines the view together with the updates it supports. The effect of updates on the base relations is explicitly defined.
2. Automatic translations for view updates: In [15], Keller defines five criteria that the translations should respect in order to be correct. Dayal and Bernstein [9] propose a translation mechanism that uses view graphs to decide if a given update translation is correct. The view graphs are constructed based on the syntax of the view definition and on the functional dependencies of base relations. A more recent work is presented in [18], but it does not consider views involving selections.
3. View complement: Bancilhon and Spyrtatos [4] introduce the notion of view complement to solve the update problem. In this approach, an update translation is considered correct if the complement of the view remains unchanged. Finding a view complement may be NP-complete even for very simple view definitions [7].
4. Views as conditional tables: A more recent technique consists in transforming a view update problem into a Constraint Satisfaction problem [22]. In this approach, views are represented as conditional tables. Each solution to the constraint satisfaction problem corresponds to a possible translation of the view update.
5. Object-based views: An extension of [16] to deal with object-based views is proposed in [5]. In this work, Barsalou propose algorithms for propagating updates in a hierarchical structure of objects. An implementation of object-based views is discussed in the Penguin Project [17].

In this paper, we follow the second approach and attempt to find an automatic translation for XML view updates. In particular, given an update against an XML view V , we wish to find a set of updates against the base relations defining V that does not cause additional side effects in the view. For example, if we were to modify the ConfName of the DEXA conference in the table of figure 2(a) to

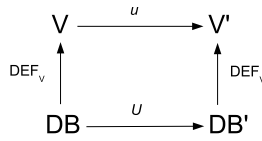


Figure 4: U exactly performs u . View V is defined over database DB using the definition function DEF_V

"NewConf", we would expect the ConfName of PODS or VLDB to remain the same. That is, a translation of this modification to the following SQL update would be incorrect:

```
UPDATE Conferences SET ConfName="NewConf"
```

However, in this case there is a side effect free translation of the view update as follows:

```
UPDATE Conferences SET ConfName="NewConf"
WHERE ConfId="DEXA"
```

DEFINITION 2.1 A translation U of an update u over a view V is exact when the diagram of figure 4 commutes [15]. ■

Having presented the meaning of *exact* translations, we can now define the notion of view updatability.

DEFINITION 2.2 A view is updatable with respect to a type of update operation (namely insertion, deletion or modification) if there is an exact translation t for every update u of this type that can be applied over the view, whenever u is syntactically correct and does not violate the semantic consistency of the underlying database. ■

This definition establishes two important points. First, view updatability depends on both the structure of the view and on the kind of update operation that is being applied on the view. As an example, a view can be updatable regarding insertions, but not modifications or deletions. Second, update operations must not cause side effects [15, 9].

3. UPDATING XML VIEWS

Our model of XML updates is very simple, and allows the insertion of a subtree at a given node, the deletion of the subtree rooted at a given node, or the modification of a node.

DEFINITION 3.1 An update operation u over an XML view V is a tuple $\langle u, \Delta, \text{ref} \rangle$, where u is the type of operation (insert, delete, modify); Δ is the XML tree to be inserted, or (in case of a modification) an atomic value; and ref is the address of a node in the XML tree. Deletions do not need to specify a Δ , since all the nodes under ref will be deleted. ■

The reference ref can be obtained by an addressing scheme such as DOM. In our examples, we use the node numbering shown in figure 3.

Since we are considering XML views of relational databases, not all XML updates will be valid since the schema of the XML view is fixed by the view definition. Therefore, updates must *respect* the schema and be *nesting compliant*.

EXAMPLE 3.1 Suppose we wish to insert a new conference location for the DEXA conference in the XML view of figure 3. In this case, we would have: $u = \text{insert}$, $\Delta = \Delta_1$ (figure 5), $\text{ref} = 5$.

Note that the insertion respects the schema of figure 2(a).

On the other hand, the following insertion does not respect the schema.

```

 $\Delta_1 = \{ \langle \text{YearLocation} \rangle$ 
   $\langle \text{Year} \rangle 2002 \langle / \text{Year} \rangle$ 
   $\langle \text{Location} \rangle \text{Aix en Provence, France} \langle / \text{Location} \rangle$ 
   $\langle / \text{YearLocation} \rangle \}$ 
 $\Delta_2 = \{ \langle \text{GeneralChair} \rangle \text{Adam Clark} \langle / \text{GeneralChair} \rangle \}$ 
 $\Delta_3 = \{ \langle \text{Conference} \rangle$ 
   $\langle \text{ConfId} \rangle \text{DEXA} \langle / \text{ConfId} \rangle$ 
   $\langle \text{ConfName} \rangle \text{Conference on Database and Expert}$ 
   $\text{Systems Applications} \langle / \text{ConfName} \rangle$ 
   $\langle \text{YearLocations} \rangle$ 
   $\langle \text{YearLocation} \rangle$ 
   $\langle \text{Year} \rangle 2003 \langle / \text{Year} \rangle$ 
   $\langle \text{Location} \rangle \text{Prague, Czech Republic} \langle / \text{Location} \rangle$ 
   $\langle / \text{YearLocation} \rangle$ 
   $\langle / \text{YearLocations} \rangle$ 
   $\langle / \text{Conference} \rangle \}$ 
 $\Delta_4 = \{ \langle \text{Conference} \rangle$ 
   $\langle \text{IdConf} \rangle \text{ER} \langle / \text{IdConf} \rangle$ 
   $\langle \text{ConfName} \rangle \text{Conference on Conceptual Modeling} \langle / \text{ConfName} \rangle$ 
   $\langle \text{YearLocations} \rangle$ 
   $\langle \text{YearLocation} \rangle$ 
   $\langle \text{Year} \rangle 2002 \langle / \text{Year} \rangle$ 
   $\langle \text{Location} \rangle \text{Tampere, Finland} \langle / \text{Location} \rangle$ 
   $\langle / \text{YearLocation} \rangle$ 
   $\langle \text{YearLocation} \rangle$ 
   $\langle \text{Year} \rangle 2003 \langle / \text{Year} \rangle$ 
   $\langle \text{Location} \rangle \text{Chicago, Illinois} \langle / \text{Location} \rangle$ 
   $\langle / \text{YearLocation} \rangle$ 
   $\langle / \text{YearLocations} \rangle$ 
   $\langle / \text{Conference} \rangle \}$ 

```

Figure 5: Δ s for the update operations

EXAMPLE 3.2 Suppose we wish to add the fact that Adam Clark was General Chair of DEXA in 2001.

$u = \text{insert}$, $\Delta = \Delta_2$ (figure 5), $\text{ref} = 6$.

As an example of a nesting violation, consider the following.

EXAMPLE 3.3 Suppose we insert information about DEXA 2003 at the root as follows:

$u = \text{insert}$, $\Delta = \Delta_3$ (figure 5), $\text{ref} = 1$.

This violates nesting since the resulting view has two tuples that represent DEXA. If this update were translated to a relational update and the view reconstructed, the resulting view would be different since it would have only one tuple representing the DEXA conference.

Deletions and modification are somewhat simpler, and are allowed as long as they do not violate the semantics (e.g. key, foreign key and non-null constraints) of the underlying database or the schema of the nested relational view.

EXAMPLE 3.4 Delete the subtree that has information about the location of VLDB 2002.

$u = \text{delete}$, $\Delta = \{ \}$, $\text{ref} = 23$.

EXAMPLE 3.5 Modify the name of the conference VLDB.

$u = \text{modify}$, $\Delta = \{ \text{"New VLDB name"} \}$, $\text{ref} = 21$.

In this example, ref points to a text node. Modifications are allowed only on leaves of the XML tree (text nodes).

3.1 Nest-last XML views

We now consider a class of XML views for which exact updates can be automatically translated. A *nest-last view* is a view defined by a nested relational algebra expression of form $\nu \dots \nu R$, where R is any relational algebra expression. We claim that this class of views can be treated by considering only the expression R , and that the nesting introduces *sets* of tuples to be inserted, deleted or modified in the underlying relational instance. Examples of this translation will be given in section 3.2.

CLAIM 3.1 Let $\nu \dots \nu R$ be a nest-last view and u an update over this view. Let $t(u)$ be the translation of u into an update over R . If R is updatable wrt $t(u)$, then $\nu \dots \nu R$ is updatable wrt to u .

PROOF: The proof is based on the fact that the nest (ν) operator is invertible [14, 23]. That is, after a nest operation, it is always possible to obtain the original relation by applying an unnest (μ) operation. Since in this type of view the nest operation is always the last operation to be applied, we can apply a reverse sequence of unnest operators to obtain the (flat) relational expression. ■

As an example, by unnesting on YearLocation in view 1, we would obtain a flat relational expression:

$$\pi(\text{IdConf}, \text{ConfName}, \text{Year}, \text{Location})(\text{Conferences} \bowtie \text{ConfLocation})$$

Claim 3.1 reduces the problem of investigating updatability of XML views to the problem of updates through relational views. Consequently, it is possible to use all the work in relational views for XML views of this class.

3.2 Translating XML updates into relational view updates

For nest-last views, we can translate XML updates into updates to the corresponding relational (flat) view. This section briefly introduces our technique based on examples.

Insertions. We unnest the subtree specified in Δ and create one relational tuple for each corresponding unnested tuple to be inserted into the relational view. If there is any missing information, we fill it in with information collected from the leaves under the elements along the path from *ref* to the root of the XML tree. In the case of example 3.1, the insertion would be translated to an insertion in the relational component of view 1 (V1) as:

```
INSERT INTO V1 (IdConf,ConfName,Year,Location)
VALUES ("DEXA", "Conference on Database and Expert
Systems Applications",2002,"Aix en Provence,France")
```

As another example, suppose we insert a new conference with no information about YearLocations. This would be translated as:

```
INSERT INTO V1 (IdConf,ConfName,Year,Location)
VALUES ("NEW", "New Conference", NULL, NULL)
```

Insertions may also be translated to a *set* of insertions in the relational view. As an example, consider the insertion of the subtree $\Delta = \Delta_4$ (figure 5), *ref* = 1.

This would be translated to

```
INSERT INTO V1 (IdConf,ConfName,Year,Location)
VALUES ("ER", "Conference on Conceptual Modeling",
2002, "Tampere, Finland")
```

```
INSERT INTO V1 (IdConf,ConfName,Year,Location)
VALUES ("ER", "Conference on Conceptual Modeling",
2003, "Chicago, Illinois")
```

Deletions. Deletions are translated in a similar way. To build the DELETE SQL statement, we use the subtree of information rooted at *ref* as well as information collected along the path from *ref* to the way to the document root. Each value found in this path becomes a condition in the WHERE clause of the deletion.

In the case of example 3.4, we would translate it using the information of the node being deleted as well as its parent (in this case, the VLDB conference). The translation would be:

```
DELETE FROM V1 WHERE Year=2002 AND
Location= "Hong Kong, China" AND IdConf= "VLDB" AND
ConfName= "Conference on Very Large Data Bases"
```

A deletion can also affect more than one tuple in the relational view. An example would be the attempt to delete node *ref* = 9. This would be translated to:

```
DELETE FROM V1 WHERE IdConf= "PODS" AND
ConfName= "Symposium on Principles of Database Systems"
```

Modifications. Modifications are treated in the same way as deletions. That is, we use information about the node and its ancestors to build the WHERE clause. In the case of example 3.5, the translation is:

```
UPDATE V1
WHERE IdConf= "VLDB" AND
ConfName= "Conference on Very Large Data Bases"
SET ConfName= "New VLDB name"
```

We have shown how to translate updates over an XML view to updates over the corresponding relational view. The techniques of [15, 9] can then be used to translate these updates to the underlying relational database.

3.3 Nest-last Project-Select-Join Views

We now investigate a special subset of nest-last views that are well behaved with respect to updates.

DEFINITION 3.2 A nest-last project-select-join view (NPSJ) is a nest-last view with the following restrictions: the relational expression is a project-select-join; the keys of the base relations are not projected out; and joins are made only through foreign keys. ■

LEMMA 3.1 NPSJ views are always updatable for insertions.

PROOF SKETCH: Claim 3.1 shows how to reduce an XML view to a relational view. Based on this result, we are now able to use the technique of Dayal and Bernstein [9] to prove that there is always an exact translation for insertions and deletions for NPSJ views. Since the nest can be ignored, we start by defining a general PSJ view that is the join of relations R_1, R_2, \dots, R_m , where the keys of R_1, R_2, \dots, R_m are preserved in the view and joins are done over foreign keys. We then draw a view graph for this view, as illustrated in figure 6. Nodes in this graph represent attributes. The upper nodes represent attributes of the base relations, and the lower ones represent view attributes. Primary keys are represented as *Ps* and foreign keys as *Bs*. Edges represent functional dependencies between attributes in the relations, join conditions or the derivation of view attributes. The proofs for insertions and deletions are based on finding paths in this directed graph. We say there is a path from a set of attributes *X* to a set of attributes *Y* if each attribute in *Y* is reachable from some subset of *X*.

Insertions. For insertions, [10] divides the problem into smaller ones. They claim that insertions are always exactly translatable if we can express the view definition as a sequence of views definitions, each one defined over only two relations. In the case of NPSJ, this is obviously true. The additional conditions are: the two relations must be equijoinable over foreign keys (true by definition of NPSJ); and there must be a path from the attributes of one of these relations to all view attributes that originated from these two relations. The latter is also obviously true. By looking at the graph of figure 6, it is easy to see that the relation containing the foreign key has always such a path. As an example, consider the two relations inside the dotted box in figure 6. There is a path from the attributes of R_2 to all attributes in the view that originated from R_1 or R_2 . ■

For modifications and deletions, even in the relational case there may fail to be an exact translation for certain types of updates over a PSJ view. This type of update attempts to change (or delete) some but not all occurrences of data that is repeated in the view, and thus causes side effects. As an example, consider the unnested version of the view 1. This view has the values of IdConf and ConfName repeated in several tuples. An attempt to modify a conference name could be stated as

```
UPDATE V1
SET ConfName= "New Name" WHERE IdConf= "VLDB"
```

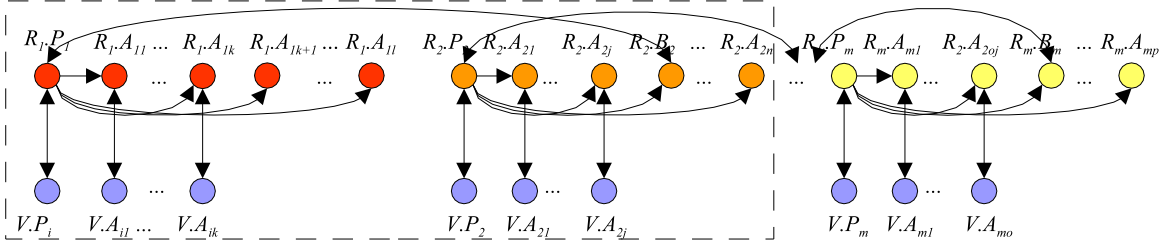



Figure 6: View graph

This is exact, since it modifies all occurrences of VLDB tuples. However, consider this same example with a slight modification.

```
UPDATE V1 SET ConfName= "New Name"
WHERE IdConf= "VLDB" AND Year=2002
```

As one can easily see, there is no way to translate this request without causing side effects, because a tuple that does not satisfy the qualification of this modification request would also be affected (more specifically, the tuple with IdConf="VLDB" and Year=2003). The same problem happens for deletions.

Fortunately, proper application of the nest operator can be used to avoid this type of ambiguity. For example, for the view shown in figure 3 this kind of bad modification (or deletion) request cannot happen. Recall the translation of the modification update example 3.5, which translates the modification to update all VLDB tuples.

However, if we had nested this view in a different way, the same update would fail to be exact. As an example, consider the same view, now nested by $\{IdConf, ConfName\}$ instead of $\{Year, Location\}$. The same IdConf and ConfName appear several times in the view, as in the relational case. Thus, not all modifications and deletions over this view would be exactly translatable.

The updatability of NPSJ views with respect to modifications and deletions depends on the way in which we traverse the foreign key constraints when nesting. In view 1, we traverse the foreign key constraint from 1 to n . That is, for each Conference tuple there are many ConfLocation tuples, so we nest ConfLocation tuples (the n 's) under their corresponding Conference tuple (the 1's). In the second example (where we nested over $\{IdConf, ConfName\}$), we nested the 1's under the n 's, causing the 1's to appear several times in the resulting view.

To define when a NPSJ view is well-nested, we reason about the foreign keys of the underlying relations. Recall that the syntax of a foreign key constraint C on table R_1 is given by C_{R_1} FOREIGN KEY (FK_1, \dots, FK_n) REFERENCES R_2 (K_1, \dots, K_n). When the attribute names (K_1, \dots, K_n) are the same as (FK_1, \dots, FK_n), they can be omitted, as in the example of figure 1.

DEFINITION 3.3 Let C_{R_1} be a foreign key constraint, and $V(R_1)$ be the set of attributes of R_1 that appear in the view. An ambiguity eliminating nest with respect to C_{R_1} is a nest of the form $\nu_{X=(D)}$, where $D = \{V(R_1)\} - \cup_i FK_i$. ■

The idea behind this definition is that by omitting the foreign keys of R_1 and the keys of R_2 in the nest, we collect their values together thus eliminating ambiguity. That is, each value appears just once in the view.

The view of example 1 has an ambiguity eliminating nest since $R_1 = ConfLocation$, $R_2 = Conferences$, $FK = \{IdConf\}$, $V(R_1) = \{IdConf, Year, Location\}$ and we are nesting over $\nu_{YearLocations = (Year, Location)}$.

DEFINITION 3.4 A NPSJ view that involves more than two base relations is well nested if

1. It has one ambiguity eliminating nest for each foreign key constraint that was used to join the base relations; and
2. The nests are executed in the opposite order of the joins. ■

An example of well-nested NPSJ is view 1. Another example is given by the following NRA expression:

$$\begin{aligned} \nu_{Papers=(IdPaper, Title)} (\nu_{YearLocations=(Year, Location)} \\ (\pi_{(IdConf, ConfName, Year, Location, IdPaper, Title)} \\ (Conference \bowtie (ConfLocation \bowtie Papers)))) \end{aligned}$$

This expression differs from previous examples because it contains two nested relations in the same nesting level. The resulting view has the following structure: $(IdConf, ConfName, \{YearLocations\}, \{Papers\})$, where $YearLocations$ and $Papers$ are nested relations. This example shows that NPSJ views are capable of expressing complex structures.

LEMMA 3.2 Well nested NPSJ views are always updatable with respect to modifications and deletions.

PROOF SKETCH: We divide the proof in two steps.

Modifications. In order to simplify the proof, we consider a view defined over two base relations, say $R_1 \bowtie R_2$. The graph of this view corresponds to the dotted box of figure 6. Using the technique of [9], there must be a path from the attributes of the relation whose attributes are being modified to all view attributes that were specified in the WHERE clause. In the case of well-nested NPSJ views, this is directly related to how we specify the update against the relational view. In order for R_1 and R_2 to be well-nested, R_2 must be nested under R_1 . If we want to modify an attribute from R_1 , the WHERE clause will have only attributes generated from R_1 . Obviously, there is a path from the attributes in R_1 to the view attributes generated from R_1 . If we want to modify attributes from R_2 , the WHERE clause will have attributes generated both from R_1 and R_2 . Since it is possible to use the arrow $R_1.P_1 - R_2.B_2$ to reach all the view attributes, the condition is satisfied. The proof can be easily generalized to views defined over more than two base relations.

Deletions. Deletions have a WHERE clause that specifies conditions that view tuples must satisfy in order to be deleted. The condition for exact translation for deletions says that there must be a path in the view graph from the relation chosen to translate this deletion to all attributes specified in the WHERE clause. Our proof supposes that all attributes of the view were specified in the WHERE clause, since this is the "worst case". It is easy to see that one can always choose the last relation joined to translate the deletion to the database because there is always a path from the attributes on this relation to all view attributes (see R_m in figure 6) due to the edges introduced by join conditions. ■

4. CONCLUSIONS

We have investigated the problem of how to translate updates on XML views over relational databases to updates on the underlying relations. In particular, we showed how updates to a nest-last view can be translated to updates on the corresponding relational view. Techniques from the relational model can then be used to determine if the nest-last XML view is updatable for a given update.

For the special class of NPSJ views, we showed that it is always possible to find exact translations for insertions. When these views are well-nested it is also possible to find exact translations for deletions and modifications. Thus, well-nested NPSJ views are updatable for all valid updates.

Well-nested NPSJ views are a very significant class of XML views. If we store an XML view of this class in a relational database exploiting the keys and semantic constraints of the document, we would be able to reconstruct the XML view using only joins over foreign keys [6]. That is, the relational instance represents a natural storage scheme for the XML view when constraints are taken into account.

Since our focus was on XML views of legacy relational databases rather than XML views of XML documents, it was reasonable to make some simplifications. First, the schema of the view was fixed which meant that limited forms of insertions and deletions were allowed. Second, it was sufficient to consider the nested relational algebra as the basis of view expressions rather than something like the XQuery algebra.

The XQuery algebra [11] expresses all the operators of NRA, as well as aggregation, quantification, sorting and iteration. It also has operators to deal with XML specific features - ordering, comments, processing instructions. It is clear that since aggregation loses information, views involving aggregation will not be updatable [16]. Furthermore, operators involving ordering are not relevant when the underlying representation is relational.

We claim that NRA is general enough to be able to represent the same type of structures as object-based views [5]. In particular, object-based views include only relations that are related by integrity constraints, and can therefore be expressed as nest-last views. The main difference between object-based views and our approach based on NRA is related to side effects. Object views can be formed by creating relationships (pointers) between simple objects and may therefore avoid repeating information. For example, a view can be defined as a set of objects representing papers, where each paper is connected to an object that represents the conference in which the paper was published. Information about conferences is not repeated, as it would be in the corresponding NRA view. Thus, changing the name of a conference would affect a single object, which is referenced by several papers, and would be side effect free. Note that by considering ID and IDREF in XML and using “normalized” XML views [3] we can achieve the same result.

In future work we plan to explore general XML views.

Acknowledgments. We would like to thank Capes for supporting this research (BEX 1123/02-5).

5. REFERENCES

- [1] XML for molecular biology as compiled by paul gordon. <http://www.visualgenomics.ca/gordonp/xml>.
- [2] S. Abiteboul and N. Bidoit. Non first normal form relations to represent hierarchically organized data. In *PODS*, pages 191–200, 1984.
- [3] M. Arenas and L. Libkin. A normal form for XML documents. In *Proceedings of PODS 2002*, Madison, Wisconsin, Jun 2002.
- [4] F. Bancilhon and N. Spyratos. Update semantics of relational views. *ACM TODS*, 6(4), Dec 1981.
- [5] T. Barsalou, N. Siambela, A. M. Keller, and G. Wiederhold. Updating relational databases through object-based views. In *SIGMOD*, pages 248–257, Denver, CO, 1991.
- [6] Y. Chen, S. B. Davidson, and Y. Zheng. 3XNF: Redundancy eliminating XML storage in relations. In *VLDB*, Berlin, Germany, 2003.
- [7] S. S. Cosmadakis and C. H. Papadimitriou. Updates of relational views. *Journal of the Association for Computing Machinery*, 31(4):742–760, Oct 1984.
- [8] A. S. da Silva, I. M. E. Filha, A. H. F. Laender, and D. W. Embley. Representing and querying semistructured web data using nested tables with structural variants. In *ER*, 2002.
- [9] U. Dayal and P. A. Bernstein. On the correct translation of update operations on relational views. *ACM TODS*, 7(3):381–416, Sep 1982.
- [10] U. Dayal and P. A. Bernstein. On the updatability of network views - extending relational view theory to the network model. *Information Systems*, 7(2):29–46, 1982.
- [11] P. Fankhauser, M. Fernández, A. Malhotra, M. Rys, J. Siméon, and P. Wadler. The xml query algebra. W3C Working Draft, Feb 2001. www.w3.org/TR/2001/WD-query-algebra-20010215.
- [12] M. Fernández, W.-C. Tan, and D. Suciu. Silkroute: Trading between relations and xml. In *Ninth International World Wide Web Conference*, 2000.
- [13] G. Hulin. On restructuring nested relations in partitioned normal form. In *VLDB*, pages 626–636, Brisbane, Australia, 1990.
- [14] G. Jaeschke and H.-J. Schek. Remarks on the algebra of non first normal form relations. In *PODS*, pages 124–138, Los Angeles, CA, March 1982.
- [15] A. M. Keller. Algorithms for translating view updates to database updates for views involving selections, projections, and joins. In *PODS*, pages 154–163, Portland, Oregon, 1985.
- [16] A. M. Keller. The role of semantics in translating view updates. *IEEE Computer*, 19(1):63–73, 1986.
- [17] A. M. Keller and G. Wiederhold. Penguin: Objects for programs, relations for persistence. In A. B. Chaudhri and R. Zicari, editors, *Succeeding with Object Databases*. John Wiley & Sons, 2001.
- [18] R. Langerak. View updates in relational databases with an independent scheme. *ACM TODS*, 15(1):40–66, 1990.
- [19] A. Makinouchi. A consideration on normal form of not-necessarily-normalized relation in the relational data model. In *VLDB*, pages 447–453, Tokyo, Japan, 1977.
- [20] W. Y. Mok, Y.-K. NG, and D. W. Embley. A normal form for precisely characterizing redundancy in nested relations. *ACM TODS*, 21(1):77–106, Mar 1996.
- [21] L. A. Rowe and K. A. Shoens. Data abstraction, views and updates in rigel. In *SIGMOD*, pages 71–81, Boston, Massachusetts, 1979.
- [22] H. Shu. Using constraint satisfaction for view update translation. In *Proc. of European Conference on Artificial Intelligence (ECAI)*, Brighton, UK, 1998.
- [23] S. J. Thomas and P. C. Fischer. Nested relational structures. *Advances in Computing Research*, 3:269–307, 1986.
- [24] L. Tucherman, A. L. Furtado, and M. A. Casanova. A pragmatic approach to structured database design. In *VLDB*, pages 219–231, Florence, Italy, Oct 1983.

Tree Automata to Verify XML Key Constraints

Béatrice Bouchou
Université de Tours
LI/Antenne Univ. de Blois
3 place Jean Jaurès
41000 Blois, France
bouchou@univ-tours.fr

Mírian Halfeld Ferrari
Université de Tours
LI/Antenne Univ. de Blois
3 place Jean Jaurès
41000 Blois, France
mirian@univ-tours.fr

Martin A. Musicante^{*}
Univ. Federal do Paraná
Dep. de Informática
C.P. 19081
81531-970 - Curitiba - Brazil
mam@inf.ufpr.br

ABSTRACT

We address the problem of checking key constraints in XML. Key constraints have been recently considered in the literature and some of their aspects are adopted in XMLSchema. However, only few works have appeared concerning the verification of such constraints.

Unranked deterministic bottom-up tree automata can be used to validate XML documents against a schema. These automata work over (unranked) trees used to represent XML documents.

In this paper we show how key constraints can be integrated in such automaton by extending the automaton to carry up values from the leaves to the root, during its run. In fact the tree automaton becomes a tree transducer. Under these conditions, the key verification is done in asymptotic linear time on the size of the document.

Keywords: XML key constraints, tree automata, XML

1. INTRODUCTION

We consider a data-exchange environment where an XML document should respect two kinds of constraints: schema constraints and key constraints. Schema constraints correspond to attribute and element restrictions. Key constraints give the possibility of identifying data without ambiguity and, therefore, introduce a value-based method of locating items in a document. Keys for XML have received more attention recently. They exist in XMLSchema [2] and some formal definitions have been introduced in [9, 10].

We address the problem of validating both schema and key constraints. To this end we use bottom-up tree transducers as an extension of the tree automata introduced in [7]. Tree automata are a natural way of describing structural constraints. However, in order to validate key constraints we need to manipulate data values. To this end, we introduce output functions whose basic task is to define the values to be carried up from children to parents in an XML tree.

In this work we concentrate our attention to the validation of key constraints as defined in [9]. We present an efficient key validator

^{*}On leave at Université de Tours. Partly supported by CAPES (Brazil) BEX1851/02-0.

whose work consists in executing a tree transducer over an XML document.

We see an XML document as a structure composed by an unranked labeled tree and functions *type* and *value*. The function *type* indicates the type of a node (*element*, *attribute* or *data*). The function *value* gives the value associated to a node. Figure 1 shows part of the labeled tree representing the document used in our examples. Each node is represented by a label and a position (for instance, position 0 is associated to the label *politicPos*). Moreover, in this figure, an XML element has both its sub-elements and attributes as children. Elements and attributes associated with an arbitrary text have a child labeled *data*. Attribute labels are depicted with a preceding @. The following example illustrates how the tree transducer performs the validation of a key constraint.

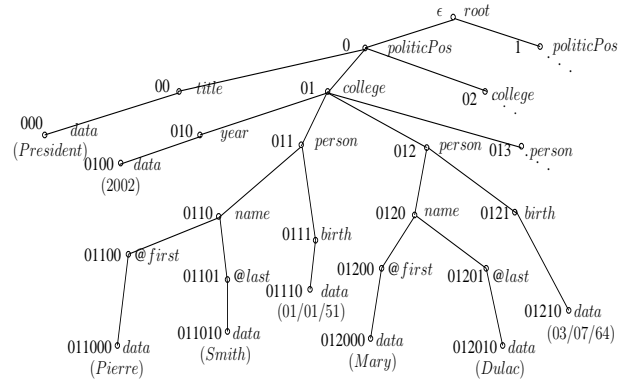


Figure 1: Tree representation of an XML document.

Example 1.1 We suppose the document of Figure 1, describing the organization of indirect elections, *i.e.*, the electoral colleges that vote for different political positions. Now consider the constraint expressed by $(/politicPos, (./college, \{./year\}))$ to indicate that for a political position, an electoral college can be uniquely identified by the year of the election. In other words, *year* is a key for an electoral college voting for a given political position. We say that *politicPos* defines context nodes, *college* defines target nodes and *year* defines key nodes.

In order to verify the above constraint, we execute our tree transducer over the tree of Figure 1. This execution consists of visiting the tree in a bottom-up manner, according to the following steps:

1. The tree transducer computes the values associated to all nodes labeled *data*.

We consider $value(0100) = 2002$, $value(0200) = 1998$ as some of the values computed in this step¹.

2. The tree transducer analyzes the parents of the *data* nodes. If they are key nodes, they receive the values computed in step 1. Otherwise, no value is carried up.

In our case, the values 2002 and 1998 are passed to the key nodes 010 and 020, respectively.

3. The tree transducer continues its execution just passing the values from children to parent until it finds a target node. At this level the values for each key are grouped in a list.

In our case, the node 01 (labeled *college*) is a target node. As the key is composed by just one item, the list contains only the value 2002.

4. This step consists of carrying up the lists of values obtained in the previous step until finding a context node. At this level, the transducer tests if all the lists are distinct, returning a boolean value.

In our case, *politicPos* is a context node. It receives several lists, each one containing the year of a college. The test verifies the uniqueness of those years.

5. The boolean values computed in step 4 are carried up to the root. At the root, the conjunction of these boolean values is obtained.

The key constraint $(/politicPos, (./college, \{./year\}))$ is satisfied if the conjunction computed at step 5 results in *true*. \square

Given the tree representation \mathcal{T} of an XML document, we can test schema and key constraints by a single bottom-up visit of the labeled tree. Example 1.1 illustrates how the key constraint verification is performed. In [7] schema constraint verification is treated in details (see Section 3 for a short explanation on this aspect).

The main contributions of our work are

- An efficient validator for both schema and key constraints. The validation is performed by the bottom-up visit of the XML tree, in only one pass.
- A method for allowing the use of DTDs with key constraints. DTDs are easily translated into a tree automata [7]. In this work we give an algorithm to add key constraint verification to the tree automata. In this way, our verification takes into account both *unique* and *not null* properties of the key.
- An unranked bottom-up tree transducer (a generalization of the ranked one [11]) where syntactic and semantic aspects are well separated. Schema validation deals with syntactic (structural) features of an XML tree (a plain tree automata can be used to this end). Key validation is about semantics. It requires an extension of tree automata allowing the manipulation of data values.

The rest of this paper is organized as follows. In Section 2 we recall the definitions of unranked labeled trees and key constraints. In Section 3, unranked tree transducers are introduced as an extension of tree automata. We present a method to express key constraints as output functions and we show that, in this way, an efficient verification of key constraints is possible. Finally, in Section 4, we consider some related work, and we discuss our perspectives for further research. Proofs are omitted due to lack of space.

¹Node 0200 does not appear in Figure 1, but from the definition of positions in Section 2, it is easy to see that it is a grand-child of node 02.

2. XML TREES AND KEY CONSTRAINTS

In this section we recall the notions of unranked Σ -valued trees and key constraints.

Firstly, let U be the set of all finite strings of positive integers with the empty string ϵ as the identity. In the following definition we assume that $dom(t) \subseteq U$ is a nonempty set closed under prefixes², i.e., if $u \preceq v$, $v \in dom(t)$ implies $u \in dom(t)$.

Definition 2.1 - Σ -valued tree t : A nonempty Σ -valued tree t is a mapping $t : dom(t) \rightarrow \Sigma$ where $dom(t)$ satisfies: $j \geq 0$, $uj \in dom(t)$, $0 \leq i \leq j \Rightarrow ui \in dom(t)$. The set $dom(t)$ is also called the set of *positions* of t . We write $t(v) = a$, for $v \in dom(t)$, to indicate that the Σ -symbol associated to v is a . For each position p in $dom(t)$, $children(p)$ denotes the positions pi in $dom(t)$, and $father(p)$ denotes the *father* of p . Define an *empty tree* t as the one having $dom(t) = \emptyset$. \square

Unranked trees can be used to represent an XML document. In fact, there are different ways to encode an XML document as a tree. The following definition introduces our choice of representation.

Definition 2.2 - XML tree \mathcal{T} : Let $\Sigma = \Sigma_{ele} \cup \Sigma_{att} \cup \{data\}$ be an alphabet where Σ_{ele} is the set of element names and Σ_{att} is the set of attribute names. An XML tree is a tuple $\mathcal{T} = (t, type, value)$ where:

- t is a Σ -valued tree (i.e., $t : dom(t) \rightarrow \Sigma$).
- $type$ and $value$ are functions defined as follows for a position $p \in dom(t)$:

$$type(p) = \begin{cases} data & \text{if } t(p) = data \\ element & \text{if } t(p) \in \Sigma_{ele} \\ attribute & \text{if } t(p) \in \Sigma_{att} \end{cases}$$

$$value(p) = \begin{cases} val \in \mathbf{V} & \text{if } type(p) = data \\ undefined & \text{otherwise} \end{cases}$$

where \mathbf{V} is an infinite (recursively enumerable) domain. \square

An XML key constraint over \mathcal{T} is defined in three steps. In the first step we identify a set of positions from the root as the context in which the key must hold. In the second step, we obtain a set of target positions on which the key is being defined. Finally, we specify the set of values that distinguish each target position. We use a subset of XPath expressions, as in [10], to specify context and target positions and to obtain the values that compose keys.

Using the syntax of [9] a key can be written as

$$(P, (P', \{P_1, \dots, P_k\}))$$

where P , P' and P_1, \dots, P_k are path expressions. P is called the *context path*, P' the *target path* and P_1, \dots, P_k the *key paths*. Figure 2 shows a Σ -valued tree with the positions we can reach by following each path. Level 3 corresponds to the root of the tree (reached by the path “/”). The context path begins at the root and specifies a set of *context positions*, shown in level 2. We say that these positions are associated to context labels. From each context position p , we define a set of *target positions* (associated to target labels) corresponding to the nodes reachable from p by following the path P' (level 1). The key constraint specified by P_1, \dots, P_k must hold for every target position. Level 0 corresponds to the positions composing a key (each of them associated to a key label).

Now, in [9], we find different types of keys, i.e., different definitions of the semantics of a key. In our work, we adopt the definition called *strong keys*. Moreover, we consider that key paths

²The *prefix relation* in U , denoted by \preceq is defined by: $u \preceq v$ iff $uw = v$ for some $w \in U$.

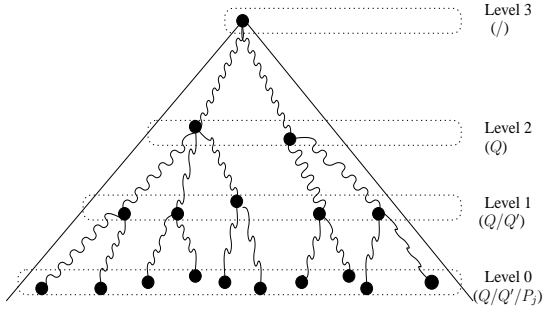


Figure 2: Context, target and key positions in an XML tree.

must define attributes or elements which occur *exactly once* and are associated to a *data* node. This restriction is also present in XMLSchema [10]. In the following we formalize the notion of key satisfaction and we present some examples of XML keys.

Definition 2.3 - Key satisfaction [9]: An XML tree \mathcal{T} is said to satisfy a key $(P, (P', \{P_1, \dots, P_k\}))$ iff for each context position p defined by P the following conditions hold:

- (i) For each target position p' reachable from p via P' there exist a unique position p_j from p' , for each $P_j (1 \leq j \leq k)$.
- (ii) For any target positions p', p'' , reachable from p via P' , whenever the values reached from p' and p'' via $P_j (1 \leq j \leq k)$ are equal, then p' and p'' must be the same position. \square

Example 2.1 Considering the document represented by Figure 1, we write the following keys:

- $K_1 = (/./politicPos, \{./title\})$

Within the context of the whole document (“/” denotes the empty path from the root), a political position is identified by its title.

- $K_2 = (/politicPos, \{./college, \{./year\}\})$

For a political position, an electoral college can be uniquely identified by the year of the election.

- $K_3 = (/politicPos/college, \{./person, \{./name/@first, ./name/@last, ./birth\}\})$

Within an electoral college, a person can be uniquely identified by the composition of his/her first, last name and birthday.

Its worth to remark that in K_3 we cannot replace “./name/@first, ./name/@last” by “./name” since in this case the key values are XML trees rather than a *data* node (i.e., a text). \square

3. TREE TRANSDUCERS FOR XML

We consider an XML tree \mathcal{T} that should respect a given schema and some key constraints. We introduce an unranked tree transducer capable of verifying both the schema and the key validity. This transducer extends the tree automaton of [7] by associating an output function to each transition rule. In this way, we allow values to be carried up from leaves to the root. In the following we formalize the concepts of output function and unranked tree transducer.

Definition 3.1 - Output function: Let \mathbf{D} be an infinite (recursively enumerable) domain and $\mathcal{T} = (t, type, value)$ be an XML tree. An *output function* f takes as arguments: (i) a position $p \in dom(t)$, (ii) a set s of pairs (att, l_v) where att is a tag associated to a list $l_v \in \mathbf{D}^*$ and (iii) a list l of items in \mathbf{D} . The result of applying $f(p, s, l)$ is a list of items in \mathbf{D} . In other words, $f : dom(t) \times \mathcal{P}(\Sigma \times \mathbf{D}^*) \times \mathbf{D}^* \rightarrow \mathbf{D}^*$. \square

The notation \mathbf{D}^* denotes all the lists of items in \mathbf{D} .

Definition 3.2 - Unranked bottom-up tree transducer (UTT):

An UTT over Σ and \mathbf{D} is a tuple $\mathcal{U} = (Q, \Sigma, \mathbf{D}, Q_f, \Delta, \Gamma)$ where Q is a set of states, $Q_f \subseteq Q$ is a set of final states, Δ is a set of transition rules and Γ is a set of output functions. For each transition rule in Δ , there is an output function f in Γ .

Each transition rule in Δ has the form $a, S, E \rightarrow q$ where (i) $a \in \Sigma$; (ii) S is a set of two disjoint sets of states, i.e., $S = \{S_{compulsory}, S_{optional}\}$ (with $S_{compulsory} \subseteq Q$ and $S_{optional} \subseteq Q$); (iii) E is a regular expression over Q and (iv) $q \in Q$. Each output function in Γ has the form $f(p, s, l) = l_1$ as in Definition 3.1. \square

Now, we consider the execution of an UTT on a Σ -valued tree t (in \mathcal{T}). As t represents an XML document, the children of any position $p \in dom(t)$ can be classified into two groups: those that are unordered, corresponding to the attributes of the node, and those that are ordered, corresponding to the sub-elements.

The transducer states two types of constraints: schema constraints and key constraints. Schema constraints are stated by the transition rules in Δ . In order to verify these constraints, i.e., to assume a state q at position p , the transducer \mathcal{U} performs the following tests:

1. If p has attribute children then the states assumed for them should correspond to those specified by the sets in S , namely, $S_{compulsory}$ and $S_{optional}$, corresponding, respectively, to p 's children that *must* appear in the tree and those that *may* appear³.
2. If p has element children then the concatenation of the states assumed at them must belong to the language generated by the regular expression E .

Key constraints are expressed by the output functions in Γ (see Algorithm 3.1). As the tree is to be processed bottom-up, the basic task of the output functions is to define the values that will be passed to the parent position, during the run.

Definition 3.3 - A run of \mathcal{U} on a finite tree t : Let t be a Σ -valued tree and $\mathcal{U} = (Q, \Sigma, \mathbf{D}, Q_f, \Delta, \Gamma)$ be an UTT. A *run* of \mathcal{U} on t is: (i) a tree $r : dom(r) \rightarrow Q$ such that $dom(r) = dom(t)$ and (ii) a function $\ell : dom(r) \rightarrow \mathbf{D}^*$, defined as follows:

For each position p whose children are those at positions⁴ $p_0, \dots, p(n-1)$ (with $n \geq 0$), we have $r(p) = q$ and $\ell(p) = l$ if and only if all the following conditions hold:

1. $t(p) = a \in \Sigma$.
2. There exists a transition $a, S, E \rightarrow q$ in Δ with an associated output function f in Γ .
3. There exists an integer $0 \leq i \leq (n-1)$ such that the children of p (i.e., the positions $p_0, \dots, p(n-1)$) can be classified⁵ according to the following rules:
 - (a) the positions $p_0, \dots, p(i-1)$ are members of a set $posAtt$ (possibly empty) and
 - (b) the positions $p_i, \dots, p(n-1)$ are members of a set $posEle$ (possibly empty) and
 - (c) every children of p is a member of $posAtt$ or of $posEle$ but no position is in both sets.

³These sets correspond to the *required* and *implied* attributes of a DTD.

⁴The notation $p(n-1)$ indicates the position resulting from the concatenation of the position p and the integer $n-1$. If $n=0$ the position p has no children.

⁵In an XML tree, the children at positions $p_0, \dots, p(i-1)$ correspond to attributes and the positions $p_i, \dots, p(n-1)$ correspond to elements.

4. The tree r and the function \mathcal{L} are already defined for positions $p0, \dots, p(n-1)$. We suppose $r(p0) = q_0, \dots, r(p(n-1)) = q_{n-1}$ and $\mathcal{L}(p0) = l_0, \dots, \mathcal{L}(p(n-1)) = l_{n-1}$.
5. The word $q_i \dots q_{n-1}$, composed by the concatenation of the states associated to the positions in $posEle$, belongs to the language generated by E .
6. The sets of S ($S_{compulsory}$ and $S_{optional}$) respect the following properties:
 - (a) $S_{compulsory} \subseteq \{q_0, \dots, q_{i-1}\}$ and
 - (b) $(\{q_0, \dots, q_{i-1}\} \setminus S_{compulsory}) \subseteq S_{optional}$.
7. Given $s = \{(t(pj), l_j) \mid 0 \leq j \leq (i-1), l_j \neq []\}$, the output associated to position p is

$$l = \mathcal{L}(p) = f(p, s, \text{concat}(l_i, \dots, l_{n-1})).$$

We say that a run r is *successful* if $r(\epsilon)$ is a final state of the automaton. The *output of a run* is given by $\mathcal{L}(\epsilon)$. \square

For a given XML tree, the existence of a successful run of an UTT implies that the document conforms to the DTD [7].

Notice that, in step (7) of Definition 3.3, the output function for each node is defined in terms of the position p (the first argument of the function f) and the result of the output functions of its children (the second and third arguments). The argument s corresponds to the set of pairs (att, l) . This set is formed by pairs containing the name and the outputs of the attributes of p . The third argument is obtained by concatenating the outputs coming from the sub-elements of p .

In what follows, we aim to construct the output function of an UTT, in such a way that the value associated to the root after a successful run ($\mathcal{L}(\epsilon)$) is a list containing a truth value, which will be true if and only if the key constraint is verified.

In order to verify a key $K = (P, (P', \{P_1, \dots, P_k\}))$ we should:

1. Collect the values associated to the positions defined by the paths P_1, \dots, P_k . Carry up these values taking into account if they correspond to attributes or elements.
2. At the target positions defined by P' , group the values for each key, in lists (of k elements). Carry up these lists to the context level.
3. At context positions defined by P , verify the uniqueness condition. Carry up a boolean value denoting the result of this test.
4. At the root, calculate the conjunction of these boolean values.

The above operations should be performed for all paths leading from the root to a key node. Values not belonging to these paths should be discarded.

Context, target and key nodes in K are defined in a top-down fashion. In order to identify these nodes with a bottom-up automaton, we must traverse the paths stated by K in reverse. We keep a representation of the reversed paths in the form of finite automata.

Before presenting the algorithm that translates keys into output functions, we give an example of such translation. We use the notation $M.e$ to represent a configuration of the automaton M , i.e., the current state e of the automaton M .

Example 3.1 We consider the verification of K_3 of Example 2.1 over the tree of Figure 1. We suppose an UTT whose transition rules represent some schema constraints and we want to add to it the output functions implementing the verification of K_3 . The finite state automata (FSA) associated to K_3 are the ones given in Figure 3. Notice that M and M' represent respectively the paths

$/politicPos/college$ and $/person$ of K_3 in reverse. The automaton M'' represents the disjunction of the paths $./name/@first$, $./name/@last$ and $./birth$ in reverse.

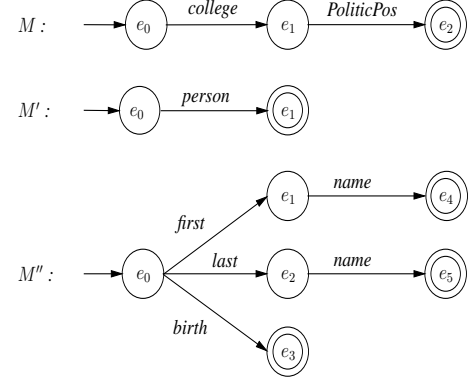


Figure 3: Automata corresponding to the paths of K_3 in reverse.

Following Figure 1, we consider firstly how the values concerning the element "person" at position 011 are carried up.

- 1) For the data nodes, the output functions have to get the data values, as well as to initiate the execution of the automaton to recognize the key paths. The output function, in this case, will return a singleton list, containing a pair (the initial configuration of the key automaton M'' and the value of the node):

$$\begin{aligned}
 f(011000, \emptyset, []) &= [(M''.e_0, [Pierre])]; \\
 f(011010, \emptyset, []) &= [(M''.e_0, [Smith])]; \\
 f(01110, \emptyset, []) &= [(M''.e_0, [01/01/51])].
 \end{aligned}$$

- 2) The fathers of the data nodes mentioned in the previous step have key labels. For each of them the output function must transmit this information together with the value received from its child. Doing that, it executes a first transition of the key FSA M'' , using each key label as input. For instance, reading the label *first* from state e_0 we reach state e_1 . Thus we should define:

$$\begin{aligned}
 f(01100, \emptyset, [(M''.e_0, [Pierre])]) &= [(M''.e_1, [Pierre])]; \\
 f(01101, \emptyset, [(M''.e_0, [Smith])]) &= [(M''.e_2, [Smith])]; \\
 f(0111, \emptyset, [(M''.e_0, [01/01/51])]) &= [(M''.e_3, [01/01/51])].
 \end{aligned}$$

where e_1, e_2, e_3 are the states reached by the key FSA M'' when reading the key labels (Figure 3).

Notice that data nodes which are not part of a key should not pass values to their fathers. Consider, for instance, the node at position 00. We have $f(00, \emptyset, [(M''.e_0, [President])]) = []$.

- 3) Next we consider position 0110. This position has two attribute children. In this case, the output function must promote the attribute values only if the current label belongs to the inversed key paths (represented by M''). In our case we have:

$$\begin{aligned}
 f(0110, \{(@first, [(M''.e_1, [Pierre])]), \\
 (@last, [(M''.e_2, [Smith])])\}, []) &= \\
 [(M''.e_4, [Pierre]), (M''.e_5, [Smith])].
 \end{aligned}$$

where e_4 and e_5 are the states reached by M'' when reading the label "name" from e_1 and e_2 , respectively (Figure 3).

4) For the node 011, the label "person" is the target label; it receives from its children two lists of values (one from attributes and one from elements). In order to transmit only key values, the output function of a target label should (i) select those that are preceded by a final state of the key automaton M'' , (ii) join them in a new list, and (iii) execute the first transition of the target FSA M' . In our case we have, for node 011:

$$f(011, \emptyset, [(M''.e_4, [Pierre]), (M''.e_5, [Smith]), (M''.e_3, [01/01/51])]) = [(M'.e_1, [Pierre, Smith, 01/01/51])].$$

The other target nodes will get their output values in a similar way. For instance, at position 012 we obtain the list $[(M'.e_1, [Mary, Dulac, 03/07/64])]$.

5) For the node 01, the label "college" is the context label, the output function should work in a similar way as in stage 4: it should (i) select the sublists that are preceded by a final state of the target automaton M' ; (ii) check if all these sublists are distinct and (iii) execute the first transition of the context FSA M . So, in our case, for node 01, we have:

$$f(01, \emptyset, [(M'.e_1, [Pierre, Smith, 01/01/51]), (M'.e_1, [Mary, Dulac, 03/07/64]), \dots]) = [(M.e_1, [B])].$$

where B is true iff all the sublists selected in the third argument of f are distinct.

6) The computation should continue up to the root, verifying whether the labels visited are recognized by the context FSA or not: in our example,

$$f(0, \emptyset, [(M.e_1, [b])]) = [(M.e_2, [b])]$$

where e_2 is the state reached by M when reading the label "Politico" (Figure 3).

7) At the root position the last output function should select the sublists that are preceded by a final state of the context FSA M and should return the conjunction of all boolean values in these sublists. In our example:

$$f(\epsilon, \emptyset, [(M.e_2, [b_1]), \dots, (M.e_2, [b_m])]) = [(M_F.e_f, [B])]$$

where B is true iff all b_i are true. The (final) configuration $M_F.e_f$ is introduced to keep the homogeneity of the lists returned by the output function. It corresponds to the final configuration of an automaton M_F accepting only the root label. \square

The following algorithm shows how output functions can be defined in order to represent a given key. Note that, since our UTT is an extension of a deterministic tree automaton having the same expression power of a non ambiguous DTD [7], a label $a \in \Sigma$ corresponds to a unique transition function and thus to a unique output function.

Algorithm 3.1 - Key constraints as output functions: Let $K = (P, (P', \{P_1, \dots, P_k\}))$ be a key. Let $M = \langle \Theta, \Sigma, \delta, e_0, F \rangle$ (respectively, $M' = \langle \Theta', \Sigma, \delta', e_0, F' \rangle$ and $M'' = \langle \Theta'', \Sigma, \delta'', e_0, F'' \rangle$) be the finite state automaton that recognizes the path P in reverse (respectively, the paths P' and $P_1 \mid \dots \mid P_k$ in reverse). Let $M_F = \langle \Theta_F, \{root\}, \delta_F, e_0, \{e_f\} \rangle$ be the automaton recognizing the path root (in reverse).

Let $\mathcal{U} = (Q, \Sigma, \mathbf{D}, Q_f, \Delta, \Gamma)$ be an UTT whose transition rules represent some schema constraints. The domain \mathbf{D} is defined as $(\Theta \sqcup \Theta' \sqcup \Theta'' \sqcup \Theta_F) \times \mathbf{V}^*$. We assume that the transition rules of \mathcal{U} have the general form $a, S, E \rightarrow q_a$. In order to express the key K , each transition rule of \mathcal{U} is associated to an output function defined according to its label a :

1. If $a = data$ (the rule has the form $data, \{\emptyset, \emptyset\}, \emptyset \rightarrow q_{data}$) then the output function is $f(p, s, l) = [(M''.e_0, [value(p)])]$.

2. If a is a target label then the output function is defined as: $f(p, s, l) = [(M'.\delta'(e_0, a), concat(fi lter_{key}(c_1, \dots, c_m)))]$ where:

$$[c_1, \dots, c_{j-1}] = \Pi_{pair}(orderByName(s)) \text{ and } [c_j, \dots, c_m] = l.$$

For the target nodes, the output list is composed by a pair containing (i) the configuration of the target automaton reached from its initial state e_0 by reading a and (ii) a list of all the values composing a key.

We impose an order to the pairs coming from attribute children (i.e., those contained in s); this is done by sorting them in the lexicographic order of their tags and then eliminating these tags.

The function " $fi lter_{key}$ " filters the key lists, leaving only the values associated to key positions. Notice that each c_k is a pair of an automaton configuration and a list of values. The filter selects the lists of values in the pairs whose configuration corresponds to a final state of M'' . The list operation " $concat$ " returns the concatenation of all its argument lists into one list.

Notice that at this step, the output function will return a singleton list, whose only element is a pair formed by a configuration of M' and a list of all the values that belong to the key.

3. If a is a context label then the output function is $f(p, s, l) = [(M.\delta(e_0, a), g(fi lter_{target}(l)))]$ where

$$g([v_1 \dots v_m]) = \begin{cases} [true] & \text{if } v_1 \dots v_m \\ & \text{are all distinct lists} \\ [false] & \text{otherwise} \end{cases}$$

The function " $fi lter_{target}$ " simply filters the target lists in a similar way as key lists were filtered in the previous case. For the context nodes, we must check that the lists formed at the target level are all different. The result of this level is a singleton list that contains a pair. This pair is formed by a configuration of M and a list containing a boolean value (the result of checking the validity of the key for each specific context).

4. If a is the root label then the output function is

$$f(p, s, l) = AND(fi lter_{context}(l)) \text{ where } AND([b_1], \dots, [b_m]) = [(M_F.e_f, [\bigwedge_{j=1}^m b_j])].$$

At the root level, we calculate the conjunction of the truth values that were obtained for each subtree rooted at the context level. The function " $fi lter_{context}$ " simply filters the context lists, as in the previous cases.

5. In all other cases (i.e., when $a \neq data$ and a is not a target label, nor a context label, nor the root) the output function is defined as:

$$f(p, s, l) = h([c_1, \dots, c_m])$$

where $[c_1, \dots, c_m]$ is the list of pairs obtained from the children of p , such that $[c_1, \dots, c_{j-1}] = \Pi_{pair}(orderByName(s))$ and $[c_j, \dots, c_m] = l$. In other words, key pairs coming from the attribute children of the node are sorted and then projected, as in step 2.

The function h is defined by the following procedure:

```

procedure h( $L$  : list of pairs);
var result : list of pairs;
begin
  result  $\leftarrow$  [ ];
  foreach  $c = (\mathcal{M}.e, v)$  in  $L$  //  $\mathcal{M}$  stands for  $M, M'$  or  $M''$ 
    if  $\delta(e, a) = e'$  is a transition in  $\mathcal{M}$  then
      result  $\leftarrow$  concat(result,  $[(\mathcal{M}.e', v)]$ );
  return result;
end;

```

This definition concerns the positions that are at the key levels or that does not belong to levels 1, 2 or 3 in Figure 2. The resulting list is formed by pairs, each of which contains an automaton configuration and a list of values. \square

The key verification is done in asymptotic linear time on the size of the document. Given the key constraint $(P, (P', \{P_1, \dots, P_k\}))$ the validation process consists in a bottom-up visit of each XML tree we want to validate. The copy, concat and filtering operations performed during this validation have constant time complexity $O(k)$. Thus, verifying if the lists are distinct, at the context level, takes time $O(c^2)$ where c is the number of target nodes, *i.e.*, those reachable by the path $/P/P'$. Note that c usually represents a very small number when compared to the size of the tree.

The following theorem states that tree transducer generated by Algorithm 3.1 validates the key constraints.

Theorem 3.1 Let $K = (P, (P', \{P_1, \dots, P_k\}))$ be an XML key over an XML tree T . Let \mathcal{U} be a tree transducer that expresses K according to Algorithm 3.1. Then,

$$T \text{ satisfies } K \text{ iff } \mathcal{E}(\epsilon) = [(M_F.e_f, [true])]. \quad \square$$

4. CONCLUSIONS

In this paper we show how both schema and key constraints can be represented and validated by bottom up tree transducers. Both constraints are verified in a single bottom-up visit of an XML tree. This approach can be very useful in the context of the incremental validation of updates to XML documents.

In [5, 9] we find different proposals for expressing keys. Some of them had been incorporated to schema languages such as XML-Schema [2] and Schematron [1]. In our work we use the notion of strong keys of [9]. Moreover, we consider that there is no inconsistency between key and schema. We refer to [4] as a survey on the problem of statically verifying the consistency of schema and key constraints.

Key validation is the subject of recent research [5, 10, 6]. In [5] a constraint language, designed to support incremental validation, is proposed. The incremental validation of constraints is done by translating constraints into logic formulas and then generating incremental constraint checking code from them. In [10] a key validator is proposed which works in asymptotic linear time in the size of the document. Our algorithm also has this property. In [10] (incremental) validation relies on the use of index. In contrast to our approach, schema constraints are not considered in [5, 10]. In [6] both schema and integrity constraints are considered in the process of generating XML documents from relational databases. They propose a formalism inspired by attribute grammars [12] with both synthesized (bottom-up evaluation) and inherited (top-down evaluation) data. Although some similar aspects with our approach can be observed, we place our work in a different context. In fact, we consider the evolution of XML data independently from any other

database sources (in this context both validation and re-validation of XML documents can be required).

We are currently pursuing the following lines of research:

- (i) The generalization of our method to treat more than one key constraint, as well as to treat more general schema definitions. We are in the process of verifying some properties respected by UTTs, such as their closure under intersection.
- (ii) The introduction of a notion similar to attribute inheritance to our output functions (they already implement synthesized attributes). This notion should be similar to those of L-attributed grammars, as used in compiler construction [3] and should be implemented in a single-pass bottom-up tree transducer. This feature will allow the validation of other kinds of constraints.
- (iii) The use of tree transducers for the incremental validation of key constraints. We aim at the extension of the incremental validation method for XML documents under schema constraints proposed in [8]. This method relies on the execution of a tree automaton only on the part of the XML tree affected by the update.

5. REFERENCES

- [1] The Schematron: An XML structure validation language using patterns in trees. Available at <http://www.ascc.net/xml/resource/schematron>.
- [2] XML schema. Available at <http://www.w3.org/XML/Schema>.
- [3] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley, 1988.
- [4] M. Arenas, W. Fan, and L. Libkin. On verifying consistency of XML specifications. In *ACM Symposium on Principles of Database System*, 2002.
- [5] M. Benedikt, G. Bruns, J. Gibson, R. Kuss, and A. Ng. Automated update management for XML integrity constraints. In *Program Language Technologies for XML (PLANX02)*, 2002.
- [6] M. Benedikt, C-Y Chan, W. Fan, J. Freire, and R. Rastogi. Capturing both types and constraints in data integration. In *SIGMOD, San Diego, CA*, 2003.
- [7] B. Bouchou, D. Duarte, M. Halfeld Ferrari Alves, and D. Laurent. Extending tree automata to model XML validation under element and attribute constraints. In *ICEIS*, 2003.
- [8] B. Bouchou and M. Halfeld Ferrari Alves. Updates and incremental validation of XML documents. Submitted paper, 2003.
- [9] P. Buneman, S. Davidson, W. Fan, C. Hara, and W.C. Tan. Keys for XML. In *WWW10, May 2-5*, 2001.
- [10] Y. Chen, S. Davidson, and Y. Zheng. Validating constraints in XML. Technical Report MS-CIS-02-03, Department of Computer and Information Science, University of Pennsylvania, 2002.
- [11] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. *Tree Automata Techniques and Applications*. Available on: <http://www.grappa.univ-lille3.fr/tata>, 1997 (new version 2002).
- [12] P. Deransart, M. Jourdan, and B. Lorho. Attribute grammars: Definitions, systems and bibliography. Number 323 in LNCS - Lecture Notes in Computer Science, 1988.

A Framework for Estimating XML Query Cardinality

Carlo Sartiani
Dipartimento di Informatica - Università di Pisa
Via Buonarroti 2, Pisa, Italy
sartiani@di.unipi.it

ABSTRACT

Tools for querying and processing XML data are increasingly available. In this context, the estimation of the cardinality of queries on XML data is becoming more and more important. The information provided by a query result estimator can be used as input to the query optimizer, and as an early feedback to user queries. Existing estimation models for XML queries focus on particular aspects of XML querying, such as the estimation of path and twig expression cardinality, and they do not deal with the problem of predicting the cardinality of general XQuery queries. This paper presents a framework for estimating XML query cardinality. The framework provides facilities for estimating result size of FLWR queries, hence allowing the model designer to concentrate her efforts on the development of adequate and accurate, while concise, statistic summaries for XML data. The framework can also be used for extending existing models to a wider class of XML queries.

1. INTRODUCTION

The last few years have seen the emerging and the wide diffusion of XML. As a consequence, tools for storing, querying, and manipulating XML data are nowadays increasingly available. In the context of XML data management system, the estimation of the cardinality of queries on XML data is becoming more and more important: the information provided by a query result estimator can be used as input to the query optimizer, as an early feedback to user queries, as well as input for determining an optimal storage schema [4].

As for many other common database tasks, the peculiar nature of XML makes result size estimation more difficult than in the relational context; in particular, the (very) non-uniform distribution of tags and data, together with the dependencies imposed by the tree structure of XML data, makes not so reasonable the usual hypothesis used in relational size estimators.

Existing estimation models for XML queries focus on particular aspects of XML querying, such as the estimation of

path and twig expression cardinality, and they do not deal with the problem of predicting the cardinality of general XQuery queries. Moreover, these models usually estimate the *raw* cardinality of queries (e.g., the number of nodes returned by a path, or the number of tuples generated by tuple-based execution engines), without providing any information about the distribution or the nature of the expected result, which is necessary for correctly predicting the size of further operations.

Our Contribution. This paper presents a framework for estimating the cardinality of FLWR XQuery queries. The framework provides facilities for estimating the raw size as well as the data distribution of query result. In particular, it offers algorithms for estimating the size of sets built by the *let* clause of XQuery, for correlating the estimation of twig branches, and for applying predicate selectivity factors according to the distribution of data.

The framework allows the model designer to concentrate her efforts on the development of adequate and accurate, while concise, statistic summaries for XML data. It can also be used for extending existing models to a wider class of XML queries.

2. ISSUES IN RESULT SIZE ESTIMATION

Result size estimation for XML queries requires the system to predict the cardinality of any query in the language. Referring to a fragment of XQuery [3] without recursive functions, the most problematic aspects concern the estimation of path and twig cardinality, the estimation of predicate selectivity, as well as the estimation of group cardinality (*let* binder of XQuery). While path and twig estimation is a peculiar issue of XML and semistructured query languages, predicate and group cardinality estimation are well-known problems in database theory and practice. Nevertheless, these problems receive new strength from the irregular nature of XML, as briefly discussed above.

Irregular tree or forest structure. XML data can be seen as node-labeled trees or forests; these trees, being commonly used for representing semistructured data, usually have a deeply nested structure, and are far from being well-balanced. Moreover, the same tag may occur in different parts of the same document with a different semantics, e.g., the tag *name* under *person* and the tag *name* under *city*.

The irregular and overloaded structure of XML documents influences cardinality estimation, since the location of a node inside a tree may determine its semantics, and, then,


```

<root>
  <persons>
    <person> <name>
      <fullname> Caius Julius Caesar
      </fullname>
      <gensname> Julia </gensname>
    </name>
    <city> Roma </city>
  </person>
</persons>
<cities>
  <city> <name>
    <ancientname> Roma </ancientname>
    <currentname> Roma </currentname>
  </name>
  <nick> Caput Mundi </nick>
  <nick> Eternal City </nick>
</city>
  <city> <name> New York </name>
  <nick> The Big Apple </nick>
</city>
</cities>
</root>

```

Figure 1: A sample XML document

its relevance in operations like path and predicate evaluation. For example, consider the XML document shown in Figure 1. The structure of the `name` element under `person` is quite different from the semantics of New York’s `name`, hence the evaluation of any query operation starting from `name` elements should take this into account.

Non-uniform distribution of tags and values. The irregular structure of XML data, together with their hierarchical tree-shaped nature, leads to the *non-uniform* distribution of tags and values in XML trees. XML non-uniformity is further strengthened by the presence of structural dependencies among elements (e.g., `name` depends on `person`, etc). As a consequence, a prediction model should track the provenance of estimated matching elements.

These typical features of XML influence the nature and the “complexity” of the previously cited estimation problems, and give rise to new requirements for prediction models. Hence, a closer look to these problems is necessary.

Path and twig cardinality estimation. Path and twig expressions are used in XQuery and in many other XML query languages for retrieving nodes from a XML tree, and for binding them to variables for later use.

The main difficulties in cardinality estimation for path and twig expressions come from the need to reduce the prediction errors induced by joins (paths and twigs are usually translated in sequences of joins), and, for twigs only, from the need to correlate results coming from different branches.

Predicate selectivity estimation. The estimation of predicate selectivity is a well-known problem in database theory and practice. The most effective and accurate solutions rely on histograms for capturing the distribution of values in the data, and on the use of the uniform distribution when nothing is known about the data involved in the predicate.

In the context of XML, predicate selectivity estimation poses new challenges. First, XML data are usually distributed in a (very) non-uniform way, hence the use of the uniform distribution can lead to many potential errors. Sec-

ond, the selectivity of a predicate such as `data($n) θ value` depends not only on θ and *value*, but also on a) the nodes bound to $\$n$, which may be heterogeneous, b) the semantics of those nodes (e.g., `name` under `person` is quite different from `name` under `city`), and c) the “region” of the document where those nodes appear.

Many existing prediction models, while very sophisticated and accurate, return raw numbers as result of the estimation. Raw numbers, denoting the cardinality of matching nodes in the data tree, do not carry sufficient information for the estimation of subsequent predicates being accurate, hence making the enclosing models not so accurate.

Groups. Unlike SQL, XQuery misses explicit constructs for performing `groupby`-like operations.¹ Nevertheless, the `let` binder can be used for creating heterogeneous sets of nodes, hence for building, together with nested queries, groups and partitions. The `let` binder, unlike the `for` binder, accumulates each node returned by its argument into a set, which is then bound to the binding variable. For example,

```

for $c in input()//city,
let $n_list := $c/name,

```

returns, for each city, the list of its names (ancient as well as modern ones).

Estimating the cardinality of the `let` binder requires the system a) to estimate the number of distinct groups created, b) to correlate each group to the variables on which it depends ($\$n_list$ depends on $\$c$), and c) to estimate the distribution of nodes and values into each group. This information is necessary since the groups created by the `let` binder can be used as starting point for further navigational operations, as argument for aggregate functions or for predicates.

The estimation of group cardinality is one of the missing points in current XML prediction model. For what is known to the author, no existing model for XML query languages faces this problem, hence the support of group cardinality estimation at the framework level becomes a *must*.

3. THE FRAMEWORK

Dealing with the estimation problems described in the previous Section requires the prediction model to estimate not only the raw cardinality of results, but also their distribution. The following Sections will explain the estimation approach employed in the framework.

3.1 Match Occurrences

The facilities offered by the framework rely on the notion of *match occurrence*. A match occurrence o is a triple (l, r, m) , where l is a node label, r is the *region* where o occurs, and m is the multiplicity of the occurrence; a match occurrence $o = (l, r, m)$, then, says that m nodes labeled l and belonging to region r are part of the result. The notion of region is wide enough to accomplish the needs of different prediction models: it may be the type of the node (*intensional* region), the type of the node together with its location in the originating document (*mixed* region, e.g.,

¹We are aware of proposals for extending XQuery with explicit `groupby` clauses. Due to time and space constraints, we cannot discuss such proposals in this paper.

```

type DB = root[persons[Person*], cities[City*]]
type Person = person[PersonName]
type PersonName = name[fullname[String],
                      gensname[String]?]
type City = city[OldCityName, OldCityNick*|
                 NewCityName, NewCityNick*]
type OldCityName = name[ancientname[String]+,
                       modernname[String]]
type OldCityNick = nick[String]
type NewCityNick = nick[String]
type NewCityName = name[String]

```

Figure 2: A schema for the sample document

the node type and its bucket in the corresponding *structural* histogram in StatiX [4]), or the location of the node in the originating database (*extensional* region, e.g., the grid cell in the related *position* histogram in TIMBER [6]). Regions, hence, can express both intensional and extensional concepts, and are the main tool for describing the distribution of data. Regions are also the basic tools for correlating match occurrences coming from different branches of a twig; as shown in Section 3.4, correlation by means of regions is one of the most interesting and useful facilities offered by the framework. The following example illustrates the notion of match occurrence.

Example 3.1 Consider the XML document shown in Figure 1, and assume that the path expression `input()//name` is being evaluated on it. Assuming an extensional partitioning of the tree based on height, the result distribution is the following:

$$\{(name, 4, 3)\}$$

This sequence denotes the occurrence of three `name` elements at level 4.

Assuming an intensional partitioning of data based on the schema of Figure 3.1, the result would be the following:

$$\{(name, PersonName, 1), (name, OldCityName, 1), (name, NewCityName, 1)\}$$

3.2 ECLSs and ETLs

Given the notion of match occurrence, the main idea behind the framework is to use estimation functions that manipulate sequences of match occurrences called Extended Context Label Sequences. An ECLS, hence, is a sequence where match occurrences for a given path are accumulated. The following example briefly illustrates this point.

Example 3.2 Consider the query fragment of the previous example, and the intensional partitioning hypothesis. Then, the ECLS generated for the path expression would be the following.

$$\{(name, PersonName, 1), (name, OldCityName, 1), (name, NewCityName, 1)\}$$

ECLSs are bound to variable symbols to form a data structure called ETLs (Extended Tuple Label Sequence), which collects estimations about all tuples produced by the system. Two key points must be noted: first, each variable is bound to a sequence (possibly, a singleton) of ECLSs, in order to support the cardinality estimation of groups; second,

the ECLS associated with a variable contains all the match occurrences found for the variable, hence only one ETLs is generated during query cardinality estimation.

The following example shows a sample ETLs.

Example 3.3 Consider the following query clause:

```

for $c in input()//city,
    $n in $c/name

```

The estimation of the cardinality for this clause (type regions) would generate the following ETLs.

$$\begin{aligned} \{ \$c : < \{ (city, City, 2) \} >, \\ \$n : < \{ (name, OldCityName, 1), \\ (city, NewCityName, 1) \} > \} \end{aligned}$$

3.3 Cardinality Notions

One key point in any estimation model is what cardinality notion is used, i.e., how the size measures returned by the model should be interpreted. The most common operational model for XML queries (at least, for queries involving variables) is based on the construction of tuples carrying the values bound to variables [1]; since usual optimization heuristics are based on the minimization of the number of granules generated during query evaluation, the *natural* cardinality notion is the number of such granules (e.g., [4]).

The proposed framework embodies the vision of intermediate tuple generation, hence it supports the number of generated tuples as cardinality notion. The way this number is computed from ETLs depends on the specific model being considered. However, the framework contains a general purpose cardinality function $\| \cdot \|$.

This notion of cardinality, while widely used, provide no information about the shape and structure of XML data returned by a query, hence models relying on this raw notion only are exposed to (many) potential estimation errors.

3.4 Correlation

The correlation problem refers to the need of correlating estimations coming from distinct branches of the same twig. Consider, for example, the following query clause:

```

for $x in input()/a,
    $y in $x/b,
    $z in $y/c,
    $w in $y/d

```

This clause matches a two-branch twig against an hypothetical document; in order to correctly predict the number of tuples in the result it is necessary to correlate the estimation for the branch `b/c` with the estimation for the branch `b/d`. Without such a correlation, computing the number of tuples represented by a given ETLs would require the model to multiply the multiplicity of `$z` with that of `$w` (cross product hypothesis), hence introducing many potential errors.²

Twig branch correlation can be performed by using regions. The idea is the following. Once estimated the cardinality of the twig branches, the number of generated tuples

²The framework makes the implicit assumption that twig cardinality is not directly stored in statistics, but, instead, computed from path cardinality. This assumption is common in statistical models for XML queries.

can be obtained from the resulting ETLs by identifying the variables having a common root variable, and multiplying the multiplicity of those match occurrences sharing the same parent region.

Example 3.4 Consider the following query fragment:

```
for $c in input()//city,
  $y in $c/name,
  $nick in $c/nick,
```

This query fragment retrieves, for each *city* element in the database, its name and the list of its nicknames. By evaluating this clause on the sample document of Figure 1, the framework produces the following ETLs (intentional partitioning):

```
{ $c :< { (city, City, 2) } >,
  $n :< { (name, OldCityName, 1) },
          (name, NewCityName, 1) } > }
$nick :< { (nick, OldCityNick, 2),
           (nick, NewCityNick, 1) } >
```

Without any correlation, the predicted number of tuple would be 6, which is clearly wrong (the right number is 3). By correlating *nick* elements and *name* elements on the basis of their parent region, the framework can correctly estimate the number of tuples as 3. ■

Correlation affects the tuple cardinality computing function, as well as other facilities of the framework: the cardinality of an ETLs is computed by multiplying the multiplicity of correlated match occurrences only, independent variables (e.g., variables bound to different documents) being considered as fully correlated (each match occurrence is correlated to any other).

3.5 Group cardinality estimation

Group cardinality estimation refers to the problem of estimating the dimension of sets created by the *let* binder, and, more generally, by the use of free path expressions outside the binding clauses for and *let*. In Section 2 three main issues were identified about groups: the estimation of the number of distinct groups; the correlation between each group and the variable instance, which it depends on; and the estimation of the distribution of data into each group.

The number of groups created by the *let* binder is equal to the multiplicity of the variable on which the groups depend. Consider, for example, the query fragment shown below:

```
for $c in input()//city,
let $n_list := $c/name,
```

For each *city* node bound to *\$c*, a distinct *\$n_list* group is created.

Thus, the framework computes the number of groups by using the following function, which sums multiplicity of match occurrences for a single variable:

$$\|etls(\$c)\| = \begin{cases} \sum_{(l, m_l, r_l) \in e} m_l & \text{if } etls(\$c) = \langle e \rangle \\ n & \text{if } etls(\$c) = \langle e_1, \dots, e_n \rangle \end{cases}$$

The second case in the definition is necessary when the root variable of the *let* binder is itself the result of the application of another *let* binder, as in the fragment shown below:

```
for $c in input()//city,
let $n_list := $c/name,
let $notes := $n_list/notes,
```

In this particular case, the number of groups is equal to the number of groups of the root variable (*\$n_list*).

Once computed the number of groups, the framework must create each group and estimate the data distribution inside it. The group creation algorithm is based on the correlation function, and, performs the following step: the algorithm, first, collects all match occurrences of the *let* path expression in a set \mathcal{S} , and creates m empty groups, where m is the estimated number of groups; then, it correlates each occurrence o in \mathcal{S} with the root match occurrences, and distributes them accordingly. The following example illustrates the group creation process.

Example 3.5 Consider our well-known query fragment:

```
for $c in input()//city,
let $n_list := $c/name,
```

By estimating for clause cardinality on the sample document of Figure 1, the framework generates the following ETLs:

```
{ $c :< { (city, City, 2) } > }
```

The list of match occurrences collected for the *let* path expression (intentional partitioning) is the following:

```
{ (name, OldCityName, 1), (name, NewCityName, 1) }
```

The framework creates 2 groups, and distributes match occurrences as shown below.

```
{ $n_list :< { (name, OldCityName, 1) },
              { (name, NewCityName, 1) } > }
```

■

The group creation algorithm has $O(n^2)$ worst case time complexity, where n is the number of match occurrences. This moderate time complexity is the price paid for obtaining a very accurate size estimation.

3.6 Predicate selectivity estimation

The estimation of predicate selectivity for XML queries shows two main issues. The first issue concerns the estimation process itself as well as the nature of selectivity factors. As already discussed, XML documents have an irregular structure, where tags and values are distributed in a way far from being uniform. As a consequence, the uniform distribution hypothesis is not suitable for predicates on XML data. Moreover, queries can contain *value* predicates (e.g., *data(\$y) > 1982*, where *\$y* is bound to *year* elements) and *structural* predicates about the existence of children nodes with a given label (e.g., *book[publisher]*); finally, even if we consider only value predicates, variables can be bound to heterogeneous nodes, for instance *\$a* bound to *author* and *publisher* nodes. Hence, the selectivity factor for a given predicate P should be a function of structural information.

Given that, the framework supports selectivity factors as functions of the tag and the region of a given match occurrence. This choice allows the framework to take structural information (even type information if an intentional partitioning is used) into account, hence increasing the accuracy

of the cardinality estimation. The following example clarifies this point.

Example 3.6 Consider the following query fragment:

```
for $c in input()//city,
  $n in op:union($c//ancientname, $c//currentname)
where data($n) = "Monticello"
```

The predicate `data($n) = "Monticello"` applies to `ancientname` and `currentname` elements, and its selectivity factor depends on the tag of the subject node, since values can have different distributions in `ancientname` and `currentname` elements (e.g., "Monticello" is a quite common name for small Italian villages, even though their Latin or medieval names were slightly different). ■

Selectivity factors for unary predicates can be defined as follows.

Definition 3.7 Given a unary predicate P , the selectivity factor of P $psf[P]$ is a function

$$psf[P] : label \times region \rightarrow [0, 1]$$

that, given a label l and a region r , returns a real number belonging to $[0, 1]$.

This definition naturally leads to histogram-based selectivity factors, even though the model designer is free to choose the preferred way of collecting and storing statistics. Histograms can be built and managed by using well-known techniques, without the need for particular changes.

The following example illustrates unary predicate selectivity factors.

Example 3.8 Consider the query fragment of the previous example, where variable $\$n$ is bound to `ancientname` as well as `currentname` elements. Assuming that statistics are gathered from a bigger document than that of Figure 1, the selectivity factor for the predicate $P \equiv data(\$n) = "Monticello"$ could be the following:

$$psf[P] = \begin{cases} (ancientname, r_1) \rightarrow 0.2 \\ \dots \\ (ancientname, r_5) \rightarrow 0.07 \\ (currentname, r_6) \rightarrow 0.75 \\ \dots \\ (currentname, r_9) \rightarrow 0.67 \end{cases}$$

The definition of predicate selectivity factors extends from unary predicates to binary and n-ary predicates. For binary and n-ary predicates, selectivity factors are built on a single variable, whose choice is left to the specific model.

The second main issue about predicate selectivity estimation concerns the way these factors are applied to ETLs in the framework. For instance, given the well-known predicate `data($n) = "Monticello"`, the values returned by $psf[P]$ are used for decreasing the multiplicity of match occurrences bound to $\$n$. Still, this is not sufficient, as the predicate cuts the number of twig instances collected on data; hence, the multiplicity decrease should be applied also to any directly or indirectly dependent variable ($\$c$ only in the example). For applying this decrease and preserving accuracy, multiplicity decrease propagation should be based on the correlation mechanism previously described.

As a consequence, the framework offers a predicate selectivity factor application function, which, starting from the predicate variable, scans match occurrences, applies the right factor, and reapplies the transformation to directly or indirectly dependent variables. The following example shows how selectivity factors are applied.

Example 3.9 Consider again the query fragment of Example 3.6, and the selectivity factor of Example 3.8; assume that the `for` clause estimation returns the following ETLs:

```
{ $c :< { (city, r1, 2), (city, r3, 1), (city, r4, 1) } >,
  $n :< { (ancientname, r5, 2), (currentname, r6, 1),
          (currentname, r9, 1) } > }
```

By applying the selectivity factor of Example 3.8, the framework generates the following ETLs:

```
{ $c :< { (city, r1, .14), (city, r3, .75), (city, r4, .67) } >,
  $n :< { (ancientname, r5, .14), (currentname, r6, .75),
          (currentname, r9, .67) } > }
```

(we assume that r_5 correlates to r_1 , and that r_6 and r_9 correlate to r_3 and r_4 respectively). ■

4. COMPLEXITY ISSUES

In this Section we will briefly expose some complexity results about the most important algorithms of the framework, namely the correlation function, the selectivity factor propagation algorithm, the group cardinality estimation algorithm, and the cardinality computing function; due to lack of space these algorithms are reported in [5].

The correlation function determines whether two match occurrences o_1 and o_2 are correlated by a common ancestor in the sequence of ECLs associated to a parent variable $\$root$. The problem is equivalent to the problem of finding a common ancestor in a portion \mathcal{R} of a graph \mathcal{G} , where nodes are labeled with pairs $(label, region)$, and edges are determined by the structure of the document and by the region partitioning scheme. Due to the constraint represented by \mathcal{R} , NCA (Nearest Common Ancestor) should be modified to be used in this context: in particular, by endowing each pair (l, r) , during statistics collection, with its reachability sets in \mathcal{G} (both ancestors and descendants), this problem can be solved in time $O(n)$, where n is the number of match occurrences bound to the variable $\$root$. These sets are used for lowering time complexity of other algorithms too.

In a similar way, the selectivity factor propagation problem can be reduced to the exploration of the (l, r) graph; hence, by exploiting the reachability set of (l, r) pairs and supplementary data structures, the propagation can be performed in $O(m + n)$ time, where n is the number of match occurrences and m is the number of edges in the graph. Due to the wide space requirements of the auxiliary data structures, in [5] we describe a more space-conscious algorithm with time complexity $O(n^2)$.

The group cardinality estimation algorithm is heavily based on the correlation function, since match occurrence distribution is performed according to the correlation relation. The algorithm just scans, for any match occurrence in the let path expression, the list of the root match occurrences, in order to find the target groups; as a consequence, the algorithm has $O(n^2)$ worst case time complexity, where n is the number of root match occurrences.

The cardinality computing function identifies trees of matching occurrences in the ETLs, and then computes their cardinality. This is performed by computing the cardinality of

each twig in the query Q , and by combining them in the cardinality of the whole query; the calculation relies again on the correlation function, hence leading to a $O(n^3)$ worst case time complexity.

5. RELATED WORKS

TIMBER result size model. In [6] authors propose two models for estimating the number of matches of twig queries. The proposed models rely on *position histograms* for predicates in a set \mathcal{P} . Position histograms can be used for estimating the raw cardinality of simple ancestor/descendant queries. The first model exploits position histograms only, while the second one also uses *coverage histograms*, a kind of structural information for increasing the accuracy of the prediction; unlike the first model, however, this one can be used only when the schema information is available, and, in particular, when the ancestor predicate in a pattern (P_1, P_2) satisfies the *no-overlap* property.

The model based on coverage histograms is much more accurate than the model based only on position histograms; unfortunately, its applicability is limited, and, in particular, it cannot be exploited in recursive documents, where the two proposed models behave badly. Moreover, the estimations are limited to ancestor/descendant paths, and it is not clear how they can be extended to complex twigs involving also parent/child relationships. Finally, the model only deals with twig matching, hence ignoring critical issues such as iterators, binders, nested queries, etc.

Niagara's models. In [2] authors present the path expression selectivity estimation models employed in Niagara. The models can be used to compute the selectivity of path expressions of the form $a/b/\dots/f$, i.e., XPath patterns without closure operators ($//$) and inline conditions; moreover, the models cannot be applied to twigs.

The first model is based on a structure called *path tree*. Since a path tree may have the same size as the database (e.g., when paths in the database are distinct from each other), summarization techniques should be applied to constrain the size of the path tree to the available main memory.

The second model is based on a more sophisticated statistic structure called *Markov table*. This table, implemented as an ordinary hash table, contains any distinct path of length up to m ($m \geq 2$), and its selectivity. As for path trees, the size of a Markov table may exceed the total amount of available main memory, hence summarization techniques are required.

The proposed approaches are quite simple and effective: the Markov table technique, in particular, delivers an high level of accuracy (much more than the pruned suffix tree methods). Unfortunately, they are limited to simple path expressions, and there is no clear way to extend them to twigs or predicates.

StatiX statistics model. In [4] authors describe a methodology for collecting statistics about XML documents; the proposed approach is applied in LegoDB for providing statistics about XML-to-relational storage policies, and, to a less extent, in the Galax system for predicting XML query result size.

The StatiX approach aims to build statistics capturing

both the irregular structure of XML documents and the *non-uniform* distribution of tags and values within documents. To this purpose, it relies on the schema associated to each document. Indeed, given a XML Schema description \mathcal{S} describing a XML document \mathcal{T} , StatiX builds $O(m + n)$ histograms, where m and n are respectively the number of edges and nodes in the graph representation of \mathcal{S} . StatiX histograms fall into two categories: *structural* histograms, which describe the distribution and the correlation of non-terminal type instances, and *value* histograms, which, as in the relational case, represent value distribution of simple elements (i.e., elements whose content is a base value).

The StatiX system can tune statistics granularity by applying *conservative* schema transformations to the original XML Schema description, i.e., transformations preserving the class of described documents, and not introducing ambiguity.

6. CONCLUSIONS

This paper has described a framework for estimating the cardinality of XML queries. The proposed framework offers tools and algorithms for predicting not only the raw size of query results, but also the distribution of data inside them, hence making the prediction of the size of subsequent operations more accurate. The facilities offered by the framework range from group cardinality estimation to twig branch correlation, and selectivity factor application; by relying on these facilities, the model designer can focus on the definition of accurate and concise statistic summaries.

7. ACKNOWLEDGMENTS

The author would like to thank Dario Colazzo for his support during and after the writing of this paper.

8. REFERENCES

- [1] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web: From Relations to Semistructured Data and XML*. Morgan Kaufman, 1999.
- [2] A. Aboulnaga, A. R. Alameldeen, and J. F. Naughton. Estimating the selectivity of xml path expressions for internet scale applications. In P. M. G. Apers, P. Atzeni, S. Ceri, S. Paraboschi, K. Ramamohanarao, and R. T. Snodgrass, editors, *VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy*, pages 591–600. Morgan Kaufmann, 2001.
- [3] S. Boag, D. Chamberlin, M. F. Fernandez, D. Florescu, J. Robie, J. Siméon, and M. Stefanescu. XQuery 1.0: An XML Query Language. Technical report, World Wide Web Consortium, April 2002. W3C Working Draft.
- [4] J. Freire, J. R. Haritsa, M. Ramanath, P. Roy, and J. Siméon. StatiX: Making xml count. In *SIGMOD 2002, Proceedings ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin, June 3-6, 2002, USA*. ACM Press, 2002.
- [5] C. Sartiani. A Framework for Estimating XML Query Cardinality - Moderately Extended Version, 2003. Available at <http://www.di.unipi.it/~sartiani/papers/wedb03ext.pdf>.
- [6] Y. Wu, J. M. Patel, and H. V. Jagadish. Estimating answer sizes for xml queries. In C. S. Jensen, K. G. Jeffery, J. Pokorný, S. Saltenis, E. Bertino, K. Böhm, and M. Jarke, editors, *Proceedings of the 8th International Conference on Extending Database Technology, Prague, Czech Republic, March 25-27, 2002*, volume 2287 of *Lecture Notes in Computer Science*, pages 590–608. Springer, 2002.

Path-expression Queries over Multiversion XML Documents

Zografoula Vagena
Computer Science and Engineering
University of California, Riverside
foula@cs.ucr.edu

Vassilis J. Tsotras
Computer Science and Engineering
University of California, Riverside
tsotras@cs.ucr.edu

ABSTRACT

In this paper we address the problem of evaluating path expression queries on multiversion XML documents. Such queries are typically implemented on static (i.e., non-versioned) documents as path joins, using numbering schemes that maintain the structural relationships among the document elements. We extend previously proposed pattern matching techniques so as to support versions. We first present an easily updatable numbering scheme that efficiently captures structural relationships among the elements of the dynamically evolving document. We then propose a variation of Pathstack, an optimal pattern matching algorithm, that addresses the characteristics of our environment. Finally, through a thorough experimental evaluation we investigate two storage techniques in terms of space utilization and query efficiency.

Categories and Subject Descriptors

H.3.m [Information Storage And Retrieval]: Miscellaneous

General Terms

Algorithms, Storage, Performance

Keywords

Path Expressions, Versioning, XML

1. INTRODUCTION

XML is gaining considerable attention as a standard for authoring, storing, exchanging and presenting documents [25]. In many applications, a document, as any other data source, may be subject to updates during its lifetime [20]. Such updates typically create a sequence of document versions making it possible for the users to query not only the most recent version but also previous ones. Moreover, users may be interested in querying the document evolution (i.e., *deltas*), too [19, 7, 6, 5, 4]. Tools and techniques that facilitate such *version management*, are very important. Multiversion support for XML documents is needed in several applications, like software configuration, cooperative authoring, web document warehouses etc.

An XML warehouse is usually modeled as a forest of rooted, ordered, labeled trees, where each tree represents a document. Each node in the tree corresponds to an element or a value, and the edges represent immediate element-sub element or element-value relationships.

Given the tree structure of XML documents, path expression queries represent the core of XML query processing. Queries (e.g. [23, 24]) in XML languages typically specify patterns of selection predicates on multiple elements that have some specified tree structure relationships. Path expressions are building components for such queries. A path expression defines a series of tree node labels, every two of which are related with an ancestor-descendant or parent-child relationship. Various works have recently addressed efficient implementation of such queries on static documents that reside in main memory [13, 14, 15] or in secondary memory [1, 3, 9, 18, 16, 21].

Our work is more related to the second group of techniques, as it deals with archives of documents, that in general are large and disk resident. One common assumption for the disk-based approaches is that a (range-based) numbering scheme [18, 1, 12] is embedded on the document tree in order to capture the above structural relationships among the document nodes. With this numbering scheme, path-expression queries are transformed to path joins [1, 3]. Holistic Joins ([3]) represent the state of the art in optimally processing path expression queries. Such joins utilize the document numbering scheme [11] to identify in constant time, the relative position of any two nodes.

Nevertheless, existing range-based numbering schemes are not easily updatable, a crucial issue when the document evolves (as is the case with versioning). While updatable prefix-based numbering schemes have been also proposed [10], they are not applicable to holistic joins. Moreover, prefix-based schemes may use variable length encodings and typically need more space than range-based numbering.

Previous work on version management of semi-structured documents [6, 5] and XML in particular [7, 8, 4, 19, 22], has considered various aspects of the storage and manipulation of multiversion documents. [7] proposed techniques that represent the document evolution as edit scripts or object references with the focus on fast reconstruction of any particular document version. [19] considers queries on the XML document's evolution changes, while [4] concentrates on the compact storage of a XML archival system, while its version querying assumes the existence of key identifiers. Finally, [8] addresses simple path expression queries (i.e., the special

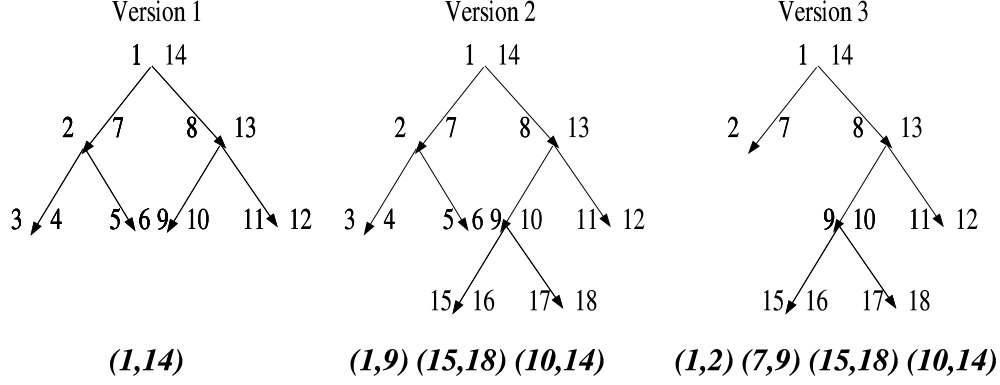


Figure 1: A Document Evolution.

case where an element cannot appear in the subtree of an element with the same type) in a multiversion XML document as combinations of partial version retrieval queries.

Nevertheless, none of these approaches is optimized for general path expression queries in a multiversion XML environment. This is a novel problem since it requires easily updatable numbering schemes to capture the structural characteristics of an evolving document. Moreover, our solutions will offer the ability to query any document version at different granularities. More specifically, we devise algorithms to efficiently answer queries that specify patterns of selection predicates on multiple XML nodes that have some specified tree structure relationships, over particular document versions.

The rest of the paper is organized as follows. Section 2 presents our techniques while section 3 contains the experimental evaluation. Finally, section 4 concludes the paper.

2. STRUCTURAL JOINS ON DOCUMENT VERSIONS

A straightforward way to use holistic joins on versioned documents is to renumber the whole document at every update. We first describe a document representation for which no document renumbering is needed in the presence of updates (section 2.1). This enables the utilization of holistic joins (with minor changes). The updated algorithm is presented in section 2.2. Moreover, we describe a physical organization of document versions that better preserves the logical order of the document elements (section 2.3).

2.1 Using Regions to Represent XML documents

Range-based numbering schemes map ancestor-descendant relationships to range inclusions. Every node in the document is assigned a range $(left, right)$, where $left$ is the number assigned to the node in a preorder traversal of the XML tree and $right$ is the corresponding number from a postorder traversal [18]. Clearly, $right > left$. A node v is descendant of node u if and only if $range(v)$ is included in $range(u)$. These schemes face the problem

of range shortage when updates happen in the document. For example, if a new node e is added under an existing node r , the range assigned to it, should be a proper subset of $range(r)$. Sooner or later the need to renumber the whole document will appear.

To mitigate the problem we propose representing a document with different granularities. We cluster the tree nodes into regions defined by the updates occurring in the document. Originally, each region is a subtree that is added or deleted from some part of the document. Updates may split that region into subregions. The structural relationships of the nodes within the *same* region are identified using ranges in the same way as before. Elements that belong to *different* regions are identified by the structural relationships of the regions that contain them. Certain nodes play eminent role in the identification of the inter-region relationships. These are the boundary nodes of each region. Moreover, for each document version, we maintain a *document map* which keeps the relative positions of the regions in that version.

We note that our mapping is reminiscent of the XID-map [19] used to create persistent object identifiers for each document node. The application of the XID-map is however different as it is used to identify deltas between versions. Moreover, it provides a mapping between an XML node and not a range but a single integer.

2.1.1 Document Evolution

For simplicity consider a single evolving document (the extension to multiple documents is trivial). Originally the whole document defines a single region. Since no splits have yet occurred, the range of this region is the range of the tree root. The document map is initialized with a single range, the range of this region. An example appears in Version 1 of Figure 1. As the document evolves and splits occur, more regions will be created. At each version, the document map maintains all the ranges associated and the regions present in this version.

Let's assume that in Version 2, a subtree is added on the XML tree. This results to the creation of a new region. This region is assigned a new range $(left, right)$ where $left$ is the smaller number that is larger than any of the numbers, in the document map of the

Algorithm Version PathStack

```
01 while not_end_of_input
02   find next element e
03   find range where element belongs
04   for every stack
05     if top_stack is not ancestor of e
06       pop top_stack
07   move e to its stack
08   if is_leaf(e)
09     showSolutions
10   pop(e)
```

Figure 2: The Version_PathStack (VP) Algorithm.

previous version (Version 1). Nodes in this new subtree are numbered in the preorder/postorder fashion, starting with *left*. Assume that the new subtree is added under node *u* with $range(u) = (u_left, u_right)$. Moreover, let *R* denote the region that contains *u* and let its range be (R_left, R_right) . Region *R* is split into two subregions with ranges (R_left, u_left) and (u_right, R_right) . The document map for Version 2 will contain the ranges of the two subregions with the range of the new subtree places between them (Figure 1).

The deletion of a subtree removes its range from the document map. For example, the deletion in Version 3, of the subtree under node with range (2,7), creates a document map that contains ranges (1,2), (7,9), (13,16) and (10,12). Moreover, a deletion may lead to range merging (if two subsequent ranges become consecutive, for example, if Version 3 had instead deleted (13,16)).

Since the document map maintains a compact representation of the XML document at each version, we expect that its size will be small and easily kept in main-memory. If its size becomes large it can easily migrate to the disk, without loss in the algorithm performance, as will be shown in section 2.2.1.

We proceed with establishing some notation. The node under which a new subtree is added is called the *root* node of the region corresponding to that subtree. Moreover, when a region splits into two subregions, it is called the *parent* of both. We define the ancestor-descendant relationships between regions recursively. It should be noted that the actual numbering scheme used for the range assignments need not be the preorder/postorder. Other schemes that leave spaces between ranges (for example the scheme in [18, 7]) can also be used.

2.2 Path Queries over a particular version

Using the numbering scheme described above one can perform structural joins over a particular version by extending pattern matching algorithms (like *PathStack*, *TwigStack* [3]). We begin with a brief introduction of *PathStack*, an optimal structural join algorithm, and extend it to adapt the document map.

2.2.1 PathStack

A path-expression query is defined as a sequence of element types. *PathStack* assumes that all elements of a given type exist in *element lists*.

Nodes in an *element list* are enhanced with the appropriate numbering range (*left*, *right*). Given a path query, say *a/b/c*, associated with every element type participating in it (in this case *a*, *b*, *c*, is a *stack*. As the algorithm proceeds, elements are pushed and popped from their corresponding stacks. Moreover, each element in the stack associated with a given type, points to an element in the stack of the parent type as the latter is defined by the query. For example, elements in the stack of type *c*, point to elements in the stack for type *b*.

There are three computation invariants in the algorithm. (a) Moving from the bottom to the top of a given stack, the elements correspond to document nodes that belong to the same path in the XML tree. (b) Each element in a given stack corresponds to an XML tree node that is an ancestor of all elements above it in the stack. (c) The set of elements in all stacks contains an encoding of total and partial query answers (for details we refer to [3]).

At each step, the *PathStack* algorithm traverses each element list sequentially, picks the next node (as defined by the inorder traversal of the document) and removes from the various stacks elements whose ranges are before the range of this node. The removed elements are guaranteed not to participate in any of the subsequent joins. The algorithm guarantees that the stacks contain elements whose ranges are either before or include the range of the current node, but never ranges located after this range.

2.2.2 The Version_PathStack

A path query in a multiversion environment is described by the path and the version of interest. For example, “find all *figure* elements that are under *chapter* elements in Version 20 of the document”. In Figure 2 we provide *Version_PathStack* (VP) which operates on multiversion documents. The input of the algorithm contains: (i) item the element lists of the query element types, and, (ii) the document map for the version defined in the query.

Each list is constructed as follows: for the first version of the document the list contains all the elements of the same type, enhanced with the numbering scheme and sorted on the *left* position of their range. As new subtrees are added, their elements are numbered and are appended in their corresponding lists. Since new regions receive larger numbers, the element lists remain sorted. Subtree deletions have no effect over the lists. This is because, each list effectively keeps all elements of this type from every version. (Equivalently, since we need to maintain all versions of the document, deletions are not physical, but logical through the use of the document map).

To be able to access different parts of the lists on demand, we maintain for each list an index that supports range queries (e.g. a B+-tree). This index allows access to document regions in the order implied by the document map. Specifically, we use the index to find the element in a list with the smallest *left* position in a given range.

At each step the VP algorithm needs to pick the appropriate next element from every input element list. For a given element list it first finds the next region from the document map for this version, and within this region, it locates the element with the smallest *left* position. To find the next region in this version it accesses the document map sequentially and for each range it probes the B+-tree to find if there is an element in the list with *left* position as above.

After finding the next element all stacks are cleared from elements that are not going to participate in subsequent joins. In this step the VP algorithm has to take into account the relative positions of the regions as exemplified by the document map. Figure 3 shows all possible relative positions between the current element X and the top of a stack. Specifically, the top element in a stack can:

1. be an ancestor of the element X ,
2. belong to the same region as X (region A) and its range is before the range of X ,
3. belong to the region that (possibly recursively) caused the split of X 's region (in the example, this is region B),
4. belong to a region with the same parent as X 's region and exist at the left of X (like region C), and,
5. belong to a region that exists at the left of an ancestor of X 's region (like regions E and D).

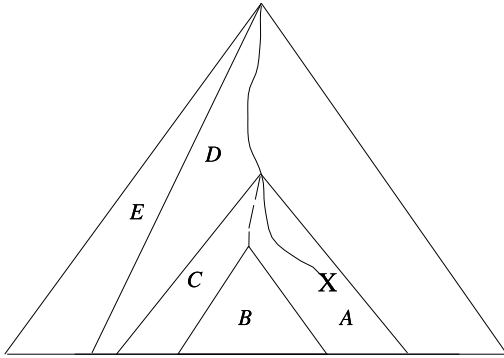


Figure 3: Relative positions between current and stack elements.

For any stack, the current top element has to be removed in all cases except case 1. Instead of checking whether a top stack element satisfies cases 2-5, it is easier to simply check whether it is an ancestor of element X (case 1). From element X we can easily find (through the document map) the root r of its region. Then we simply check whether the top stack element is an ancestor of r (and thus an ancestor of X , too). If both the top stack element and X happen to be in the same region, the ancestor test is performed by checking range containment.

2.3 Making the element lists persistent

Each element list is “append only”; new elements are added at the end of the list. This may lead to poor physical clustering of the elements and as a result poor retrieval performance. Precisely, one might need to get the same page from disk over and over, depending on which parts of the list are stored there. It would be beneficial if the lists were created in a way that maintains the element document (logical) sequence at each version. To this end, we propose a solution that, for each version:

1. retrieves only list pages that contain enough elements that are valid for the version (these are called “useful” pages for this version), and,
2. the elements within the page follow the logical order of the element list.

This is achieved by keeping the list of useful pages in logical order for each version. Moreover, we employ split and merge techniques similar to the ones used in the multiversion B-Tree [2, 17]. Specifically, after we find where in the list we will insert the new element, we check whether there is any space left in the current page. If so, we insert the element. Otherwise we copy only the elements valid for the current version in a new page. The old page is replaced with the new one in the list of useful pages for the version. If all the elements in the page are valid for this version, we split the page into two new ones that replace the former for the current version. While elements are removed the page can become underutilized. In this case we copy valid records of the page into a new one, making the old one non-useful.

The changes needed in the *Version PathStack* so that it works with the new storage scheme are minimal. Specifically, while accessing each list in its physical order to find the new current element for the algorithm, we need to keep track which document region the list pointer accesses at each time. To do that we can sequentially search the document map, or in the case where the former is too large to fit in main memory we can employ a content index (like B+-Tree) to speedup the search.

3. PERFORMANCE ANALYSIS

We compare the performance of three algorithms, namely the original *PathStack* algorithm, the VP that indexes the element lists with a B+-tree (we denote it as *VP-B*) and the VP with persistent element lists (*VP-PL*). We use *PathStack* as a measure of the minimal query time required for the structural join [3]. This however assumes that a snapshot of each element list is maintained for each version. This is impractical since it results in very high space; moreover, it requires renumbering the document for each update.

We have generated synthetic data sets (XML document) from which we generated 10 more versions. Our XML document conforms to the very simple DTD where there is a root element under which there can exist multiple a elements, under which may exist multiple b elements, under which may exist multiple c elements. We used the expression $a//b//c$ as our path query.

Each version differs from the previous about 10%. Half of the changes are subtree additions, and the other half subtree deletions. The changes are uniformly distributed among data pages. The page size is set to 8K and the generated element lists are approximately 11M. The buffer pool size was set to 100 blocks (i.e. approximately 2% of the input size). To generate the element lists from the document, we used an event based SAX parser. For *PathStack* we parsed the whole document for each new version. For the case of persistent lists we set the usefulness threshold to 50%, i.e. whenever 50% of the records in the page are cease to be valid for the current version then all the valid records are copied to a new page.

3.1 Storage Utilization

Figure 4.a shows the space occupied by the input (element lists) along the versions. The *PathStack* is not shown in the figure as

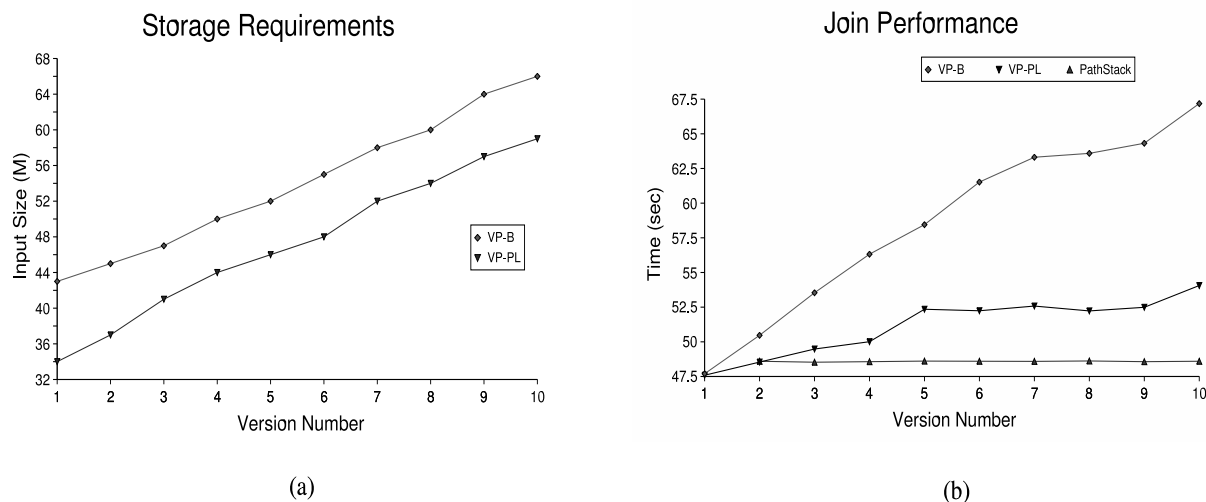


Figure 4: Performance Results.

the space requirements of its implementation quickly exceed the graph (it took only two versions to reach 68M). Among the two VP approaches, VP-PL is more space efficient than VP-B. This is because VP-B incurs the cost of storing the index pages, along with the added nodes while in VP-PL the storage is proportional to the added nodes. We expect the difference in storage requirements between the two schemes to decrease as the number of deletions increases, since this will cause more page invalidations for VP-PL and thus more replication.

3.2 Join Performance

Figure 4.b shows the join (query) performance of all three algorithms. The performance of the *PathStack* remains the same along the different versions, since the version changes in our experiments do not affect the input size and the join selectivity. The performance of VP-PL is slightly larger than the *PathStack*. This is attributed to the fact that the size of each element list per version is slightly larger than the snapshot of this list for that version (this is because page invalidation creates copying).

The VP-B approach performs worse than the other algorithms. This is because VP-B has to access parts of the input lists in a different order than their physical storage (something which is exacerbated with the persistent lists where the elements within the page have the same physical sequence as the algorithm is going to access them). Moreover, VP-B has to incur the overhead of the index traversal. As a conclusion, the VP-PL algorithm provides a robust solution for path-expression queries in a multiversion document environment. Its space overhead is limited (typically linear to the size of updates) while its query performance behaves closely to the optimal.

4. CONCLUSIONS

We examined the problem of answering path queries over versions of an XML archive. We first proposed a representation of the document that reveals structural relationships between document nodes and adapts gracefully when updates occur. We then extended previous optimal path-expression query algorithms so as to answer the

same queries on different document versions (the SP algorithm). Finally, we considered two alternative storage implementations for the element lists (SP-PL and SP-B) and presented an experimental comparison revealing the advantages of the SP-PL approach (limited space overhead, fast query time). It should be noted that our methodology and range numbering can be further adapted to more complex pattern matching XML queries.

Acknowledgements: This work was partially supported by NSF grants IIS-9907477, IIS-0220148 and UCdimi01-10122.

5. REFERENCES

- [1] S. Al-Khalifa, H. V. Jagadish, N. Koudas, J. M. Patel, D. Srivastava and Y. Wu, "Structural Joins: A Primitive for Efficient XML Query Pattern Matching", *Proc. of 18th International Conference on Data Engineering (ICDE)*, San Jose, CA, 2002.
- [2] B. Becker, S. Gschwind, T. Ohler, B. Seeger, P. Wildmayer "An Asymptotically Optimal Multiversion B-Tree", *VLDB Journal*, Vol. 5, No. 4, 1996, pp 264-275.
- [3] N. Bruno, N. Koudas, D. Srivastava, "Holistic Twig Joins: Optimal XML Pattern Matching", *Proc. of ACM SIGMOD Conference*, Madison, WI, 2002.
- [4] P. Buneman, S. Khanna, K. Tajima and W.C. Tan, "Archiving Scientific Data", *Proc. of ACM SIGMOD Conference*, Madison, WI, 2002.
- [5] S. Chawathe and H. Garcia-Molina, "Representing and Querying changes in semistructured data", *Proc. of 14th International Conference on Data Engineering (ICDE)*, Orlando, FL, 1998.
- [6] S. Chawathe, S. Abiteboul and J. Widom, "Managing Historical Semistructured Data", *TAPoS*, 5(3):143-162, 1999.
- [7] S.-Y. Chien, V.J. Tsotras, C. Zaniolo, "Efficient Management of Multiversion Documents by Object Referencing", *Proc. of 27th International Conference on Very Large Data Bases (VLDB)*, Roma, Italy, 2001.

- [8] S.-Y. Chien, V.J. Tsotras, C. Zaniolo and D. Zhang, "Efficient Complex Query Support for Multiversion XML Documents", *In Proc. of 8th Int. Conference on Extending Database Technology (EDBT 2002)*, Prague, Czech Republic, pp. 161-178, 2002.
- [9] S.-Y. Chien, Z. Vagena, D. Zhang, V. J. Tsotras, and C. Zaniolo, "Efficient Structural Joins on Indexed XML Documents", *Proc. of 28th International Conference on Very Large Data Bases (VLDB)*, pp. 263-274, Hong Kong, China, 2002.
- [10] E. Cohen, H. Kaplan and T. Milo, "Labeling Dynamic XML Trees", *Proc. of the 21st ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, Madison, WI, 2002.
- [11] M. P. Consens and T. Milo, "Optimizing Queries on files", *Proc. of SIGMOD Conference*, Minneapolis, MN, 1994.
- [12] T. Fiebig and G. Moerkotte, "Evaluating Queries on Structure with eXtended Access Support Relations", *Proc. of the 5th International Workshop on the Web and Databases (WebDB)*, Dallas, TX, 2000.
- [13] G. Gottlob, C. Koch and R. Pichler, "Efficient Algorithms for Processing XPath Queries", *Proc. of 28th International Conference on Very Large Data Bases (VLDB)*, Hong Kong, China, 2002.
- [14] G. Gottlob, C. Koch and R. Pichler, "XPath Processing in a Nutshell", *ACM SIGMOD Record* 32(1): 12-19, 2003
- [15] G. Gottlob, C. Koch, and R. Pichler, "XPath Query Evaluation: Improving Time and Space Efficiency", *Proc. of 19th International Conference on Data Engineering (ICDE)*, Bangalore, India, 2003.
- [16] H. Jiang , H. Lu, W. Wang, and B. C. Ooi, "XR-Tree: Indexing XML Data for Efficient Structural Join", *Proc. of 19th International Conference on Data Engineering (ICDE)*, Bangalore, India, 2003.
- [17] A. Kumar, V. J. Tsotras, C. Faloutsos, "Access Methods for Bi-Temporal Databases", *IEEE Transactions on Knowledge and Data Engineering*, Vol. 10, No. 1, 1998, pp 1-20.
- [18] Q. Li and B. Moon, "Indexing and Querying XML Data for Regular Path Expressions", *Proc. of 27th International Conference on Very Large Data Bases (VLDB)*, Roma, Italy, 2001.
- [19] A. Marian, S. Abiteboul, G. Cobena, L. Mignet, "Change-Centric Management of Versions in an XML Warehouse", *Proc. of 27th International Conference on Very Large Data Bases (VLDB)*, Roma, Italy, 2001.
- [20] I. Tatarinov, Z.G. Ives, et. al., "Updating XML", *Proc. of ACM SIGMOD Conference*, pp 413-424, Santa Barbara, CA 2001.
- [21] W. Wei, J. Haifeng, H. Lu and J-X. Yu, "PBTree Coding and Efficient Processing of Containment Join" *Proc. of 19th International Conference on Data Engineering (ICDE)*, Bangalore, India, 2003.
- [22] R.K. Wong and N. Lam, "Managing and Querying Multi-Version XML data with Update Logging" *Proc. of DocEng*, 2002.
- [23] World Wide Web Consortium, "XML Path Language (XPath)", Version 1.0. Nov. 16, 1999. See <http://www.w3.org/TR/xpath.html>
- [24] World Wide Web Consortium, "XQuery 1.0: An XML Query Language", W3C Working Draft Jun 7, 2001 (work in progress). <http://www.w3.org/TR/xquery/>
- [25] WWW Distributed Authoring and Versioning (webdav). See <http://www.ietf.org/html.charters/webdav-charter.html>.

TypEx: A Type Based Approach to XML Stream Querying

George Russell
Department of Computer and
Information Science
University of Strathclyde
Glasgow, U.K.

george@cis.strath.ac.uk

Mathias Neumüller
Department of Computer and
Information Science
University of Strathclyde
Glasgow, U.K.

mathias@cis.strath.ac.uk

Richard Connor
Department of Computer and
Information Science
University of Strathclyde
Glasgow, U.K.

richard@cis.strath.ac.uk

ABSTRACT

We consider the topic of query evaluation over semistructured information streams, and XML data streams in particular. Streaming evaluation methods are necessarily event-driven, which is in tension with high-level query models; in general, the more expressive the query language, the harder it is to translate queries into an event-based implementation with finite resource bounds.

We consider an alternative model by introducing a two-phase evaluation strategy. A query Q is decomposed into an event driven primary filter query Q' , which incrementally gathers relevant data from the input stream, and another query Q'' which consumes this data as it becomes available. Evaluation of $Q(s)$ is then equivalent to $Q''(Q'(s))$. The importance of the separation is that it allows the first phase Q' to be expressed in a non-Turing complete algebra which may therefore be generally amenable to event-based interpretation. The second phase Q'' may be expressed in an arbitrary higher-order language, so long as its execution takes no longer than the extraction of the next input instance from the input stream.

In this paper a type algebra is used to express the first-phase query. This builds on previous work, which shows how traditional programming language types may be given a semantics within XML, therefore allowing their projection onto XML resources. A side-effect of this definition of projection is that instances of a type may be extracted in a form available for computation within a traditional domain. The use of type projection in this context also allows Q'' to be statically typed according to the type filter used for Q' , which itself may be deduced by inference over Q'' . A mechanism for translating a type into a network of event driven automata, which has the effect of gathering all data captured by that type from a semistructured input stream, is described. Although at an early stage of investigation, initial results suggest this approach provides a credible alter-

native to stream-based querying in at least some application domains.

Categories and Subject Descriptors

H.2.4 [Database Management]: Languages—*Query Languages*; D.3.3 [Programming Languages]: Language Constructs and Features—*Data Types and Structures*

Keywords

Type projection, stream processing, query typing, language integration, semistructured data

1. INTRODUCTION

The requirement for efficient, stream-based XML querying is by now established. There are several applications of such systems, notably in *selective dissemination of information* (SDI) and publish / subscribe systems [1] and for very large data environments and data integration [8]. Another motivation is the efficient transformation of streaming XML data, in applications such as XSLT processing and continuous data streams [9]. Such applications require processing to occur in real time as data becomes available, rather than after the end of the input stream is reached as in most XML query models.

SAX¹ provides a fully general XML stream-processing abstraction, giving the programmer an event-based interface based on callbacks. This interface provides full real-time streaming functionality; however the nature of the interface makes programming challenging for many applications, as conversion from the event stream to the logical structure of the underlying data is entirely the burden of the programmer.

Given this difficulty, a number of authors have investigated translation from a higher-level expressive form into an event-based model, expressed most commonly as a deterministic finite state automata (DFA) network [2, 7, 9, 11, 12]. Partial translations from both XPath and XQuery have been discussed, and while there are still open issues, significant progress has been made. One of the major outstanding issues however is the identification of the level of expression which can be sensibly handled in this model. Both XPath and XQuery contain expressions which can not usefully be translated into a DFA model without compromising computational thresholds underlying the purpose of the transla-

¹<http://www.saxproject.org>

tions; the same of course is true for any single query language with sufficient expressive power to handle any reasonable range of queries.

We propose a significant departure from this methodology which we show works well for certain classes of application. The query is coded as a function in a Turing-complete programming language, with the assumption that the execution of this function will occur within acceptable bounds if its input can be isolated and passed to it as a process separated from the parsing of the input stream. This function could represent the query in its entirety, but in this domain the whole query is more likely to be represented by the repeated application of the function to instances of its input as they are extracted from the XML stream.

The function is typed according to its input, and this type is used to generate a deterministic state automata based input filter which extracts corresponding values from the XML input stream. These values are then passed, as they occur, to the query. In this way fully general queries can proceed in parallel with the parsing of the input. This builds on our previous work [5, 14, 15, 6, 10], in which we show how traditional programming language types may be given a semantics within XML, therefore allowing their extraction from XML resources.

The method will only work well if the query function is short-lived, and its type represents a significantly small subset of the XML stream; however we believe there exist many instances of this pattern. The main properties of the method are as follows:

- only the type-based extraction requires to be translated into the event-based paradigm
- only the second part of the query requires to be expressed, as the filter can be automatically generated from it
- the two phases of the execution can proceed in parallel

Compared with single-phase deterministic automata translations from XPath or XQuery, the method seems likely to work relatively well for complex queries over core regular data within a loosely structured stream, while it is likely to work relatively badly for simple queries over data that is inherently unstructured. The tradeoffs with respect to simplicity of expression and efficiency are complex and require further investigation, but in this paper we have at least shown credibility in these domains with respect to some classes of application.

In more generality, not considered further in this paper, the same basic two-part query framework may be used entirely within the XML standards domain, by expressing the filter by a projection defined over XML Schema. Instances of XML would then be generated by the first phase, allowing the secondary part to be expressed in any XML query language. One particularly interesting aspect of this is that, if the entire data stream is known to be valid with respect to a given schema, then the soundness of the filter may be

statically assessed, and furthermore user-level tools for generating it from the XML Schema stream description could be envisaged.

2. A MOTIVATING EXAMPLE

For motivation, an example streamed application is coded in Java using various alternative implementation techniques, namely SAX, XPath, and TypEx, the system based on the approach described. SAX only creates temporary data structures to report parsing events which need to be transferred manually into the application specific data model for processing. The XPath code uses an XPath expression to define a superset of the required data, in conjunction with DOM code to perform the required query; the TypEx code uses a Java class definition to define a superset of the required data, in conjunction with a method of that class to query it. In the cases of XPath and TypEx the initial extraction may be performed incrementally and processed in parallel with the code that uses the extracted values, whereas in the SAX application the entire processing must be performed within the parsing callbacks.

The example is a news ticker application, which extracts news items from an arbitrary XML stream. Schema information of the expected stream is incomplete and restricted to the relevant data. The application programmer knows that there are *item* elements which contain at least two direct child elements named *title* and *description*. Both these items contain only textual content and can occur only once within any *item*. This data may be embedded at any point in the source and additional content may appear beneath *item* elements. The application is to display the content of these two fields as they are detected within an unbounded stream of XML.

2.1 Parsing Event Based Model

An example of an approach in which the query is directly contained in the application code is the event-based abstraction model used by SAX. Mappings between parsing events and the data model used are spread over various callback methods, making it hard to understand and thus maintain. The mixture of selection and computation also increases the coupling between user-specified computation and parsing process and is thus undesirable.

Listing 1: The SAX handler for the news ticker

```
public class NewsHandler extends DefaultHandler {
    private StringBuffer _title , _description ;
    private Stack elements = new Stack();
    public void characters(char[] ch,
        int start , int length) {
        if (elements.search("item") == 2) {
            if (elements.search("title") == 1)
                _title .append(ch, start , length);
            if (elements.search("description") == 1)
                _description .append(ch, start , length);
        }
    }
    public void startElement(String namespaceURI,
        String localName, String qName, Attributes atts) {
        elements.push(qName);
        if (qName.equals("item")) {
```

```

        _description = new StringBuffer();
        _title = new StringBuffer();
    }
20 }
    public void endElement(String namespaceURI,
        String localName, String qName) {
        elements.pop();
        if (qName.equals("item"))
25         System.out.println( _title + "\n" + _description);
    }
    public static void main(String[] args) {
        SAXParser parser =
            SAXParserFactory.newInstance().newSAXParser();
30         parser.parse(new URL(args[0]).openStream(),
            new NewsHandler());
    }
}

```

Listing 1 shows that the required state information is maintained by using a stack containing all opened tags (line 15). String buffers are created once an opening *item* tag is found. Upon occurrence of character data the top two elements of the stack are checked. If they fit the structural constraints, the content is appended to the relevant buffers. The content of these buffers is printed when the end of a news item is detected. Note that this implementation does not verify order, multiplicity or even existence of required fields, but just checks that identified fields suffice the structural constraints. Other constraints would need to be checked using either a more complicated handler or a partial schema validation process, which is currently not part of any of the relevant standards.

2.2 XPath/DOM Based Model

The combination of *XPath* [16] and *DOM* allows a different approach. Programs specify the desired set of nodes using a path expression, and operate over the results using tree traversal code. XPath expressions are navigation expressions over tree structured data which are sent to an execution engine in a similar fashion as embedded SQL statements. Selection is mechanically performed by the system and returns a collection of trees, requiring tree traversal code in the user specified computation. Typically a superset of the data required is returned because XPath returns the entire subtrees rooted at the selected nodes, regardless of the data requirements of the subsequent computation. However, as there is no strong coupling between the two phases, it cannot be guaranteed that the returned data actually satisfies the computational needs.

Listing 2: The same application using XPath

```

public class BBCNewsXPath {
    public static void main(String[] args){
        DocumentBuilderFactory factory =
            DocumentBuilderFactory.newInstance();
5         DocumentBuilder builder =
            factory.newDocumentBuilder();
        Node doc = builder.parse(
            new InputSource( new URL (
                args [0]). openStream()));
10         NodeList results =
            XPathAPI.selectNodeList(doc,

```

```

        "//item[description and child::text()]"
        + "[title and child::text()]"
        + "[count(title)=1]"
        + "[count(description)=1]");
15         for (int i=0; i<results.getLength(); i++) {
            Node result = results.item(i);
            Element aItem = (Element) result;
            NodeList titles = aItem
                .getElementsByTagName("title");
            String title =
                ((Element) titles.item(0))
                .getFirstChild().getNodeValue();
            NodeList descriptions = aItem
                .getElementsByTagName("description");
25             String desc =
                ((Element) descriptions.item(0))
                .getFirstChild().getNodeValue();
            System.out.println( title + "\n" + desc);
30         }
    }
}

```

The program shown in Listing 2 uses the single XPath statement stretching from lines 12–15 to declare the navigational steps required to select the relevant data. The XPath execution engine returns a **NodeList** containing the selected nodes (line 10). The loop starting in line 16 iterates through this list and extracts the data from the relevant text nodes using the DOM API, and in particular the method *getElementsByTagName* which selects nodes based on their name. Structural checks upon the input format have been added to the XPath query in Listing 2, which allows them to be omitted from the result processing code which otherwise would contain explicit structural checking. The XPath implementation used is not streaming, but this does not affect the purpose of the example.

2.3 Type Projection Based Model

The approach suggested by this paper has been implemented in the *TypEx* system. The extraction phase identifies data that may be relevant for the query by means of a filter type and binds this data to an instance of this type for use in the second phase. Since the computation is specified in terms of the filter type, the returned instances of this type will always contain a superset of the information required.

Listing 3: The filter class *item*

```

public class item {
    String title , description;
    public String toString() {
        return title+"\n"+description;
5    }
}

```

In our example application, the class used to store and print news items is also used as the filter type (Listing 3). This defines the data extraction in terms of the host programming language and acts as a data model for the following computation, ensuring a match between the two phases and a seamless integration with the language. The actual com-

Listing 4: The TypEx Newsticker

```

public class BBCNews implements Observer {
    Extractor stories;
    public BBCNews() {
        stories = new Extractor(item.class);
5       stories.addObserver(this);
    }
    public void parse(InputStream in) {
        stories.parse(in);
    }
10   public void update(Observable o, Object i) {
        System.out.println((item) i);
    }
    public static void main(String[] args){
        BBCNews p = new BBCNews();
15       p.parse(new URL(args[0]).openStream());
    }
}

```

putation, i.e. the printing of the content is also defined by this type. Listing 4 shows the usage of this filter. It creates an **Extractor** parameterised by the type (line 4), which extracts **item** objects during parsing and passes them to the observer for display.

3. TYPE BASED EXTRACTION

In this section we outline the translation of a core type language into a network of co-operating, deterministic automata. The resulting network is capable of extracting values of those types.

The type language includes both record and list constructors, but does not allow anonymous lists to occur as they have no semantic projection in XML. In addition, lists cannot occur at the top level, as this would be incongruous with the purpose of the mechanism, which is to release typed data as soon as possible. A grammar for the type language is given below.

```

typedef ::= label : type
type    ::= record | scalar
record  ::= {label1: field1, ..., labeln: fieldn}
field   ::= scalar | record | list [ type ]
scalar  ::= int | string

```

A type expression is translated into a set of named component types as shown by the following example:

$p: \{a: \text{int}, b: \text{list}[\{d: \text{int}\}], c: \{d: \text{int}\}\}$

is transformed into

```

p : T1
T1: {a: T2, b: list[T3], c: T3}
T2: int
T3: {d: T2}

```

Each type T_i in this representation is mapped to an automaton, whose job is to extract an instance of that type from the input stream and return it as a value. The topology

produced by this example is shown in Figure 1. The size of the automata network equals the size of the corresponding type graph plus one for the additional **sink** machine.

The **root** and **sink** machines always occur as single instances; the others are generated according to the individual type components. Each internal connection is a two-way link, passing input events in one direction and results in the other. Only one machine at a time actively processes events; a machine become active when it receives an *init()* event, and remains so until either a *return()* or *fail()* event is passed back to its initiator. Non-active machines pass events on to the currently active machine. Events are propagated synchronously in that they are not passed to the active machine until the previous event has been processed; this avoids synchronicity problems with the return events being passed back up the chain.

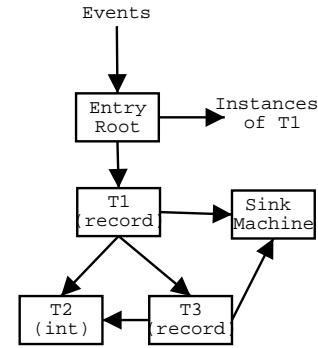


Figure 1: The example automata network

The following messages are defined: *open(l)*, *close(l)*, *text(v)*, *init(l)*, *return(v)*, and *fail()*. A parameter l stands for a label corresponding to an XML tag label, and a parameter v stands for a value. *open(l)*, *close(l)* and *text(v)* are events corresponding to a simplified input stream, and correspond to the textual form of the XML being processed. *init(l)*, as described, causes a machine to become active; its parameter is a label which, when subsequently received within a *close(l)* message, will cause it to pass control back to its initiator. This is either by means of a *return(v)* message, in the case it has been able to extract a value corresponding to its type from the input stream, and by means of a *fail()* message otherwise. There are four different classes of machines: **entry**, **sink**, **scalar** and **record**, as defined by the following behaviours:

- **Entry**: discard every event until an *open(l)* occurs, where l is the label corresponding to the name of the top-level *typedef*. At this point, *init(l)* is passed on to the machine corresponding to the type of the top-level *typedef*, which then becomes the active machine. Further events are passed to this machine until either a *return(v)* or *fail()* is received from it. On a *return(v)*, the parameter v is passed on to the network's receiver; on *fail()*, no further action occurs.
- **Sink**: the purpose of this machine is to discard an XML subtree, which cannot be of interest to the extraction. On receipt of *init(l)*, it becomes the active machine: its only required function is to discard events

until the corresponding *close(l)* event is received, ensuring that any contained subtrees using the same label *l* are also discarded. No value is returned.

- **Scalar:** on receipt of *init(l)*, a scalar machine requires the first event it receives to be a *text(v)* message, and its second to be a *close(l)* message. Depending on the particular scalar type, the string parameter *v* is examined to ensure it is structurally compatible, and if so is coerced and returned. Any other combination of events causes the machine to pass a *fail()* event back to its initiator.
- **Record:** a record machine starts with internal state variables corresponding to each of the field names occurring in its progenitor type description. For each one, the value is initialised to *undefined* if the field is a scalar or record type, and to an empty list if it is a list type. These variables are used to build up the state corresponding to the record value it attempts to extract. It may receive *text(v)*, *open(l)* and *close(l)* events from its initiator:

- *text(v)*: the event is discarded
 - *open(l)*: the behaviour depends on whether there is an internal state variable corresponding to *l*, and if so what its value is. If there is no internal variable, an *init(l)* message is sent to the **sink** machine. If there is a variable which is a list, an *init(l)* message is passed to the machine corresponding to the field type; if this returns with a value, it is appended to the list, otherwise no action is taken. If there is a variable whose value is defined, this results in termination with *fail()*. Finally, if there is a variable whose value is not yet defined, an *init(l)* message is passed to the machine corresponding to the field type; if this returns with a result, it is assigned to the internal variable, otherwise no action is taken.
- This behaviour corresponds to a structural subtyping approach to extracting the corresponding data values in the XML whilst ignoring order fields and any extra fields that are not pertinent to the query.
- *close(l)*: if all internal state fields are defined, a record value based on them is constructed and returned to the calling machine in a *return(v)* message; otherwise, a *fail()* message is passed back.

4. IMPLEMENTATION

The automata networks within *TypEx* form the middle layer of the system architecture. Below this layer an event handler translates parsing events generated by an underlying XML parser into automata events. Above the automata layer is the transformation layer, which generates networks from filter specifications and transforms extracted data graphs into instances of the specified type. The top layer is the programmer visible API, which allows the specification of an input source and associated filter types. It allows multiple listeners per filter and multiple filters per data stream, affording a degree of parallelism in the query process.

Each automaton is implemented as a Java class. Automata networks are generated using reflection to examine the field

types and names of filter types. Reflection is also used during data instantiation process.

5. EXPERIMENTAL RESULTS

To determine the viability of our approach, we have processed a more complex example query than the one discussed in Section 2. The data set under consideration has been generated using the XMark benchmark [13] scalable data generation tool and describes auction site details containing items for sale, persons (bidders and sellers) and some more information not relevant for our purposes. In particular, we have queried large XML files (up to 11GB in size) for people (with attributes such as name, address, etc.). We compare programs based on SAX, DOM XPath and TypEx both in terms of the complexity of the code and their runtime performance. The number of lines have been taken as a simple measure of the complexity of the code (Table 1).

System	Lines of Code		Comment
	Select	Extract	
SAX		150	Selection and extraction cannot be separated. Uses DOM extraction code.
DOM		43	
XPath	1	39	
TypEx	6	17	

Table 1: Length of Query

System	1 MB	10 MB	15 MB	30 MB	100 MB
DOM	1.73	147.0	326.0	1296	N/A
XPath	0.38	4.6	7.7	45	N/A
SAX	0.19	1.1	1.5	3	13
TypEx	0.43	3.6	5.6	10	21

Table 2: Length of Query Execution (s)

Table 1 supports the observation gained from the news ticker example and illustrates the conciseness possible using the TypEx approach. The SAX program, contains a mix of high-level application code and low-level parsing callbacks and results in an order of magnitude increase of code. DOM and XPath approaches lie between these two extremes and are almost identical because of their common tree traversal code.

The SAX query execution time scales linearly with the size of the input data and its memory usage depends only the level of element nesting. DOM is unsuitable for streaming due to its inherent whole document approach. As the input size increases, the time taken to complete the query increases more than linearly, while the memory usage increases linearly. With an 100 MB source file, the query fails to complete due to an *OutOfMemory* error using a heap size of 512 MB. We were unable to obtain a complete implementation of XPath to operate upon streaming data. The implementation we used for this experiment, Xalan/J, is backed by an incrementally built tree structure, and thus scales similarly to a DOM implementation in space. It does not however, provide results in an incremental fashion and thus does not allow a direct comparison with either SAX or TypEx. TypEx, while slower to execute than the equivalent SAX program also scales linearly with the size of the input data. The memory requirements are determined by the size of the extracted type, i.e. are independent of the size of the document. We have used it to successfully query inputs of up to 11 GB in size.

6. RELATED WORK

To the best of our knowledge, this is the first attempt of using types as filters in a two stage stream query process. The requirements for stream processing have been identified in [3] and elsewhere.

Much of the effort concentrates on implementing the XPath [16] language over streams. A number of independent efforts are proceeding, such as XMLTK [2], SPEX [11], and XSQ [12]. Green et al. [7] describe the implementation of an XPath subset using a lazily generated network of DFA and provide a comparison with other XPath implementations over streams. The work concentrates on the construction of a single automata network representing a large number of XPath queries which are executed simultaneously. Olteanu et al. [11] describe the translation of a subset of XPath expression into equivalent expression using only forward axis, which can then be efficiently processed by their SPEX XPath system using a network of event driven automata. Finally the XSQ system [12] is based upon push-down transducers with associated buffers and aims at complete XPath support.

A more recent approach by Ludäscher et al. [9] describes the implementation of the computationally complete query language XQuery over streaming data, using transducer networks derived from a query expression. Their XSM system optimises the derived network based on static analysis and available schema information. Optimisation strategies used may prove useful for our approach.

Type projection is introduced in [5] and formalised in [15]. Language integration with Java is described in [14] in more depth, while its use within query languages for XML is also outlined [10, 6].

7. CONCLUSIONS

We have outlined a novel mechanism which allows XML stream queries to be decomposed into an extraction and a computation phase over the extracted data based on a single type description. While the concept requires further investigation, it has a number of advantages for some classes of application. In particular, it may allow the programmer to use a higher-level, and therefore more succinct, query without gravely affecting properties of efficiency. Queries are formulated within the domain of the host language and can thus be statically typed.

The mechanisms have been implemented and first results show some promise. However the investigation is at an early stage and a great deal of work remains to be done.

8. ACKNOWLEDGEMENTS

We would like to thank Fabio Simeoni, David Lievens, Steve Neely and the anonymous referees for making useful suggestions on the presentation of this work. It has been financially supported by EPSRC (GR/M72265) and BBSRC (17/BIO12052.) George Russell and Mathias Neumüller are supported by PhD studentships funded by the University of Strathclyde.

9. REFERENCES

- [1] M. Altinel and M. J. Franklin. Efficient filtering of XML documents for selective dissemination of information. In A. E. Abbadi, et al., editors, *VLDB 2000*, pages 53–64, 2000. Morgan Kaufmann.
- [2] I. Avila-Campillo, T. J. Green, et al. XMLTK: An XML toolkit for scalable XML stream processing. In *PLAN-X: Programming Language Technologies for XML*, 2002.
- [3] B. Babcock, S. Babu, and other. Models and issues in data stream systems. In L. Popa, editor, *PODS 2002*, pages 1–16, Madison, Wisconsin, USA, 2002. ACM.
- [4] P. A. Bernstein, Y. E. Ioannidis, et al., editors. *Proceedings of the 28th International Conference on Very Large Databases*, Hong Kong, China, 2002. Morgan Kaufmann.
- [5] R. Connor, D. Lievens, et al. Extracting typed values from XML data. In *OOPSLA Workshop on Objects, XML and Databases*, 2001.
- [6] R. Connor, D. Lievens, et al. Projector – a partially typed language for querying XML. In *PLAN-X: Programming Language Technologies for XML*, 2002.
- [7] T. J. Green, G. Mikklau, et al. Processing XML streams with deterministic automata. In D. Calvanese et al., editors, *ICDT 2003*, volume 2572 of *LNCS*, pages 173–189. Springer, 2002.
- [8] T. Kiesling. Towards a streamed XPath evaluation. Diplomarbeit, Universität München, 2002.
- [9] B. Ludäscher, P. Mukhopadhyay, and Y. Papakonstantinou. A transducer-based XML query processor. In Bernstein et al. [4].
- [10] P. Manghi, F. Simeoni, et al. Hybrid applications over XML: Integrating the procedural and declarative approaches. In *Fourth International Workshop on Web Information and Data Management (WIDM'02)*, Virginia, USA, Nov 2002.
- [11] D. Olteanu, T. Kiesling, and F. Bry. An evaluation of regular path expressions with qualifiers against XML streams. In *ICDE 2003*, Mar 2003.
- [12] F. Peng and S. S. Chawathe. XPath queries on streaming data. In *SIGMOD 2003*, 2003.
- [13] A. R. Schmidt, F. Waas, et al. XMark: A benchmark for XML data management. In Bernstein et al. [4], pages 974 – 985.
- [14] F. Simeoni, D. Lievens, et al. Language bindings to XML. *IEEE Internet Computing*, 7(1), Jan/Feb 2003.
- [15] F. Simeoni, P. Manghi, et al. An approach to high-level language bindings to XML. *Information & Software Technology*, 44(4):217–228, 2002.
- [16] World Wide Web Consortium. *XML Path Language (XPath) Version 1.0*, W3C recommendation 16 November 1999 edition, 1999. <http://www.w3.org/TR/1999/REC-xpath-19991116>.

A Comparison of Peer-to-Peer Search Methods

Dimitrios Tsoumakos
Computer Science Department
University of Maryland
dtsouma@cs.umd.edu

Nick Roussopoulos
Computer Science Department
University of Maryland
nick@cs.umd.edu

ABSTRACT

Peer-to-Peer networks have become a major research topic over the last few years. Object location is a major part in the operation of these distributed systems. In this work, we present an overview of several search methods for *unstructured* peer-to-peer networks. Popular file-sharing applications, through which enormous amounts of data are daily exchanged, operate on such networks. We analyze the performance of the algorithms relative to their success rates, bandwidth consumption and adaptation to changing topologies. Simulation results are used to empirically evaluate their behavior in direct comparison.

1. INTRODUCTION

Peer-to-Peer (hence P2P) computing represents the notion of sharing resources available at the edges of the Internet. After its initial success, which resulted in the subsequent appearance of numerous P2P systems, it now emerges as the dominant model for the networks of the future. The P2P paradigm dictates a fully-distributed, cooperative network design, where nodes collectively form a system without any supervision. Its advantages (although application-dependent in many cases) include robustness in failures, extensive resource-sharing, self-organization, load balancing, data persistence, anonymity, etc.

Today, the most popular P2P applications operate on *unstructured* networks. In these networks, peers connect in an ad-hoc fashion, the location of the documents is not controlled by the system and no guarantees for the success of a search are offered to the users. Bandwidth consumption attributed to such applications amounts to a considerable fraction (up to 60%) of the total Internet traffic [1]. It becomes obvious that the nature of data discovery and retrieval is of great importance to the user and the broad Internet community.

In this work, we examine the problem of object discovery in *unstructured* P2P networks. Nodes make requests for ob-

jects they want to retrieve and cannot be found in their local repositories. The search process includes aspects such as the query-forwarding method, the set of nodes that receive query-related messages, the form of these messages, local processing, locally stored indices and their maintenance, etc.

Current search algorithms aim for bandwidth-efficient and adaptive object discovery for these networks. Search methods can be categorized as either *blind* or *informed*. In a *blind* search, nodes hold no information that relates to document locations, while in *informed* methods, there exists a centralized or distributed directory service that assists in the search for the requested objects.

In this paper, we describe current approaches from both categories and analyze their performance. We focus on the behavior of these algorithms for each of the following metrics: Efficiency in object discovery, bandwidth consumption and adaptation to rapidly changing topologies. The first metric measures search *accuracy* and the number of discovered objects per request. The latter is important for many applications, as it gives users a much broader choice for object retrieval. Minimizing message production always represents a high-priority goal for all distributed systems. Finally, it is important that any search algorithm adapts to dynamic environments, since in most P2P networks users frequently enter and leave the system, as well as update their collections.

To evaluate our analysis, we simulate many of the described methods and present a quantitative comparison of their performance. We also identify the conditions under which each method would be most or least effective.

In Section 2 we present related work. Section 3 categorizes and describes current search techniques, while in Section 4 we present the simulation results.

2. RELATED WORK

Peer-to-Peer networks have been studied a lot in the last few years. A large amount of information for P2P computing with taxonomies, definitions, current trends, applications and related companies can be obtained at [2, 3], as well as individual sources (e.g., [4, 5]). P2P computing is also described in [6], with basic terminology, taxonomies and description of some systems. A brief summarization of Gnutella [7] and Napster [8] search, together with ap-

proaches for *structured* networks are also included.

Gnutella and Napster are the focus of two measurement studies; [9] attempts a detailed characterization of the participating end-hosts, while [10] measures the locality of stored and transferred documents. In [11], a traffic measurement for three popular P2P networks is being conducted at the border routers of a large ISP. Extensive results for traffic attributed to HTTP, Akamai and P2P systems are also presented in [12].

Quantitative comparisons between the search methods in [13, 14] and the original Gnutella algorithm are presented in these two papers. Their main comparison metric is bandwidth consumption. The work in [15] presents a thorough comparison between the proposed algorithm and two blind search schemes ([14, 16]) on a variety of metrics.

Our work focuses exclusively on search in *unstructured* P2P networks, together with an experimental comparison of the methods under certain criteria.

3. P2P SEARCH ALGORITHMS

3.1 Our Framework

In our general framework, peers communicate either when they search for an object or when they share one. In this work, we examine proposed algorithms for the first part only.

Each peer retains a local collection of documents, while it makes requests for those it wishes to obtain. The documents are stored at various nodes across the network. Peers and documents (or objects) are assumed to have unique identifiers, with object IDs used to specify the query target. Search algorithms cannot in any way dictate object placement and replication in the system. They are also not allowed to alter the topology of the P2P overlay. Nodes that are directly linked in the overlay are *neighbors*. A node is always aware of the existence and identity of its neighbors. Nodes can also keep *soft state* (i.e., information that is erased after a short amount of time) for each query they process. Each search is assigned an identifier, which, together with the soft state, enables peers to make the distinction between new queries and duplicate ones received due to a cycle.

A search is *successful* if it discovers at least one replica of the requested object. The ratio of successful to total searches made is called the *success rate* (or *accuracy*). A search can result to multiple discoveries (or *hits*), which are replicas of the same object stored at distinct nodes. A global *TTL* parameter represents the maximum hop-distance a query can reach before it gets discarded.

3.2 Search Taxonomy

There are two possible strategies for search in an unstructured P2P network: Search in a *blind* fashion, trying to propagate the query to a sufficient number of nodes in order to satisfy the request; or utilize information about document locations and thus perform an *informed* search. The semantics of the used information range from simple forwarding hints to exact object locations. The placement of this information can also vary: In centralized approaches (e.g., [8]),

a central directory known to all peers exists. In distributed approaches ([17, 13], etc.), each individual peer holds a piece of the information.

One can also categorize search algorithms according to the model of P2P network they are designed to operate on. In *pure* P2P systems, all participating peers play both the roles of the client and the server. Other algorithms operate on *hybrid* P2P architectures, where certain nodes assume the role of a *super-peer* and the rest become *leaf-nodes*. Each super-peer acts as a proxy for all its neighboring leaves by indexing all their documents and servicing their requests.

3.3 Blind Search Methods

GNUTELLA [7]: The original Gnutella algorithm uses flooding (BFS traversal of the underlying graph) for object discovery and contacts all accessible nodes within the *TTL* value. Although it is simple and manages to discover the maximum number of objects in that region, the approach does not scale, producing huge overhead to large numbers of peers.

Modified-BFS [13]: This is a variation of the flooding scheme, with peers randomly choosing only a ratio of their neighbors to forward the query to. This algorithm certainly reduces the average message production compared to the previous method, but it still contacts a large number of peers.

Iterative Deepening: Two similar approaches that use consecutive BFS searches at increasing depths are described in [18, 14]. These algorithms achieve best results when the search termination condition relates to a user-defined number of hits and it is possible that a “small” flood will satisfy the query. In a different case, they produce even bigger loads than the standard flooding mechanism.

Random Walks [14]: In *Random Walks*, the requesting node sends out k query messages to an equal number of randomly chosen neighbors. Each of these messages follows its own path, having intermediate nodes forward it to a randomly chosen neighbor at each step. These queries are also known as *walkers*. A walker terminates either with a success or a failure. Failure can be determined by two different methods: The *TTL*-based method and the *checking* method, where walkers periodically contact the query source asking whether the termination conditions have been satisfied.

The algorithm’s most important advantage is the significant message reduction it achieves. It produces $k \times TTL$ messages in the worst case, a number which seldom depends on the underlying network. Simulation results in [14, 15] show that messages are reduced by more than an order of magnitude compared to the standard flooding scheme. It also achieves some kind of local “load balancing”, since no nodes are favored in the forwarding process over others.

The most serious disadvantage of this algorithm is its highly variable performance. Success rates and number of hits vary greatly depending on network topology and the random choices made. Finding an object depends slightly on its hop-distance. Another drawback of this method is its inability to adapt to different query loads. Queries for popular and unpopular objects are treated in the exact same manner. *Random Walkers* cannot *learn* anything from its previous successes or failures, displaying high variability in

all ranges of requests.

Recently, two new search protocols which operate on *hybrid* P2P networks made their appearance:

GUESS [16]: This algorithm builds upon the notion of *Ultra-peers* [19]. Each ultra-peer is connected to other ultra-peers and to a set of leaf-nodes (peers shielded from other nodes), acting as their proxy. A search is conducted by iteratively contacting different ultra-peers (not necessarily neighboring ones) and having them ask all their leaf-nodes, until a number of objects are found. The order with which ultra-peers are chosen is not specified.

Gnutella2 [20]: In Gnutella2, when a super-peer (or *hub*) receives a query from a leaf, it forwards it to its relevant leaves and also to its neighboring hubs. These hubs process the query locally and forward it to their relevant leaves. No other nodes are visited with this algorithm. Neighboring hubs regularly exchange local repository tables to filter out unnecessary traffic between them.

Although the details of these protocols are still formulating, we observe they rely on a dynamic hierarchical structure of the network. They present similar solutions for reducing the effects of flooding by utilizing the structure of hybrid networks. The number of leaf-nodes per super-peer must be kept high, even after node arrivals/departures. This is the most important condition in order to reduce message forwarding and increase the number of discovered objects.

3.4 Informed Search Methods

Intelligent-BFS [13]: This is an *informed* version of the *modified-BFS* algorithm. Nodes store query-neighborID tuples for recently answered requests from (or through) their neighbors in order to rank them. First, a peer identifies all queries similar to the current one, according to a query similarity metric; it then chooses to forward to a set number of its neighbors that have returned the most results for these queries. If a hit occurs, the query takes the reverse path to the requester and updates local indices.

This approach focuses more on object discovery than message reduction. At the cost of an increased message production compared to *modified-BFS* (because of the update process), the algorithm increases the number of hits. It achieves very high accuracy, enables knowledge sharing and induces no overhead during node arrivals/departures. On the other hand, its message production is very large and only increases with time as knowledge is spread over the nodes. It shows no easy adaptation to object deletions or peer departures. This happens because the algorithm does not utilize negative feedback from searches and forwarding is based on ranking. Finally, its accuracy depends highly on the assumption that nodes specialize in certain documents.

APS [15]: In *APS*, each node keeps a local index consisting of one entry for each object it has requested per neighbor. The value of this entry reflects the relative probability of this node's neighbor to be chosen as the next hop in a future request for the specific object. Searching is based on the deployment of k independent walkers and probabilistic forwarding. Each intermediate node forwards the query to one of its neighbors with probability given by its local index. Index values are updated using feedback from the walkers. If a walker succeeds (fails), the relative probabilities of the nodes on the walker's path are increased (decreased). The update

procedure takes the reverse path back to the requester and can take place either after a walker miss (*optimistic* update approach), or after a hit (*pessimistic* update approach).

APS exhibits many plausible characteristics as a result of its *learning* feature. Every node on the deployed walkers updates its indices according to search results, so peers eventually share, refine and adjust their search knowledge with time. Walkers are directed towards objects or redirected if a miss or an object deletion occurs. *APS* is also very bandwidth-efficient, achieving very similar levels with *Random Walks*. It induces zero overhead over the network at join/leave/update operations and displays a high degree of robustness in topology changes. The *s-APS* modification adaptively switches between the optimistic and pessimistic approaches and further reduces the amount of messages. Finally, [15] showed that the algorithm's majority of discovered objects are located close to the requesters. These advantages are mainly seen when many different peers contribute with big workloads. This is because *APS* gains from knowledge build-up and increased peer cooperation.

Local Indices [18]: Each node indexes the files stored at all nodes within a certain radius r and can answer queries on behalf of all of them. A search is performed in a BFS-like manner, but only nodes accessible from the requester at certain depths process the query. To minimize the overhead, the hop-distance between two consecutive depths must be $2r + 1$.

This approach resembles the two search schemes for hybrid networks. The method's accuracy and hits are very high, since each contacted node indexes many peers. On the other hand, message production is comparable to the flooding scheme, although the processing time is much smaller because not every node processes the query. The scheme also requires a flood with $TTL = r$ whenever a node joins/leaves the network or updates its local repository, so the overhead becomes even larger for dynamic environments.

Routing Indices (RI) [17]: Documents are assumed to fall into a number of thematic categories. Each node knows an approximate number of documents from every category that can be retrieved through each outgoing link (i.e., not only from that neighbor but from all nodes accessible from it). The query termination condition always relates to a minimum number of hits. The forwarding process is similar to DFS: A node that cannot satisfy the query stop condition with its local repository will forward it to the neighbor with the highest "goodness" value. Three different functions which rank the out-links according to the expected number of documents discovered through them are also defined. The algorithm backtracks if more results are needed.

This is another keyword-search approach which trades index maintenance overhead for increased accuracy. While a search is very bandwidth-efficient, RIs require flooding in order to be created and updated, so the method is not suitable for highly dynamic networks. Moreover, stored indices can be inaccurate due to thematic correlations, over-counts or under-counts in document partitioning and network cycles.

In [21], each node holds d bloom filters for each neighbor. A filter at depth i summarizes documents that can be found i hops away through that specific link. Nodes forward queries to the neighbor whose smaller depth bloom filter matches a

hashed representation of the object ID. After a certain number of steps, if the search is unsuccessful, it is handled by a deterministic algorithm instead of backtracking.

This method exhibits the same characteristics as the two previous ones. The scheme’s expectation is to find only one replica of the object with high probability. Index maintenance requires flooding messages initiated from nodes that arrive or update their collections.

Distributed Resource Location Protocol (DRLP) [22]: Nodes with no information about the location of a file forward the query to each of their neighbors with a certain probability. If an object is found, the query takes the reverse path to the requester, storing the document location at those nodes. In subsequent requests, nodes with indexed location information directly contact the specific node. If that node does not currently obtain the document, it just initiates a new search as described before.

This algorithm initially spends many messages to find the locations of an object. In subsequent requests, it might take only one message to discover it. Obviously, a small message production is achieved only with a large workload that enables the initial cost to be amortized over many searches. In rapidly changing networks, this approach fails and more nodes have to perform blind search. This also affects the number of hits: If many blind searches are made, then many results are found; if many direct queries take place, then only one replica is retrieved. So, this scheme is very dependent on network/application parameters.

4. SIMULATION RESULTS

In this section we present results for six of the described methods (*GUESS*, *Random Walks*, *Modified-BFS*, *Intelligent-BFS*, *s-APS*, *DRLP*). Three of the simulated methods are representative *blind* search schemes. The rest are *informed* methods that do not require user-initiated index updates.

Due to space limitations, we will briefly summarize our simulation model here. More details can be found in [15]. We use a *random* graph of 10000 nodes and an average degree of 10 generated by GT-ITM [23] to simulate our P2P overlay structure. We assume a *pure* P2P model, where all peers equally make and forward requests. Results comparing *APS* and *GUESS* in *hybrid* topologies can be found in [15].

Queries are made for 100 objects, with object 1 being the most popular and object 100 the least. Qualitatively similar results are produced when using a larger number of objects in the simulations. Objects are stored over the network according to the *replication distribution*, while nodes make requests according to the *query distribution* (e.g., popular objects get many more requests than unpopular ones). A zip-fian distribution with parameter $\alpha = 0.82$ is used to model both and achieve workloads similar to the observations in [10]: The highest-ranked 10% of objects amount to about 30% of the total number of stored objects and receive about 30% of all requests. Requester nodes are randomly chosen and represent about 20% of the total number of nodes. Each requester makes about 1500 queries. The *TTL* parameter was set to 5 for all algorithms, since larger values produced very similar results.

To simulate dynamic network behavior, we insert “on-line”

nodes and remove active ones with varying frequency. We always keep approximately 80% of the network nodes active, while arriving nodes start functioning without any prior knowledge. The objects are also re-distributed (less frequently though) to model file insertions and deletions. Object re-location always follows the initial distribution parameters.

The *Intelligent-BFS* method was modified to allow for object-ID requests. Index values at peers now represent the number of replies for an object through each neighbor. Nodes simply choose the 5 highest ranked neighbors in query forwarding. For *Modified-BFS* and *DRLP*’s blind search, nodes randomly choose half of their neighbors (5 on average) to forward a query to. In our *GUESS* implementation, peers deploy k random walkers with $TTL = 4$. The last nodes on the paths of these walkers forward the query to all their neighbors. In our simulations, $k = 12$ for *Random Walks*, *APS* and *GUESS*.

We simulate the algorithms at three different environments: In the static case there are no dynamic operations. In the less dynamic setting, the topology changes on average 240 times and objects are relocated 120 times at each run. In the highly dynamic setting, the topology changes about 1200 times and objects are relocated about 500 times during each run.

Figures 1, 2 and 3 present the results for the six methods. As expected, both *Modified* and *Intelligent-BFS* show extremely high accuracy and return many hits. Although their message production is an order of magnitude less than that of the original Gnutella scheme, they still produce almost 2 orders of magnitude more than the other four schemes. The informed method produces 3 times more messages, but also manages to find 3 times more objects. Both algorithms are not affected by the changing environment and achieve similar results in all settings. For environments similar to our setup, the modified method will be preferred to the intelligent one, since its performance is equally high and it is much simpler. We expect that the informed method will perform better in specialized environments (like the one described in [13]), mainly in the number of hits, which is one of the algorithm’s goals.

Random Walks displays low accuracy and finds less than one object per query on average. Its bandwidth consumption is quite low (between 33 and 40 messages) and its overall performance is hardly affected by the dynamic operations. *GUESS* behaves similarly, with the exception of being steadily over 60% accurate and discovering about 2 objects per query. On the other hand, it produces twice the number of messages of *Random Walks*. In general, these algorithms appear very robust to increased network variability. This is reasonable, as walkers are randomly directed with no regard to topology or previous results.

s-APS achieves a success rate of over 90% in the static run, a number that drops by 6% and 15% in the following settings. The metric that is reasonably affected is the number of discovered objects, which are almost cut to a third (from 6 to 2.2). This happens because it takes some time for the learning feature to adapt to the new topology and

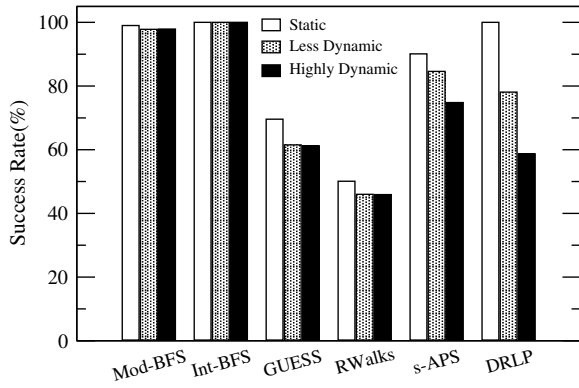


Figure 1: Success rate

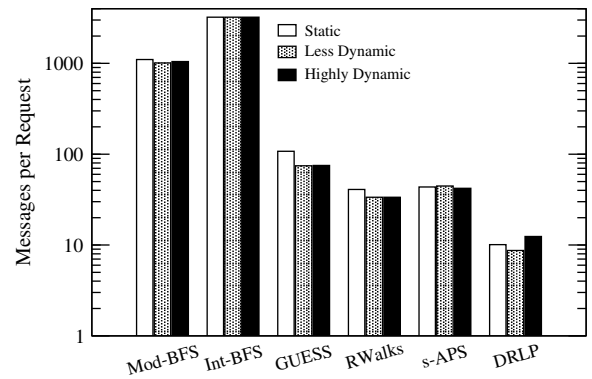


Figure 2: Messages per query

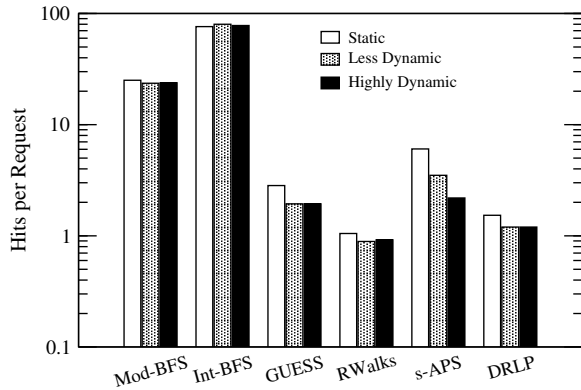


Figure 3: Hits per query

paths to discovered objects frequently “disappear”. On the other hand, it manages to keep its messages at the same levels, producing about 42 per search. *s-APS* exhibits a very good overall performance compared to the four non-BFS related schemes, being much more bandwidth-efficient than the BFS-related techniques.

The *DRLP* algorithm exhibits some interesting characteristics. First, its message production is very low (from 9 to about 12 messages per request), with only a small increase as the network becomes more dynamic. Our simulations count the direct contact of a node in *DRLP* as one message, although a link between them might not exist in the overlay. Dynamic behavior causes the stored addresses to become more frequently “stale”, thus the initial flooding is performed more often. This is the reason for the decrease in its accuracy from 100% in the static case to 80% and 60% respectively. *DRLP* produces the same amount of messages for its initial search with *Modified-BFS*, so it needs many successful requests to amortize this initial cost. The number of objects it discovers is small, ranging from 1.5 to 1.2. If *DRLP* is forced to use flooding many times, then the number of hits increases. If it is successful and produces few messages, then it only finds one replica per request. Despite this fact, we notice that it proves very bandwidth-efficient, while one would expect an increase to justify more flooding requests. This is due to the fact that, with many nodes making requests, most of them obtain a pointer for every

object after a while. So, even if some node initiates a flood, most of its neighbors will only forward to one node. This scheme seems ideal for relatively static environments and large workloads, with the exception that the number of hits will be very close to one. Another observation we made is that *DRLP* is affected more by object relocation than by node departures.

Figure 4 shows how object popularity affects the accuracy of the six schemes in the highly dynamic environment. The results are similar for the other two settings. Popular objects are stored in more nodes and receive more requests. Popularity decreases as we move to the right along the x-axis. The first data point represents the accuracy of the methods for objects 1-10, the second for objects 11-20, etc. The two BFS methods exhibit high accuracy with *Intelligent-BFS* performing marginally (2-5%) better than *Modified-BFS*. *Random Walks*, *GUESS* and *s-APS* show decreasing accuracy as popularity drops, with *s-APS* clearly outperforming the other two. This difference becomes large for medium-popular objects. *DRLP* exhibits increasing accuracy as the popularity drops. This can be explained by the fact that less popular objects receive considerably fewer queries. Therefore, object relocations and node departures which affect the algorithm happen less frequently during requests for such objects.

Figure 5 shows how the number of hits is affected by the number of requests per object. Each object is uniformly stored in about 2% of the network nodes. The two BFS schemes show a stable performance as in the previous simulations, with the exception that *Modified-BFS* now produces more messages and discovers a few more objects than *Intelligent-BFS* (3900 to 3500 messages, 70 to 65 hits respectively). We notice that *GUESS* and *Random Walks* do not gain from the increased workload and exhibit results similar to the previous simulations. *s-APS* takes advantage of the increased requests to discover almost 7 times more objects. On the other hand, *DRLP* discovers a decreasing number of documents (nearly 7 times fewer hits at the final run), as the requests for each object increase. This happens because with more requests, fewer nodes perform flooding and only one object is discovered in almost every search. At the same time, the message production of *DRLP* also drops for the same reason.

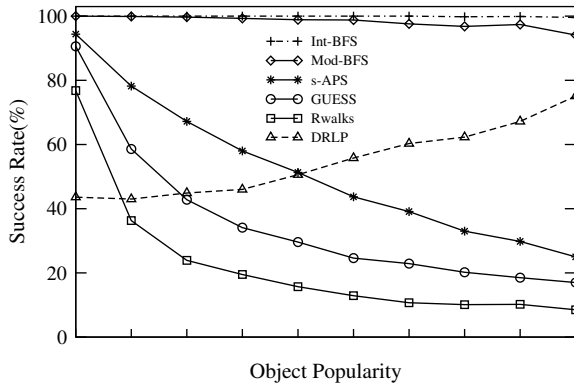


Figure 4: Success rate vs object popularity. Popularity decreases from left to right

5. CONCLUSIONS

This paper presents an overview of current search techniques for *unstructured* P2P networks. Our analysis and simulations focus on three metrics: accuracy, bandwidth production and discovered objects. Flood-based schemes (e.g. *Int-BFS*, *Mod-BFS*) exhibit high performance at a very high cost. Other *blind* methods (e.g. *Random Walks*, *GUESS*) are simple and can greatly reduce bandwidth production but generally fail to adapt to different workloads and environments. Conversely, most *informed* methods achieve great results but incur large overheads due to index updates. *DRLP* and *s-APS* require no costly updates. The former performs best in relatively static environments, while the latter uses its adaptive nature to achieve good performance at low cost. *s-APS* particularly favors nodes with a prolonged stay in the network and the discovery of popular objects.

For future work, we plan on giving emphasis to the index update process and the imposed overhead over the network. This will allow for a more thorough comparison of the informed search methods.

6. REFERENCES

- [1] The impact of file sharing on service provider networks. An Industry White Paper, Sandvine Inc.
- [2] openP2P website: <http://www.openp2p.com>.
- [3] Peer-to-peer working group: <http://www.peer-to-peerwg.org/>.
- [4] Project JXTA: <http://www.jxta.org>.
- [5] Microsoft .NET: <http://www.microsoft.com/net>.
- [6] D. Milojevic, V. Kalogeraki, R. Lukose, K. Nagaraja, J. Pruyne, B. Richard, S. Rollins, and Z. Xu. Peer-to-Peer Computing. Technical Report HPL-2002-57, HP, 2002.
- [7] Gnutella website: <http://gnutella.wego.com>.
- [8] Napster website: <http://www.napster.com>.
- [9] S. Saroiu, P. Gummadi, and S. Gribble. A measurement study of peer-to-peer file sharing

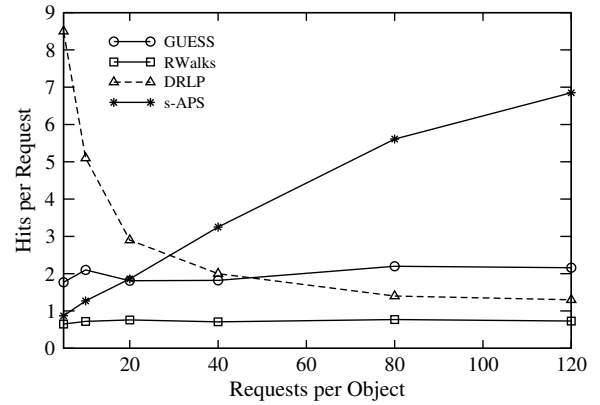


Figure 5: Number of hits as the number of requests per object increases in a static network

systems. Technical Report UW-CSE-01-06-02, Un. of Washington, 2001.

- [10] J. Chu, K. Labonte, and B. Levine. Availability and Locality Measurements of Peer-to-Peer File Systems. In *SPIE*, 2002.
- [11] S. Sen and J. Wang. Analyzing peer-to-peer traffic across large networks. In *SIGCOMM Internet Measurement Workshop*, 2002.
- [12] S. Saroiu, K. Gummadi, R. Dunn, S. Gribble, and H. Levy. An Analysis of Internet Content Delivery Systems. In *OSDI*, 2002.
- [13] V. Kalogeraki, D. Gunopulos, and D. Zeinalipour-Yazti. A Local Search Mechanism for Peer-to-Peer Networks. In *CIKM*, 2002.
- [14] C. Lv, P. Cao, E. Cohen, K. Li, and S. Shenker. Search and Replication in Unstructured Peer-to-Peer Networks. In *ICS*, 2002.
- [15] D. Tsoumakos and N. Roussopoulos. Adaptive Probabilistic Search (APS) for Peer-to-Peer Networks. Technical Report CS-TR-4451, Un. of Maryland, 2003.
- [16] S. Daswani and A. Fisk. Gnutella UDP Extension for Scalable Searches (GUESS) v0.1.
- [17] A. Crespo and H. Garcia-Molina. Routing Indices for Peer-to-Peer Systems. In *ICDCS*, July 2002.
- [18] B. Yang and H. Garcia-Molina. Improving Search in Peer-to-Peer Networks. In *ICDCS*, 2002.
- [19] A. Singla and C. Rohrs. Ultrapeers: Another Step Towards Gnutella Scalability.
- [20] M. Stokes. Gnutella2 Specifications Part One: http://www.gnutella2.com/gnutella2_search.htm.
- [21] S. Rhea and J. Kubiatowicz. Probabilistic Location and Routing. In *INFOCOM*, 2002.
- [22] D. Menascé and L. Kanchanapalli. Probabilistic Scalable P2P Resource Location Services. *SIGMETRICS Perf. Eval. Review*, 2002.
- [23] E. Zegura, K. Calvert, and S. Bhattacharjee. How to model an internetwork. In *Infocom*, 1996.

ODISSEA: A Peer-to-Peer Architecture for Scalable Web Search and Information Retrieval

Torsten Suel* Chandan Mathur Jo-Wen Wu Jiangong Zhang
Alex Delis Mehdi Kharrazi Xiaohui Long Kulesh Shanmugasundaram

Department of Computer and Information Science
Polytechnic University
Brooklyn, NY 11201

ABSTRACT

We consider the problem of building a P2P-based search engine for massive document collections. We describe a prototype system called ODISSEA (Open DISTRibuted Search Engine Architecture) that is currently under development in our group. ODISSEA provides a highly distributed global indexing and query execution service that can be used for content residing inside or outside of a P2P network. ODISSEA is different from many other approaches to P2P search in that it assumes a two-tier search engine architecture and a global index structure distributed over the network.

We give an overview of the proposed system and discuss the basic design choices. Our main focus is on efficient query execution, and we discuss how recent work on top- k queries in the database community can be applied in a highly distributed environment. We also give preliminary simulation results on a real search engine log and a terabyte web collection that indicate good scalability for our approach.

1. INTRODUCTION

Due to the large size of the Web, users increasingly rely on specialized tools to navigate through the vast volumes of data, and a number of search engines, directories, and other IR tools have been built to fill this need. While there is a plethora of smaller specialized engines and directories, the main part of the search infrastructure of the web is supplied by a handful of large crawl-based search engines, such as Google, AllTheWeb, AltaVista, and a few others. Such search engines are typically based on scalable clusters, consisting of a large number of low-cost servers located at one or a few locations and connected by high-speed LANs or SANs [4]. A lot of work has focused on optimizing performance on such architectures, which support up to tens of thousands of user queries per second on thousands of machines.

The last few years have also seen an explosion of activity in the area of peer-to-peer (P2P) systems, i.e., highly distributed computing or service substrates built from thousands or even millions of typically non-dedicated nodes across the internet that may join or leave the system at any time. Examples range from widely used unstructured ad-hoc communities such as Napster, Gnutella, and FreeNet to recent academic work on scalable and highly structured peer-to-peer substrates such as Chord [31], Tapestry [39], Pastry [28], or CAN [25] that can support a variety of applications.

From the perspective of search engines and large-scale IR this development raises two interesting issues. First, since an increasing amount of content now resides in P2P networks, it becomes necessary to provide search facilities within P2P networks. Second, the significant computing resources provided by a P2P system could also

be used to implement search and data mining functions for content located outside the system, e.g., for search and mining tasks across large intranets or global enterprises, or even to build a P2P-based alternative to the current major search engines. This second issue can be seen in the context of the following more general question: Which of the *Giant Scale Services* [4] currently provided by cluster-based architectures can and should be provided by more highly distributed or P2P systems? It has been established that applications such as the sharing of large static files can be very efficiently implemented in a P2P environment. However, other applications that, e.g., involve frequent updates to massive data, are more challenging, and may turn out to be more appropriately implemented on clusters or on highly-robust distributed systems of dedicated nodes with limited changes in topology (due to faults, or nodes joining or leaving).

In this paper, we describe a prototype system called ODISSEA (Open DISTRibuted Search Engine Architecture) that is currently under development in our group. ODISSEA attempts to address both of the above issues, by providing a “distributed global indexing and query execution service” that can be used for content residing inside or outside of a P2P network. ODISSEA is different in several ways from many other approaches to P2P search, as explained below. It encounters some basic challenges typical of those that arise when implementing more dynamic applications involving frequent updates on P2P systems, leading to interesting algorithmic problems and solutions. We describe and discuss the basic design choices and motivation and give some initial results, with focus on the issue of efficient distributed query processing.

1.1 ODISSEA Design Overview

ODISSEA is a distributed global indexing and query execution service, i.e., a system that maintains a global index structure under document insertions and updates and node joins and failures, and that executes simple but general classes of search queries in an efficient manner. This system provides the lower tier of a proposed two-tier search infrastructure. In the upper tier, there are two classes of clients that interact with this P2P-based lower tier:

1. *Update clients* insert new or updated documents into the system, which stores and indexes them. An update client could be a crawler inserting crawled pages, a web server pushing documents into the index, or a node in a file sharing system.
2. *Query clients* design optimized query execution plans, based on statistics about term frequencies and correlations, and issue them to the lower tier. Ideally, the lower tier should enable query clients to use or implement various ranking methods.

There are two main differences that distinguish ODISSEA from other P2P search systems. First, the assumption of a two-tier architecture that aims to give as much freedom as possible to clients to implement their own user interfaces and search and ranking policies. This is motivated by the goal of providing an “open” search infrastructure that

*Contact author. Email: suel@poly.edu. Research partly supported by NSF CAREER Award NSF CCR-0093400 and by the Othmer Institute for Interdisciplinary Studies at Polytechnic University.

Copyright is held by the author/owner.
International Workshop on the Web and Databases (WebDB).
June 12–13, 2003, San Diego, California.

allows the creation of a rich variety of client-based search and navigation tools running on user desktops. There are trade-offs between efficiency and flexibility that may limit the full realization of this goal, and one of our main research goals is to investigate these trade-offs.

The second difference is our assumption of a *global inverted index* structure. Many current approaches (see [15, 19, 26] for exceptions) to full-text search in P2P systems assume a *local inverted index*, where each node maintains an index for all local documents (or the documents of a few surrounding nodes), and queries are broadcast to all, or on average at least a significant fraction, of the nodes, in order to get the best results. In a global index, the inverted index for a particular term (word) is located at a single node, or partitioned over a small number of nodes in some hybrid organizations. Thus, queries with multiple keywords require “combining” the data for the different keywords over the network, at possibly significant cost. We discuss this decision later as it has consequences for the overall design.

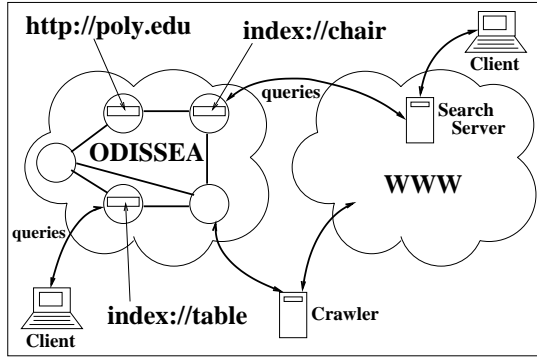


Figure 1: ODISSEA as a web search infrastructure, with a web crawler as update client, and two query clients (one client-based and one as a web-based search service). Also shown are indexes for the words “chair” and “table”, and a node holding the document `http://poly.edu`.

Figure 1 shows the basic design. We decided to implement the system on top of an underlying global address space provided by a DHT structure, in particular Pastry [28]. Each object is identified by a hash of its *name* (i.e., a URL or a string such as `index://chair` for the index structure for the term *chair*) and is assigned a location determined by the DHT mapping. Thus, the only way to move an object is to rename it, resulting in a mapping to a random other node. (We note that the real mapping scheme is actually more complicated, to enable replication and load balancing.)

1.2 Target Applications

We have four main application scenarios that motivate our research:

- (1) **Search in P2P networks:** To provide full-text search for large document collections located within P2P communities.
- (2) **Search in large intranet environments:** Large organizations may use distributed search to share machine resources within more controlled and maybe less bandwidth-limited environments.
- (3) **Web search:** Our most ambitious application is a P2P-based search infrastructure for the web that provides an alternative to the major search engines, with a powerful API (more low-level than, e.g., the Google API) that supports the anticipated shift towards client-based search tools that exploit the resources of today's desktop machines. This scenario may not be feasible in the near term but we believe still deserves study.
- (4) **Search middleware:** Instead of inserting documents, clients could directly insert “postings”, i.e., index entries. The system would then act as “global middleware” on top of a system of local index structures, where nodes might periodically insert

some of their postings into the system. The middleware could then use a combination of local and global indexes for query processing, resulting in increased efficiency for certain queries.

Paper outline: In the next section we justify our main design decisions and assumptions. Technical details and preliminary experimental results on query processing are provided in Section 3. Section 4 discusses related work, and Section 5 mentions some open problems. A more detailed version of this paper appears in [32], and up to date information on the ODISSEA project is available at <http://cis.poly.edu/westlab/odissea/>.

2. DISCUSSION AND JUSTIFICATION

Two-tier approach: This choice was originally motivated by the web search application scenario. Given the expected increases in speed and bandwidth of desktop systems, we see the potential for a rich variety of novel search and navigational tools and interfaces that more fully exploit client computing resources, and that rely on access to a powerful lower-level web search infrastructure. These tools may perform a large number of web server or search engine accesses during a single user interaction, in order to prefetch, analyze, aggregate, and render content from various sources into a highly optimized form. Early examples of these types of client-based tools are browsing assistants such as the Alexa and Google Toolbars, Zapper, Leticia and PowerScout [20], the Stanford Power Browser [8], or tools built with the Google API. Specialized search engines (Google News, citeSeer) or meta engines could also be supported by such an infrastructure.

Thus, the proposed system could be used to provide such a lower-level search infrastructure, with an powerful open and *agnostic* API that is accessed by client- and proxy-based tools. By *agnostic*, we mean an API that is not limited to a single method for ranking pages (e.g., the Google API, which returns pages according to Google's ranking strategy), but that ideally allows clients to implement their own ranking strategies. There clearly are limits and trade-offs to this goal. The most general solution of performing most of the ranking at the client requires large amounts of data to be transferred. On the other hand, we conjecture that limited but powerful classes of ranking functions could be efficiently supported by providing appropriate “hooks” and algorithmic techniques inside the system.

Global vs. local index: The other important decision is the use of a global index instead of the more commonly used local index organization. We now define some terms. First, an *inverted index* for a document collection is a data structure that contains for each word in the collection a list of all its occurrences, or a list of *postings*. Each *posting* contains the document ID of the occurrence of the word, its position inside the document, and other information such as whether the word is in the title or in bold face. Each postings list is best visualized as an array sorted by document ID.

In a local index organization, each node creates its own index for all documents that are locally stored. Thus, every node will have its own small postings list for common words such as *chair* or *table*, and a query *chair, table* is first broadcast to all nodes and then the results are combined. In a global index organization, each node holds a complete global postings list for a subset of the words, as determined, e.g., by a hash function. Thus, every node has a smaller number of longer lists, and under the standard query evaluation strategy a query *chair, table* is first routed to the node holding the list for *chair* (the shorter list), which then sends its complete list to the node holding the list for *table*. We emphasize here that our approach does in fact not send the entire list, as explained later. There have been a number of performance comparisons between local and global index organizations and several hybrid organizations on parallel architectures [3, 9, 34], but these do not directly apply to widely distributed environments.

The main issue with local index organizations is that all or most nodes need to be contacted for most queries, and thus such schemes

are unlikely to scale beyond a few hundred nodes. There have been attempts to overcome this issue by routing queries only to those nodes that are likely to have good results¹ or are in the vicinity [11, 18, 29]. However, we do not believe that this approach will scale if result quality is a major concern, since document collections are simply not naturally clustered in a way that allows queries to be routed to only a small fraction of the nodes. This is certainly the case for the current web, where a search infrastructure based on local indexes at each site would be extremely inefficient. This could be somewhat improved by clustering the entire document collection, though this seems quite challenging to do [19]. Moreover, the statistics needed to intelligently route queries are quite large for large collections and many nodes as the number of distinct words grows with collection size; the existing literature has only evaluated collections up to a few gigabytes.

In a global index organization, however, large amounts of data need to be transmitted between nodes, since large collections result in lists of megabytes or more for all except fairly rare words. This has led some people to reject global indexes as unrealistic for environments with limited bandwidth, and for moderate numbers of nodes a local index is probably a better choice. However, we believe that this problem can be overcome through smart algorithmic techniques. One technique was recently applied in this context in [19, 26], where Bloom filters are used to decrease the cost of intersecting lists of postings over the network, though this only improves results by a constant factor. We study in Subsection 3.1 how recent results on top- k queries in the database literature [13] can be applied to asymptotically reduce communication requirements. We believe that these techniques, combined with other query optimization techniques, allow interactive response times even on massive data sets.

Crawling punt: We assume in the web search application that crawling is performed by *crawling clients* that fetch and insert documents. The main reason is that from our own experiences with large-scale crawling [30] we are not sure a P2P solution is appropriate. Large crawls generate many management issues due to queries or complaints from web site operators and network administrators. It is important to be able to reconfigure a crawler quickly to avoid web sites or subnetworks or to modify its behavior, and failure to do so can result in problems with local administrators or upstream providers.² Moreover, smart crawling strategies beyond BFS are hard to implement in a P2P environment without a centralized scheduler.

Thus, we would expect that a handful of powerful crawling clients would provide most documents, and we plan to use our Polybot crawler [30] to initially populate the system with data. It might be more feasible to incorporate recrawling into the system, though. Thus, an inserted page could be labeled with an expiration date, after which it is automatically refreshed by the node holding the page. Alternatively, web sites could also push their pages into the system.

P2P systems and fault tolerance: Utilizing idle remote resources is one of the main motivations for building P2P systems. However, there is a fundamental challenge facing applications that use large amounts of disk space on remote nodes, such as a search engine. Given current network speeds, it would take days or weeks to transfer enough data to a newly joined node to utilize any significant fraction of a 200 GB disk, and during this time the node would probably consume more resources than it adds to the system. Thus, such applications are maybe best restricted to the more stable end of the P2P spectrum, where most nodes remain in the system for longer times.

Our system design relies on this assumption of a more stable system. However, we distinguish between nodes that are temporarily unavailable and nodes that have permanently left the system. When

a node rejoins after an extended period of unavailability, an interesting problem arises: how do we efficiently *synchronize* its data structures, in this case the index structures, with an up-to-date copy held by another node, to incorporate any updates missed while unavailable? Other problems involve distinguishing between failed and unavailable nodes, when to rebuild data on failed or unavailable nodes, and how quickly data should be pushed to newly joined nodes.

3. QUERY PROCESSING IN ODISSEA

In this section we describe query processing in the proposed system. A naive implementation of ranked queries with a global index structure would result in transfers of many megabytes of data for many queries from a typical query load. Since realistic bandwidths in WAN environments are on the order of a few hundred Kb/s, this would result in response times of many seconds or even minutes. We now describe how to adapt recent techniques by Fagin and others [12, 13, 14] to our scenario, and give measurements of the expected savings based on a real search engine query log and a set of 120 million web pages from a recent crawl that we have carried out.

3.1 Background and Algorithmic Techniques

Ranking in search engines: We first give some background on ranking in search engines. Search engines rank pages based on many criteria, including classical term-based techniques from IR, global page ranks as provided by Pagerank [5] and similar methods, whether text is in bold face or within a hyperlink, and distances between the search terms in the documents, among others. Formally, a *ranking function* is a function F that, given a query consisting of a set of search terms q_0, q_1, \dots, q_{m-1} , assigns to each document d a score $F(d, q_0, \dots, q_{m-1})$. The top- k ranking problem is then the problem of identifying the k documents in the collection with the highest scores. We focus on two families of ranking functions,

$$F(x) = \sum_{i=0}^{m-1} f(d, q_i) \quad \text{and} \quad F(x) = g(d) + \sum_{i=0}^{m-1} f(d, q_i).$$

The first family includes the common families of term-based ranking functions used in IR, where we add up the scores of each document with respect to all words in the queries. In particular, this includes the well-known class of *cosine measures*; see, e.g., [37]. The second formula adds a query-independent value $g(d)$ to the score of each page; this could for example be a suitably normalized Pagerank value. Thus, these two families include many important ranking functions, and we could in fact use any other monotone function instead of addition to combine the various functions in the above formula. Note however that techniques using the distances between the terms in a document would lead to an additional function $h(d, q_0, \dots, q_{m-1})$ that depends on all terms; this would impact the efficiency of our methods.

Queries to search engines have on average less than three terms, and engines typically evaluate a query by considering all documents in the intersection of the inverted lists, i.e., all documents that contain all search terms.³ An information-theoretic argument shows that determining the intersection of two lists located at different nodes requires transmitting an amount of data linear in the size of the shorter list. However, recent work in the database community [13] shows how to evaluate top- k queries without scanning the entire intersection.

Fagin's Algorithm (FA): We now describe the first algorithm, which was originally proposed in [12] for the case of *multimedia queries*, e.g., to retrieve images from an image database. We will state them directly for our scenario, first for the case of the first family of ranking functions without $g(d)$. Intuitively, the algorithm exploits the fact that

¹This is also known as the database selection problem [21].

²Of course, for certain types of crawling activities, e.g., to surreptitiously monitor certain web sites, a P2P solution may be preferable for the very same reasons.

³This is in contrast to "traditional" IR systems that tend to consider the union of the lists, and where typical queries consist of a dozen terms or more. Our results do not really depend on this choice.

an item that is ranked in the top is likely to be ranked very high in at least one contributing subcategory.

Consider the inverted lists for a search query with two terms q_0 and q_1 . For the moment, assume they are located on the same machine, and that the postings in the list are pairs $(d, f(d, q_i))$, $i \in \{0, 1\}$, where d is an integer identifying the document and $f(d, q_i)$ is real-valued. Assume each inverted list is sorted by the second attribute, so that documents with largest $f(d, q_i)$ are at the start of the list. Then the following algorithm, called *FA*, computes the top- k results:

- (1) Scan both lists from the beginning, by reading one element from each list in every step, until there are k documents that have each been encountered in both of the lists.
- (2) Compute the scores of these k documents. Also, for each document that was encountered in only one of the lists, perform a lookup into the other list to determine the score of the document. Return the k documents with the highest score.

It is not difficult to see that this indeed returns the top- k results overall. It is shown in [12] that if the orderings of documents in the two lists are independent, then the algorithm terminates after looking at only $O(\sqrt{kn})$ entries in each list, where n is the number of documents in the collection (not the length of the list). In the case of queries with m terms, the bound becomes $O(n^{\frac{m-1}{m}} k^{\frac{1}{m}})$. Thus, for long lists this significantly improves over scanning the entire list. If terms are positively correlated, then the result improves, while it gets worse for negatively correlated terms. Note that the result is independent of the actual “shapes” of the distributions of the $f(d, q_i)$, though refinements could potentially exploit special distributions such as Zipfians.

Threshold Algorithm (TA): The following refinement was proposed by several authors; see [13] for a discussion. We again simultaneously scan both lists, so that in each step we read an item $(d, f(d, q_0))$ from the first and an item $(d', f(d', q_1))$ from the second list. In each step we compute $t = f(d, q_0) + f(d', q_1)$; note that d and d' will usually be different documents. Also, whenever we encounter a document in one list, we immediately perform a lookup into the other list to compute its complete score. As soon as we have found k items with score larger than the current t , we return these as results. It can be shown that *TA* is correct and always terminates at least as early as *FA*, though the asymptotic bounds are the same.

Integrating query-independent scores: We can naively adapt both algorithms to the second family of ranking functions as follows. Instead of sorting each list by $f(d, q_i)$, we sort by $f(d, q_i) + \frac{1}{2} \cdot g(d)$, so that the total score is the sum of the sort attributes from both lists. Note that this should increase efficiency, as it introduces significant correlation between the orderings of the two lists.

However, in reality we cannot combine term-based and link-based scores simply by adding them up. Instead, it is preferable to normalize the scores in a query-dependent way that minimizes the effect of outliers. Following [27] we do this by normalizing using the mean of the top-100 term-based and link-based scores that appear in the two (or more) lists; see [27] for details. This means that the inverted lists cannot be completely organized in sorted order before the arrival of the query, though they can usually be kept approximately sorted. In our distributed setting this is not a problem since we are interested in minimizing bandwidth consumption rather than CPU cost.

3.2 Experimental Results on Real Data

We run some initial experiments to determine the potential savings of these schemes. Note that these experiments are in a centralized setting; we consider distributed implementations in the next subsection. There have been previous evaluations of the *FA* and *TA* algorithms on data sets from other application domains, but not on large-scale web data or in conjunction with global measures such as Pagerank.

For the experiments, we use queries selected from a log of over 1 million queries posted to the Excite search engine on December

	Top 1	Top 10	Top 100
Lists	1,056,746	1,056,746	1,056,746
Intersect	654	1,536	8,954
FA(cosine only)	7,860	17,087	47,024
TA(cosine only)	2,445	6,353	17,962
CA(cosine only)	932	2,978	13,585
FA(cosine + pagerank)	6,137	11,046	37,991
TA(cosine + pagerank)	1,652	4,651	16,533
CA(cosine + pagerank)	529	1,785	11,025

Table 1: Average costs on a data set of 120 million pages.

20, 1999. Our document collection consists of about 120 million web pages crawled by the Polybot crawler [30] in October 2002, for a total of about 1.8 TB of data. The numbers reported here are limited to a few hundred queries with two terms. Also, stop words were removed, as done by many engines, and we removed queries with less than 100 results in the intersection. For the first family of ranking functions, we used a standard cosine measure. For the second family, we defined $g(d)$ as an appropriately normalized Pagerank score computed from a web graph extracted from our crawl.

Table 1 shows the average number of postings that have to be scanned from each list under the various algorithms. In the first row we have the number of postings in the shorter of the two inverted lists; this represents the cost incurred by the unoptimized algorithm where we transmit the entire list. In the next line, we have the number of postings that are scanned if we are only interested in getting an arbitrary k elements that contain both query terms. This is a reasonable lower bound⁴ on what we could hope to achieve with the optimized methods, and was measured by ordering indexes by document ID and scanning from the beginning until k elements in the intersection are found. We note that this cost can in some cases be quite significant, say for two inverted lists of length $\Theta(\sqrt{n})$ where we might have to scan most of the lists. In the experiments, we also include an idealized algorithm called *CA* (Clairvoyant Algorithm) that stops as soon as it has encountered the top- k elements; this shows the cost between finding the top- k results and being certain that we have found them.

We show results for *FA*, *TA*, and *CA*, with and without Pagerank. All three algorithms perform significantly better than the basic algorithm. The results for *TA* and *CA* show that we can usually terminate the scan much earlier without impact on the result. Including the Pagerank score usually results in improved performance. The results indicate that an appropriate distributed protocol based on these algorithms might have the potential to achieve interactive response times in WAN environments even for massive data sets.

3.3 A Simple Distributed Protocol

We now adapt these techniques to a highly distributed environment with limited bandwidth as well as high latency. Thus, we have to limit ourselves to one or a few roundtrips between the nodes holding different inverted lists. There is also a potential bottleneck in the random lookups performed by the *FA* and *TA* algorithms. In a high-bandwidth environment, this is a serious drawback of the algorithms since large index structures have to reside on disk. As a result, other pruning methods have been proposed for this case [1, 24] that avoid such accesses but instead need to scan a significant part of the inverted lists. In a P2P environment this is less of a concern, and a large set of random lookups could be resolved by performing a local scan over the inverted list. Following is our proposed distributed implementation, called *DPP* (Distributed Pruning Protocol), for the case of two search terms and a ranking function from the first family (i.e., without $g(d)$).

- (1) The node holding the shorter list, called node *A*, sends the first x postings of its inverted list to node *B*. (Assume for the mo-

⁴If we discount correlations between query terms.

	shortest 30%	middle 30%	longest 30%
Lists	23,620	305,557	3,092,772
Postings A to B	5,105	7,405	4,264
Postings B to A	5,572	7,360	4,183
Tot. bytes sent	85,336	118,120	67,576
Time 400kbps (ms)	2,456	3,151	2,077
Time 2mbps (ms)	977	1,151	882

Table 2: Communication costs and times for top-10 queries.

ment that A somehow knows the best value of x .) Also, let r_{min} be the smallest (last) value $f(d, q_0)$ transmitted.

- (2) Node B receives the postings from A , and performs lookups into its own list to compute the total scores of the corresponding documents. Retain the k documents with the highest score. Let r_k be the smallest score among these.
- (3) Node B now transmits to A all postings among its first x postings with $f(d, q_1) > r_k - r_{min}$, together with the total scores of the k documents from Step (2).
- (4) Node A performs lookups into its own list for the postings received from B , and determines the overall top k .

One remaining question is how to choose the value of x . This could be done by deriving appropriate formulae based on extensive testing. Alternatively, we could use sampling-based methods [6] to estimate the number of documents appearing in both prefixes. In either case, a wrong estimate could be corrected at the cost of an extra roundtrip.

3.4 Evaluation of DPP

In our system, we open a new TCP connection between the participating nodes for each query. To model the effect of the TCP congestion window on performance, which is significant in our scenario, we use a model for file transfer cost under TCP recently proposed in [36] with typical parameters for a broadband connection between the East and West coast of the US.⁵ In particular, we assume a roundtrip signaling delay of 50ms, and a bandwidth limit between 400 kbits and 2 mbits per second on the first and last leg. For both directions, we incur the cost due to the congestion window, and for the first message we have the additional cost of establishing the connection.

We assume each posting is transmitted in 8 bytes, as follows: We hash the 80-bit document IDs down to z bits, where z is chosen such that the likelihood of a collision between the transmitted prefix and the other list is less than, say, 0.1%. We then encode the hashes using standard gap compression techniques [37]. This results in at most 40 to 48 bits per hash; the remaining bits are used for an approximation of the term value $f(d, q_0)$. The protocol could be adapted to recognize when a collision occurs, in which case an additional roundtrip is used to fix the problem. (Observe that the scheme is a bit like using a very precise compressed Bloom filter with one hash function.)

Table 2 shows the estimated cost of the algorithm, using the same data set as before. There are two assumptions in the measurements. First, we choose the length x of the prefix that should be sent from A to B by using the results of the experiments on the TA algorithm. This is optimistic since the parties do not have these results available; on the other hand, the results from the CA algorithm indicate that even a low estimate would often return the correct result (or we could choose an additional roundtrip to be sure). Second, we do not measure internal computation within nodes. Of course, this internal computation is also incurred by standard (non-P2P) search engines, and most of it is overlapped with communication anyway. We believe that neither of these assumptions changes the measurements fundamentally.

Large engines such as Google in fact use data sets that are 20 to 30 times larger than ours. According to the theoretical bound of \sqrt{kN}

⁵The model in [36] is similar to others that have been proposed.

this would result in an additional factor of about 5 on the amount of data transmitted. Use of more than two keywords would also increase communication. On the other hand, the above algorithm is really only a baseline as discussed in the following.

3.5 Optimizing Query Execution Plans

The above protocol is a first step towards efficient query execution. There are two ways to get further improvements: (1) use of Bloom filters as studied in [26], and (2) use of a hybrid partitioning where large inverted lists are split among several nodes [32]. We note that the second approach does not actually decrease the total cost of a query, but it can improve latency by splitting communication and computation among several nodes. As it turns out, Bloom filters can be combined in several interesting ways with our protocol. The end result is that there are a large number of possible ways to execute a query on three or more search terms. We are currently studying in detail how to derive the best possible plans.

The design of a good query plan is up to the query client in our system, and is done in two phases. The client first inquires basic statistics such as term frequencies, mean values for the normalization, and possibly samples [6] to estimate term correlations from the system. The system returns the statistics and the IP addresses of the nodes holding the lists. This type of information can be very efficiently cached in the system as it is small compared to the rest of the data. In fact we really only need to keep statistics for inverted lists of significant length (e.g., more than a few thousand postings).

Given the statistics, the client knows which term has the shortest inverted list, and which of the lists are partitioned between several nodes. Next, a query plan is designed as a directed labeled graph, where the nodes are nodes in the network identified by address, and the edges are labeled with the operation to be performed, e.g., send complete list if small, send a Bloom filter of the list, send a prefix as done in the baseline DPP protocol, or send a Bloom filter of a prefix.

4. RELATED WORK

There has been a lot of recent interest in the pruning techniques of Fagin et. al [12, 14]; see also [13] for a survey and [10] for early related ideas. Most of the interest has been focused on multimedia and meta search scenarios, and we are not aware of previous applications in a peer-to-peer environment. On the other hand, there has also been significant work in the IR community, much of it preceding the above, on pruning techniques for vector space queries. Some early work is described in [7, 17, 24, 35, 38], and more closely related recent work is in [1, 2]. One difference between these two strains of work is that in the IR case, a random lookup of a posting is much more expensive than scanning. Thus, recent pruning techniques from IR typically restrict access to scans, resulting in more limited savings. We are mainly concerned with bandwidth, making this less of an issue.

There has been significant interest in search in distributed and P2P systems over the last few years. We note, however, that the problem of full-text search on terabyte-size collections is different from that on smaller collections or on systems that only index titles and keywords for multimedia objects (e.g., mp3 files). Some recent work on text search in P2P systems with local index organization appears in [11, 18, 29, 33]. As explained, the global index organization is one of the aspects that distinguish our system from others. Another very different approach to distributed search is taken by systems such as JXTA [22], STARTS [16], and the Z39.50 standard [23], which are mainly concerned with issues of combining outputs from diverse search tools.

Global index organizations in a peer-to-peer environment have recently been discussed in [15, 19, 26]. The work by Reynolds and Vahdat [26] considers the benefits of using Bloom filters instead of sending an entire inverted list during query execution. Subsequent work in [19] estimates the potential benefit of using a combination of techniques, including Bloom filters, clustering, compression, caching, and

adaptive set intersection, compared to the naive algorithm that transfers the entire list. The paper concludes that these techniques together save a significant constant factor and bring the approach close to feasibility for terabyte data sets. The authors also mention the possibility of using Fagin's pruning technique [13] for additional improvements, but no details are provided. Combining Fagin's technique with those in [19] is possible, as indicated in Subsection 3.5, but the details are tricky and the returns diminish as more techniques are applied.

5. OPEN QUESTIONS AND FUTURE WORK

In this paper, we have given an overview of the ODISSEA system, and presented some early results on query processing in the system. There are numerous open questions for future work. We are currently working on a framework for generating optimized query execution plans for multi-keyword queries based on a combination of pruning techniques, Bloom filters, and compression. Once this is complete, we plan to perform a more thorough experimental evaluation for queries with multiple keywords and phrase searches, and for ranking functions that use term distance within documents.

We are also studying techniques for synchronizing outdated indexes and for load balancing and rebuilding of lost replicas in an environment where nodes hold large amounts of data but may be temporarily unavailable. Beyond these specific items, the general question remains whether the near future will see massive P2P-based systems for challenging applications such as web search and large-scale IR, beyond simple applications such as file sharing.

Acknowledgements: We thank Hojun Lee and Malathi Veeraraghavan for their help with the TCP performance model.

6. REFERENCES

- [1] V. Anh, O. Kretser, and A. Moffat. Vector-space ranking with effective early termination. In *Proc. of the 24th Annual SIGIR Conf.*, pages 35–42, September 2001.
- [2] V. Anh and A. Moffat. Compressed inverted files with reduced decoding overheads. In *Proc. 21st Annual SIGIR Conf.*, 1998.
- [3] C. Badue, R. Baeza-Yates, B. Ribeiro-Neto, and N. Ziviani. Distributed query processing using partitioned inverted files. In *Proc. of the 9th String Processing and Information Retrieval Symposium (SPIRE)*, September 2002.
- [4] E. Brewer. Lessons from giant scale services. *IEEE Internet Computing*, pages 46–55, August 2001.
- [5] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. In *Proc. of the 7th WWW Conference*, 1998.
- [6] A. Broder. On the resemblance and containment of documents. In *Compression and Complexity of Sequences*, 1997.
- [7] C. Buckley and A. Lewit. Optimization of inverted vector searches. In *Proc. 8th SIGIR Conf. on Research and Development in Information Retrieval*, June 1985.
- [8] O. Buyukkokten, H. Garcia-Molina, A. Paepcke, and T. Winograd. Power browser: Efficient web browsing for PDAs. In *Proc. of the Human-Computer Interaction Conference*, 2000.
- [9] B. Cahoon, K. McKinley, and Z. Lu. Evaluating the performance of distributed architectures for information retrieval using a variety of workloads. *IEEE Transactions on Information Systems*, 18(1):1–43, January 2000.
- [10] S. Chaudhuri and L. Gravano. Optimizing queries over multimedia repositories. *Data Engineering Bulletin*, 19(4):45–52, 1996.
- [11] F. Cuenca-Acuna and T. Nguyen. Text-based content search and retrieval in ad hoc p2p communities. In *Proc. of The Int. Workshop on Peer-to-Peer Computing*, May 2002.
- [12] R. Fagin. Combining fuzzy information from multiple systems. In *ACM Symp. on Principles of Database Systems*, 1996.
- [13] R. Fagin. Combining fuzzy information: an overview. *SIGMOD Record*, 31(2):109–118, June 2002.
- [14] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *Proc. of ACM Symp. on Principles of Database Systems*, 2001.
- [15] O. Gnawali. A keyword-set search system for peer-to-peer networks. Master's thesis, MIT, 2002.
- [16] L. Gravano, C. Chang, H. Garcia-Molina, and A. Paepcke. STARTS: Stanford Proposal for Internet Meta-Searching. In *ACM SIGMOD Int. Conf. on Management of Data*, 1997.
- [17] D. Harman and G. Candela. Retrieving records from a gigabyte of text on a minicomputer using statistical ranking. *J. of the American Society for Information Science*, 41(8), August 1990.
- [18] A. Kronfol. FASD: a fault-tolerant, adaptive, scalable, distributed search engine. June 2002. Unpublished manuscript.
- [19] J. Li, B. Loo, J. Hellerstein, F. Kaashoek, D. Karger, and R. Morris. On the feasibility of peer-to-peer web indexing. In *Proc. of the 2nd Int. Workshop on Peer-to-Peer Systems*, 2003.
- [20] H. Lieberman, C. Fry, and L. Weitzman. Exploring the web with reconnaissance agents. *Communications of the ACM*, 44(8):69–75, August 2001.
- [21] W. Meng, C. Yu, and K. Liu. Building efficient and effective metasearch engines. *ACM Computer Surveys*, March 2002.
- [22] Sun Microsystems. JXTA. <http://www.jxta.org>.
- [23] National Information Standards Organization. Information Retrieval (Z39.50): Application Service Definition and Protocol Specification. Technical report, NISO, Bethesda, MD, 1995.
- [24] M. Persin, J. Zobel, and R. Sacks-Davis. Filtered document retrieval with frequency-sorted indexes. *J. of the American Society for Information Science*, 47(10), May 1996.
- [25] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proc. of the ACM SIGCOMM Conference*, 2001.
- [26] P. Reynolds and A. Vahdat. Efficient peer-to-peer keyword searching. February 2002. Unpublished manuscript.
- [27] M. Richardson and P. Domingos. The intelligent surfer: Probabilistic combination of link and content information in pagerank. In *Advances in Neural Information Processing Systems*, 2002.
- [28] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM Int. Conf. on Distributed Systems Platforms*, pages 329–350, November 2001.
- [29] Y. Shen and D. L. Lee. An mdp-based peer-to-peer search server network. In *Proc. of the 3th International Conf. on Web Information Systems Engineering*, pages 269–278, 2002.
- [30] V. Shkapenyuk and T. Suel. Design and implementation of a high-performance distributed web crawler. In *Proc. of the Int. Conf. on Data Engineering*, February 2002.
- [31] I. Stoica, R. Morris, D. Karger, M.F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proc. of ACM SIGCOMM Conference*, August 2001.
- [32] T. Suel, C. Mathur, J. Wu, J. Zhang, A. Delis, M. Kharrazi, X. Long, and K. Shanmugasunderam. Odissea: A peer-to-peer architecture for scalable web search and information retrieval. TR-CIS-2003-01, Polytechnic University, 2003.
- [33] C. Tang, Z. Xu, and M. Mahalingam. pSearch: Information retrieval in structured overlays. In *Proc. of ACM HotNets-I*, October 2002.
- [34] A. Tomasic and H. Garcia-Molina. Performance of inverted indices in distributed text document retrieval systems. In *Proc. of the 2nd Int. Conf. on Parallel and Distributed Information Systems (PDIS)*, 1993.
- [35] H. Turtle and J. Flood. Query evaluation: strategies and optimizations. *Information Processing and Management*, 31(6):831–850, November 1995.
- [36] M. Veeraraghavan, H. Lee, and R. Grobler. A low-load comparison of TCP/IP and end-to-end circuits for file transfers. In *Proc. of INET 2002*, June 2002.
- [37] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes*. Morgan Kaufmann, second edition, 1999.
- [38] W. Wong and D. Lee. Implementations of partial document ranking using inverted files. *Information Processing and Management*, 29(5):647–669, September 1993.
- [39] B. Zhao, J. Kubiawicz, and A. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Tech. Report UCB//CSD-01-1141, UC Berkeley, April 2000.

On Distributing XML Repositories

Jan-Marco Bremer and Michael Gertz

Department of Computer Science
University of California, Davis
One Shields Ave., Davis, CA 95616, U.S.A.
{bremer|gertz}@cs.ucdavis.edu

ABSTRACT

XML is increasingly used not only for data exchange but also to represent arbitrary data sources as virtual XML repositories. In many application scenarios, fragments of such a repository are distributed over the Web. However, design and query models for distributed XML data have not yet been studied in detail.

In this paper, we introduce a distribution approach for a virtual XML repository. We present a fragmentation method and outline an allocation model for distributed XML fragments. We also discuss an efficient realization based on small, local index structures. The index structures encode global path information and provide for an efficient, local evaluation of the most common types of global queries.

1. INTRODUCTION

Recent advancements in the development of native XML database systems (e.g., [5, 16]) clearly indicate that XML is not only considered for data exchange but also as a data representation format. The management of XML data in such native systems, compared to managing XML in (object-) relational database systems, leads to new application and research perspectives. In particular, we conjecture that distribution aspects of XML will play an important role. In fact, several application domains for XML, such as Web services, e-commerce, collaborative authoring of large electronic documents (e.g., WebDAV [14]), or the management of large-scale network directories [7], show that XML data is inherently distributed on the Web. Systems managing distributed XML data thus have to take this aspect into account to allow for an efficient and reliable usage of XML data at different sites.

Although concepts for the distribution of XML data are clearly important, to this day, there is only little related work that deals with distribution aspects of XML. In [13], distributed query evaluation techniques are investigated. Recently, in [1] the problem of distributed and replicated (dynamic) XML documents has been studied for the first time and fundamental query processing models have been proposed. Interestingly, the W3C has published a candidate recommendation for XML fragment interchange [6], but respective concepts are neither well-founded nor used in practice. While the above works address some distribution as-

pects for XML, to the best of our knowledge, there is no work that investigates the top-down design of a distributed XML data source, following traditional and well-studied distribution design principles for relational databases.

In this paper, we lay the groundwork for the distribution design of XML data in the context of large-scale XML repositories that manage XML data on the Web for the application scenarios mentioned above. In particular, we present a fragmentation scheme on a global conceptual schema structure and outline how fragments, as XML data, are allocated at different sites. We also present a realization of the distribution design in which small index structures at each site efficiently encode information about local and remote fragments. The index structures allow for processing most global queries at local sites without accessing other sites.

In Section 2, we introduce the data, schema, and query model underlying our approach. The fragmentation and allocation schemes are discussed in Section 3. In Section 4, we detail the representation of local and remote XML fragment information at local sites and illustrate how local and global queries are efficiently evaluated.

2. FOUNDATIONS

2.1 Data Model

As customary, we assume that XML data is modeled as a (single) rooted, node-labeled tree $(\mathbf{V}, \mathbf{E}, root, label, text)$. Nodes from the set \mathbf{V} are connected by edges from $\mathbf{E} \subseteq \mathbf{V} \times \mathbf{V}$ to form a tree with root node $root \in \mathbf{V}$. The function *label* assigns an element name (or label) to each node in \mathbf{V} . For presentation purposes, we assume that element attributes are modeled as regular nodes. The function *text* assigns text strings to nodes.

We assume a partial order among nodes in which only the relative positions among sibling nodes carrying the same label are considered significant. This order criterion is sufficient to support most types of queries. For example, while for a book element, the order of authors (as child nodes) matters, the order between title and author child nodes of a book element is likely much less relevant.

In the following, we use the notion *data source* or just *source* to refer to a representation of XML data according to the above tree model. Such a data source can be considered a single (large) XML document. For a data source, we furthermore adopt the common definitions of (*rooted*) *label paths* as

sequences of node labels (starting with the label of the root node), and analogously (*rooted*) *data paths* as sequences of source nodes. Naturally, several rooted data paths are associated with a single rooted label path. For example, Figure 3 highlights a rooted data path for the rooted label path `/DigitalLibrary/Loc/Books/Bk/A`.

2.2 RepositoryGuide

Underlying our XML distribution approach is a global (conceptual) schema. While it is conceivable that such a schema exists in the form of a DTD or XML Schema, we chose to use a simplified schema structure. This structure is called *RepositoryGuide (RG)* and resembles basic features of a DataGuide for tree-structured data in that all admissible rooted label path are enumerated.

There are several reasons for our choice. First, a tree-structured schema representation is easier to “fragment” or decompose than a grammar-based schema specification. Second, in the case of a large-scale XML repository, it is likely that local sites express their information and data representation needs using a schema formalisms that is more expressive than a (local) RG. Such schemas (in the form of a DTD or XML Schema) can always be translated into a (local) RG and combined into one global RG to model the data to be managed in the distributed XML repository, thus resembling basic concepts of the view integration process.

A RepositoryGuide is strictly less expressive than a DTD or XML Schema. However, we assume that we can preserve information that might get lost in transforming DTDs or XML Schemas (if these exist) into a tree structure ($\mathbf{V}_{RG}, \mathbf{E}_{RG}$), which represents the RG. In particular, we assume that for each node v in the RG tree, the minimum/maximum number v_{min}/v_{max} of times this node (element type) occurs as child node is recorded. For instance, for a node v , as child of a node v' , the pair (0,2) specifies that v is an optional child element and occurs at most 2 times as child of v' . Obviously, different min/max values can be associated with nodes having the same label in the RepositoryGuide.

Depending on how a RepositoryGuide is constructed, i.e., either from scratch or through “combining” local RepositoryGuides, DTDs, or XML Schemas, further information can be associated with nodes in *RG*. This includes in particular co-occurrence information about pairs of rooted label paths in *RG*. Such co-occurrence information describes that if a document contains a data path corresponding to one label path, then it must also contain a data path corresponding to another label path. Required siblings of a node are the most simple case of such co-occurrences, which can be used in query optimization schemes and in verifying a fragmentation for correctness, which is discussed in Section 3.1.

2.3 Query Model

In our distribution scheme, we assume that a query can be issued at any local site and query results, i.e., complete data fragments, are delivered to that site. Queries consist of path and tree patterns to be matched against the distributed source. Path and tree pattern queries build the core of most XML query languages. Edges in such patterns represent parent-child and ancestor-descendant relationships. Node labels or text values under certain nodes further constrain

the patterns. A distinguished node called selection node determines the roots of fragments to be returned as query result. Figure 1 gives two examples of patterns that have matches in the form of book titles in the data source shown at the center of Figure 3.

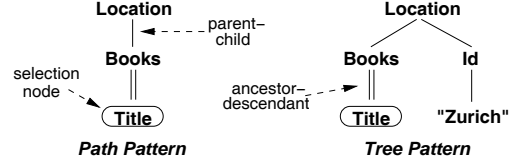


Figure 1: Path and tree pattern queries

The most common approach to process path and tree pattern queries are *structural joins* (e.g., [2, 9, 15]). Structural joins (SJs) are based on index structures that store lists of logical identifiers for nodes in a source. Each list is related to a certain node predicate, which is a common node label or label path, or a word occurrence. Node identifiers allow a query engine to determine parent-child and ancestor-descendant relationships between nodes. This in turn allows for reassembling instances of the full pattern as they occur in a source by joining node id lists, which represent atomic patterns [2].

3. DISTRIBUTION DESIGN

The objective of the distribution design for an XML repository is to formally describe what and how XML repository data is fragmented and allocated at local sites. Although the design strategy follows basic distribution principles known for relational and object-oriented databases [8, 12], the hierarchical structure and the mix of schema information and data in XML requires substantial adjustments to respective design approaches. In the following, we introduce basic properties of fragmentation approaches, illustrate a specific fragmentation scheme, and outline how fragments are allocated at local sites.

3.1 Fragmentation Alternatives

A fragmentation scheme has to satisfy certain correctness criteria in order to ensure that the semantics of the data to be fragmented does not change once fragment data is distributed and managed at local sites. In the context of relational databases, these criteria are known as *completeness*, *reconstruction*, and *disjointness rules* [12]. Similar rules can be devised for fragmenting XML data, as illustrated below.

We assume a set $\mathbf{S} = \{S_1, \dots, S_r\}$ of local sites. With each site S_i , a set of queries Q_{S_i} is associated from which typical access patterns in the form of a sublanguage of XPath can be derived. Such access patterns include path and tree patterns as they are used in local, site-specific queries formulated in XQuery, XPath, or XSLT. The RepositoryGuide representing a global schema not only serves as basis for such queries, but it is also used to specify fragments. Informally, a fragmentation scheme has to ensure that the RepositoryGuide is decomposed into a disjoint and complete set of tree-structured fragments. The allocation scheme applied to respective fragments then will ensure the reconstruction of the repository as outlined in Section 3.2.

The specification of XML fragments is based on a sublanguage of XPath, called XF , that includes the context node ($.$), descendant and child axis ($//$, $/$), and wildcard ($*$). A fragment specification $f = sf - \{ef_1, \dots, ef_n\}$ consists of two components: (1) a *selection fragment* sf , and (2) an optional set $\{ef_1, \dots, ef_n\}$ of *exclusion fragments*. Both types of fragments are XF expressions. The semantics of such fragment specifications is fairly straightforward: first, the selection fragment sf is evaluated against the RG and results in a set of nodes. The evaluation can be done efficiently since an RG (as single XML document) is much smaller than a source corresponding to the RG. Each such node determines a subtree in RG, called *RG fragment*. Optional specifications $\{ef_1, \dots, ef_n\}$ of exclusion fragments are evaluated on such subtrees and again determine RG fragments. For example, assume a fragment specification $//A//C - \{./D/* / E\}$. First, all subtrees in RG are selected that are rooted at a C node and have an A node as ancestor; such subtrees describe selection fragments. From these fragments, all subtrees are “cut off” that are rooted at an E node, have a D node as grandparent, which is child of a C node selected by the fs expression. Figure 2 illustrates this case where fragment f_2 is excluded from fragment f_1 .

Since XF expressions represent path expressions, subtree containment can easily be verified for two fragment specifications based on prefixes of rooted label paths. Let $desc(f)$ denote the nodes in RG that are contained in fragment f (excluding those nodes specified by optional ef 's). Given a set $\mathbf{F} = \{f_1, f_2, \dots, f_l\}$ of fragment specifications. \mathbf{F} is said to be *complete*, if all nodes \mathbf{V}_{RG} are contained in at least one fragment, i.e., $\mathbf{V}_{RG} = \bigcup_{i=1}^l desc(f_i)$. The elements in \mathbf{F} are said to be *disjoint* if there are no two fragment specifications $f_i, f_j \in \mathbf{F}, i \neq j$, such that $desc(f_i) \cap desc(f_j) \neq \emptyset$. For a given set \mathbf{F}' of fragment specifications, disjointness can be verified in a naive fashion by marking each node v in RG with the number i of the fragment specification f_i that contains v . No node then must have more than one number assigned to it. If \mathbf{F}' is not complete, a complete set \mathbf{F} can be obtained by identifying the rooted label paths of subtrees that have not been marked (including exclusions that already have been marked). From a practical point of view, it is very likely that local sites are interested in fragments close to the leaf nodes since such fragments contain most of the data. The “upper” part of an RG, including the root node (see fragment f in Figure 2) then provides a hook for fragments located in the lower portion of the RG.

Before we illustrate the allocation approach for a complete and disjoint set of fragment specifications, it is important to note that the language underlying fragment specifications can easily be extended to include branching ($[]$) and/or conditions on attribute and text values. Thus far, the language XF supports a *vertical fragmentation* approach based on schema structures since all rooted label paths to fragments are disjoint (and common prefixes of label paths will serve as “join attributes” for reconstruction later). The addition of branching and conditions on text/attribute values naturally leads to a *horizontal fragmentation* approach since then the root nodes of two fragments are allowed to have the same label path. Adding branching, however, can lead to some undesirable features regarding checks for disjointness and completeness. For example, assume two fragment

specifications $f_1 = /A/B[C]/G, f_2 = /A/B[D]/G$, none of them containing exclusion fragments. If it is known that every B node always has a C and a D node as children (e.g., based on the co-occurrence information in the RG, see Section 2.3), then these two specifications are not disjoint. Naturally, the more complex branching expressions are used in selection and exclusion fragments, the more complex the decision problem regarding disjointness becomes. We are currently studying such types of language extension in the context of more expressive fragmentation schemes, utilizing recent results on query containment (in the presence of schema information), e.g., [10, 11].

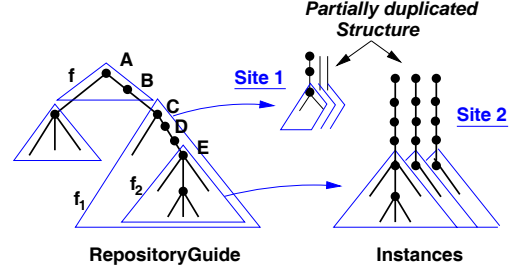


Figure 2: Fragmentation of a RepositoryGuide (1) and distribution of fragment instances to sites (r)

The inclusion of conditions on attributes and text nodes can be dealt with in a fashion analogous to horizontal fragmentation in relational databases. For example, for a fragment specification $f = /A/B[C < 12]$, one can easily derive the complement fragment $f' = /A/B[C \geq 12]$ to ensure completeness.

3.2 Fragment Allocation

The allocation based on a set of complete and disjoint fragment specifications \mathbf{F} involves (1) determining which fragments to allocate at which sites, (2) placing schema structures at local sites, and (3) placing suitable instances of fragments at local sites. In the following, we will outline our approach underlying steps (2) and (3). Step (1) can be dealt with using existing allocation models for relational databases [3, 12]. In the following, we assume that with each site $S \in \mathbf{S}$, a set $\mathbf{F}_S = \{f_1, f_2, \dots, f_s\}$ of fragment specifications is associated. Although we assume that a fragment $f \in \mathbf{F}$ can be replicated at several sites, for each fragment, there is exactly one master site.

Local Schema Structures. Recall that a RepositoryGuide (RG) serves as a global conceptual schema in the proposed distribution approach. To support the processing of global queries at local sites, the RG is fully distributed to each site and extended in the following way. For each node $v \in \mathbf{V}_{RG}$, the site that stores the fragment is recorded. In case of replicated fragments, the master site and replicating sites are recorded. We call such an extension of the RG a *Distribution RG (DRG)*.

Placing Fragments on Sites. Fragments at a site consist of local structure and text content according to the DRG's fragment specification. In addition, the *global context* of each fragment is kept in the form of the data path from the global root node to a local fragment's root (s. Figure

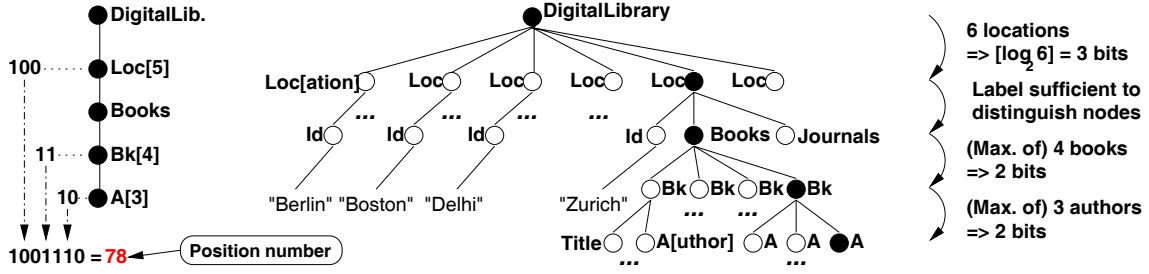


Figure 3: Encoding of data path /DigitalLibrary/Loc[5]/Books/Bk[4]/A[3] in example source

2). This global context serves as “join attribute” for the reconstruction of the full source from distributed fragments.

Adding new data to a virtual XML repository involves the following steps, which are independent of the site where the operation is issued. First, the DRG has to be consulted to determine the fragment f in the DRG that contains the root node of the inserted data. Next, the add operation is sent to the site S that holds the master copy for f . At site S , the operation is executed and at the same time propagated to replicating sites using proper protocols to ensure consistency among all sites. If parts of the added data reach down into another distribution fragment f' below f in the DRG, the master site for f also has to propagate the add operation with content related to f' to the master site for f' . There, the same procedure as described above is applied.

For instance, in the example in Figure 2, if a new C-node with all its related content is added, the master sites for f_1 and f_2 have to be involved. However, notice that because all fragments on local sites replicate the global context path, it is not necessary to involve sites related to distribution fragments higher in the DRG tree (f in the example) when placing new fragments. Furthermore, global context data paths that already exist at a site suffice to compute the global context path for newly added data. Update operations other than insertions have to be propagated in a fashion analogous to placing new data.

4. REALIZATION

The efficiency of our distribution scheme is based on duplicating global path information that leads to XML fragments at local sites. With the global context available locally, it is possible to (i) answer (most) global queries locally, and (ii) easily reconstruct fragments that are distributed over multiple sites. However, storing the global context for every local fragment is potentially expensive. We address this issue by utilizing a new node identification scheme called μ PIDs (“micro-Path IDs”) [4] that effectively encodes rooted data paths and thus global context. Furthermore, we employ a dense storage of these node ids within path and term index structures. We introduce the node id scheme and index structures in Sections 4.1 and 4.2, respectively. In Section 4.3 we outline a distributed query processing approach. In Section 4.4, we evaluate our approach experimentally.

4.1 μ PID Node Identification Scheme

μ PID node ids as used in our approach consist of a pair of integers. The first integer, called *node number*, is an identifier

for a rooted label path as found in the Distribution RepositoryGuide (DRG). The second number is called *position number* and contains all information to identify a particular instance of the label path in a (distributed) source.

Figure 3 shows the construction of the position number for the data path /DigitalLibrary/Loc[5]/Books/Bk[4]/A[3] in the source shown at the center of the figure (the data path is highlighted by black nodes). The position number (bit-)accurately encodes sibling positions whenever multiple siblings with the same label occur. In the figure, this is the case for the Loc[ation], Bk, and A[uthor] nodes. For node types without such multiple siblings, no explicit sibling number has to be encoded (e.g., for Books in the example). The number of bits required at each step within a data path is determined by the maximum fanout v_{max} (s. Section 2.2) of a certain node type anywhere in the source. In Figure 3, assuming every Books node in the source has at most four Bk children, two bits are sufficient to encode Bk sibling positions.

Maximum fanout information is part of an RG. For a mostly static source that already exists at the time of distribution, the fanout can be derived from the source. Otherwise, the fanout can be derived from estimates and specific schema information that might exist for parts of the repository. This procedure to determine the fanout is also state of the art in other node id schemes [9] that, however, are less suitable to support distributed storage and querying. Furthermore, our empirical studies on real-world data sources show that most node types vary only little in their number of siblings, and therefore our encoding of node ids is quite effective (s. Section 4.4 and [4]).

Position numbers are constructed by appending bits related to single sibling numbers within a rooted data path. Therefore, all position numbers related to a certain node number, i.e., rooted label path, have the same bit length. Hence, they can be interpreted as k -bit integers, as shown on the left side of Figure 3.

A node’s μ PID allows for directly deriving the node’s parent node, ancestors, and preceding siblings. Other node relationships need to be derived as in related approaches through structural joins. For this, relationships between two nodes can simply be determined by comparing node numbers and matching position number prefixes. Furthermore, μ PIDs require structural joins only at branch positions within a query pattern rather than for every edge.

Source	Size	P-index	A-index	T-index	Index total	Pos#max	Pos#avg
Big10	1243.1	23.1 (1.9%)	51.9 (4.2%)	347.1 (27.9%)	422.1 (34.0%)	41	22.5/24.4
Reuters	1354.0	55.9 (4.1%)	70.2 (5.2%)	518.9 (38.3%)	645.0 (47.6%)	29	25.7/26.9
XMark 1Gb	1118.0	24.2 (2.2%)	63.8 (5.7%)	310.6 (27.8%)	398.6 (35.7%)	29	21.1/19.8
Fin'l Times	564.1	0.9 (0.2%)	10.5 (1.9%)	115.0 (20.4%)	126.4 (22.4%)	21	20.1/18.0
DBLP	127.7	5.1 (4.0%)	11.4 (8.9%)	35.7 (28.0%)	52.2 (40.9%)	26	20.2/19.2
SwissProt	109.5	10.3 (9.4%)	10.0 (9.1%)	25.8 (23.6%)	46.1 (42.1%)	30	21.5/22.8

Table 1: Source and index size in Mb (% of source size), and pos# length in bits for some XML sources

4.2 Index Structures

Core design. Our distribution scheme employs the same core index structures as other, centralized structural join approaches (e.g., [2, 9, 15]). However, in our approach, lists of μ PIDs are always grouped by node number. Thereby, we avoid having to structurally join node id lists that cannot have matches based on their label path. Furthermore, in a list of μ PIDs, only μ PID position numbers need to be stored repeatedly. In each list, these numbers are of fixed length even though path identifiers in general vary in their length.

In our scheme, a *path index* (P-index) maps node numbers to lists of position numbers in document order. Not all position numbers in the P-index are actually stored. Because of the way μ PIDs are constructed, position numbers in document order are consecutive sequences of (k-bit) integers with some gaps [4]. Therefore, for each consecutive sub-sequences, we store only the first position number and its index position in the full sequence. These index positions have another function as direct pointers into a second index structure. This additional index (*address index* or A-index) maps logical μ PIDs to physical data addresses. Analogously to the path index, a *term index* (T-index) maps terms to lists of μ PIDs for nodes the terms occur in. These lists are grouped by node number, too, but do not utilize the sparse storage structure outlined above.

The μ PID-based P-index and T-index together allow a query engine to process *path* pattern queries without having to reconstruct these patterns from ancestor-descendant relationships through expensive structural join operations. Path patterns are matched against the DRG and result in μ PID lists for node numbers that match the patterns. If a path pattern ends in a term containment condition, the T-index is accessed, otherwise the P-index. Tree patterns have to be resembled from path patterns. For this, lists related to path patterns have to be joined for every branch point in a tree pattern [4]. On the left side, Figure 4 shows core

index structures and access paths as employed for a centralized indexing and query processing approach. The rest of the figure presents the index distribution scheme and is discussed next.

Index distribution. Our distribution scheme distributes nodes in a source by common label paths. Label paths uniquely relate to node numbers in the (D)RG, which relate to well-defined units within all index structures. Therefore, it is straightforward to distribute source fragments together with their accompanying index portions as indicated in Figure 4 for three sites.

However, depending on the size of index structures, it can be beneficial to replicate frequently used index portions at sites that do not keep the related fragments. In particular, we assume that the DRG is fully replicated at each site. Furthermore, P-indexes are remarkably small, as shown in Section 4.4. Therefore, in most cases, fully replicating the P-index is possible at little overhead. For the T-index, duplicating at least parts related to frequently used terms can be cheap and very effective in speeding up query processing. A-index portions, which provide for a translation from logical node ids to physical addresses as the last step in query processing, only need to be kept with their related source fragments at a single (or replicating) site(s).

4.3 Distributed Query Processing

Queries can be processed as follows. A query tree pattern is matched against the local DRG to determine potential matches based on label paths. Instances of these pattern matches then can be stitched together based on the locally available P-index. Only for term conditions whose related T-index portions are not locally replicated, a remote site has to be accessed. Information about sites providing the missing T-index parts can directly be found in the DRG.

For distributed query processing, the most prominent cost factors to process pattern queries is the transfer of node id lists to a common site in order to join the lists. Therefore, the main goal of an efficient distribution scheme using structural joins is to minimize transfers of these lists.

For example, in the source in Figure 3, Books-related fragments with their T-index portions might be stored at Site A while all other information is stored at Site B. The tree pattern query in Figure 1 can thus be processed at Site A by first matching the `/Location/Books//Title` pattern locally. Then, only those node ids for `/Location/Id` fragments that contain the term “Zurich” are fetched from Site B and joined with the earlier matches. If the local result is small, another option to process this query would be to send the matches at Site A to Site B, execute the structural join at Site B, and

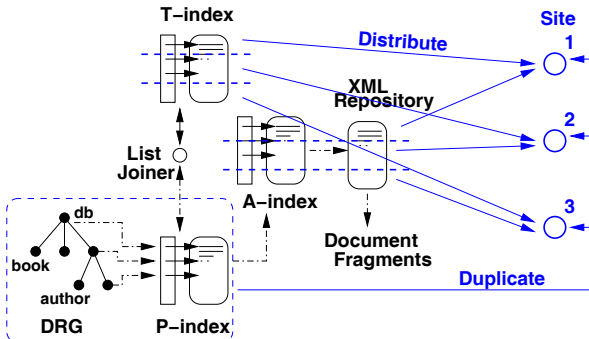


Figure 4: Index structures and their distribution

send the results back. Such optimization options are analogous to optimizations in distributed query processing for relational database and are subject to our current research.

4.4 Evaluation

Our goal in this section is to clearly show that P-index and even parts of the T-index are small enough to be fully replicated, thus reducing distributed query processing to mostly local query processing. Efficiency of local query processing has already been shown [2]. Therefore, even without an extensive, fully distributed prototype system, these observations allow us to conclude that the XML distribution approach we have presented in this paper is realistic and efficient in terms of storage space and query processing.

Table 1 summarizes the sizes of the P-, A-, and T-indexes for different non-distributed sources. Sizes for distributed or replicated index structures can be directly derived from the numbers given in the table. In our implementation, physical addresses as stored in the A-index are offsets into an underlying plain text source. The sources presented in the table are generated by the XMark XML generator,¹ or originate from TREC² disk 4 and 5, Reuters corpus,³ or from the Web.⁴ They have different structural complexity, which is the main reason for variations in relative index sizes. For instance, *Financial Times* has little structure and source depth and thus, very small indexes. *SwissProt* on the other hand is unusually rich in structure and thus, the path index in particular is relatively large. *Big10* is a composite source consisting of nine sources from the aforementioned sites. Table 1 also lists the maximum and average (for P-index on the left and T-index on the right) μ PID position number length. In earlier, equivalent indexing approaches, index size is frequently ignored [2, 9], or reported to be at least four times as large as shown here without providing a mapping of logical node ids to physical addresses [15].

5. CONCLUSIONS AND FUTURE WORK

In this paper, we have presented a complete distribution approach for XML repositories, including fragmentation, allocation, and efficient realization. Fragmentation of XML data and allocation of fragments at local sites build on known concepts from relational databases and are extended to account for tree structured data. In particular, we have identified label path-based fragmentation for XML as analogous to vertical fragmentation for relational system. Path and value conditions provide for an analogy to horizontal fragmentation.

In our realization, the expense of replicating global data paths in order to identify fragments at local sites is offset by an efficient encoding of path information using a new node identification and indexing scheme for XML. Our path index structure in particular is small enough to be fully distributed to local sites and thus bringing the performance of query processing close to that of centralized systems.

As indicated earlier, we are currently investigating more

¹monetdb.cwi.nl/xml/generator.html

²trec.nist.gov

³www.reuters.com/researchandstandards/corpus

⁴www.cs.washington.edu/research/xmldatasets

expressive fragment specifications in our distribution approach (e.g., those similar to derived horizontal fragmentation based on primary/foreign keys). In particular, in the context of data modifications and complex (global) queries, we are studying cost models for the management of (replicated) fragments that span multiple sites.

6. REFERENCES

- [1] S. Abiteboul, A. Bonifati, G. Cobéna, I. Manolescu, T. Milo. Dynamic XML documents with distribution and replication. To appear in *Proc. of the ACM SIGMOD Conf. 2003*, 2003.
- [2] S. Al-Khalifa, H.V. Jagadish, N. Koudas, J.M. Patel, D. Srivastava, Y. Wu. Structural joins: A primitive for efficient XML query pattern matching. In *Proc. of the IEEE Int'l Conf. on Data Engineering*, 141–152, 2002.
- [3] P. M. G. Apers. Data Allocation in Distributed Database Systems. *ACM Transactions on Database Systems (TODS)* 13(3): 263-304 (1988)
- [4] J.-M. Bremer, M. Gertz. An efficient XML node identification and indexing scheme. Technical Report CSE-2003-04, Dept. of Computer Science, University of California at Davis, 2003.
- [5] T. Fiebig, S. Helmer, K.-C. Kanne, G. Moerkotte, J. Neumann, R. Schiele. Anatomy of a native XML base management system. *The VLDB Journal*, (11):292–314, 2002.
- [6] P. Grosso, D. Veillard. XML Fragment Interchange. W3C Candidate Recommendation, W3C, February 2001. www.w3.org/TR/xml-fragment.
- [7] H.V. Jagadish, Laks V.S. Lakshmanan, T. Milo, D. Srivastava, D. Vista. Querying network directories. In *Proc. of the ACM SIGMOD Conf. 1999*, 133–144, 1999.
- [8] K. Karlapalem, Q. Li: Partitioning Schemes for Object-Oriented Databases. In *5th Int'l Workshop on Research Issues in Data Engineering-Distributed Object Management*, 42–49, IEEE, 1995.
- [9] Q. Li, B. Moon. Indexing and querying XML data for regular path expressions. In *Proc. of the 27th Int'l Conf. on Very Large Data Bases (VLDB)*, 361–370, 2001.
- [10] G. Miklau, D. Suciu. Containment and Equivalence for an XPath Fragment. In *21th ACM Symposium on Principles of Database Systems (PODS)*, 65–76, 2002.
- [11] F. Neven, T. Schwentick. XPath Containment in the Presence of Disjunction, DTDs, and Variables. In *9th Int'l Conf. on Database Theory*, LNCS 2572, 315–329, 2003.
- [12] M. T. Özsu, P. Valduriez: *Principles of Distributed Database Systems (2nd ed.)*. Prentice Hall, 1999.
- [13] D. Suciu. Distributed query evaluation on semistructured data. *ACM Transactions on Database Systems (TODS)*, 27(1):1–62, March 2002.
- [14] E. J. Whitehead Jr., M. Wiggins. Webdav: IETF standard for collaborative authoring on the Web. *IEEE Internet Computing*, 34–40, September 1998.
- [15] C. Zhang, J. Naughton, D. DeWitt, Q. Luo, G. Lohman. On supporting containment queries in relational database management systems. In *Proc. of the ACM SIGMOD Conf. 2001*, 425–436, 2001.
- [16] University of Michigan - Timber Project. www.eecs.umich.edu/db/timber.

Index Structures for Querying the Deep Web

Jian Qiu Feng Shao Misha Zatsman¹ Jayavel Shanmugasundaram

Department of Computer Science
Cornell University

{jianq,fshao,jai}@cs.cornell.edu mz36@cornell.edu

1. INTRODUCTION

Most current web search engines can only crawl, index, and query over static web pages, also referred to as the “surface web”. A large fraction of the Internet data, however, is stored in Internet-attached databases or the “deep web”. For example, the data about auctions in ebay.com is stored in an Internet-attached database, but is not visible to current web search engines. Some studies estimate that the size of the deep web is 400-500 times the size of the surface web [1]. Consequently, current web search engine technology can be used to query only a small fraction of all the data available off the Internet.

As part of the Deep Glue project at Cornell University, we are building a query engine for deep web data sources. A key component of the Deep Glue system, and the focus of this paper, is the **indexer module**. Given a user query, the indexer module uses index structures to identify a *superset* of the deep web data sources that are relevant to the user query. The query module (not described here) then evaluates the user query by contacting the potentially relevant data sources identified by the indexer module.

Index structures for the deep web differ in two fundamental respects from traditional inverted list index structures used by current web search engines. First, index structures for the deep web have to deal with *structured data* because the underlying database is typically richly structured and typed; this is in contrast to the mostly unstructured HTML data available off the surface web. Second, index structures for the deep web must deal with data volumes that are *orders of magnitude larger* than that for the surface web [1]. To address the above issues, we devise new index structures for the deep web. The index structures understand the structure/typing of the underlying data, and can thus be used to support both equality and range queries. The index structures can also be heavily compressed so that their space requirements are far less (up to an order of magnitude less) than the size of the original index.

Aggressive compression of our index structures is possible because we allow the indexing module to return a *superset* of the data sources that are relevant to a user query. By

returning a superset of the relevant data sources, no relevant data sources are missed. However, the query module has the extra overhead of contacting some data sources that are not relevant to the query. To evaluate this tradeoff, we present preliminary experimental results over 1000 synthetically generated deep web data sources. Our results are promising and indicate that if we compress the index size by a factor of 10, the number of extra data sources contacted is less than 10 (more details are presented in Section 4). Further, the compression factors in our index structures are tunable, and can be used to tradeoff index size for precision.

The rest of this paper is organized as follows. In Section 2, we describe our system and query model, and in Section 3, we describe various deep web index structures. In Section 4, we experimentally evaluate the performance of the index structures. In Section 5 we discuss related work, and in Section 6, we present our conclusions and outline directions for future research.

2. SYSTEM AND QUERY MODEL

Figure 1 shows the architecture of the Deep Glue system. Given a user query, the query engine contacts the indexer to determine a superset of the data sources that are relevant to the user query. The query engine then evaluates the user query by contacting the relevant data sources. The indexer constructs the deep web index structures offline; this can be done either by crawling deep web data sources [13] or by using some previously agreed upon protocol for transferring index data [4]. This paper focuses on the organization of the deep web index structures once all the indexing data is obtained from the deep web data sources.

For the purposes of this paper, we make the following assumptions. We assume that the deep web data sources are pre-classified into a set of domains such as online car dealers, online auctions, and online travel agents. We further assume that all the deep web data sources within the same domain export their data using the same logical database schema (the mapping from different physical database schemas to the same logical database schema could be done using mediators [16]). For simplicity, we also assume that the database schema conforms to the relational data model. In the relational schema, a subset of the attributes are referred to as the indexing attributes;

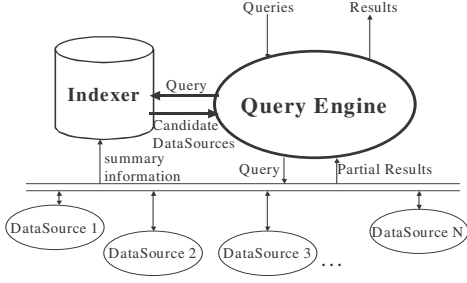


Figure 1: System Architecture

these could be attributes such as price, date, make, model, isbn number, etc. The values of the indexing attributes are indexed by the Deep Glue system, and are used to direct queries to the relevant data sources.

In this paper, we focus on equality and range selection queries on a single indexing attribute. For example, in the domain of online car dealers with relational schema Car(Id, Make, Model, Year, Price), we can answer queries such as “find all year 2003 cars (Year = 2003)” or “find all cars that cost less than \$1000 (Price < 1000)”, where date and price are each indexing attributes. We do not consider multiple attribute selection queries or join queries in this paper, and this is a subject for future research.

3. INDEX STRUCTURES

We now turn to the main focus of this paper, which is devising space-efficient index structures for evaluating equality and range queries over deep web data sources. We first present a simple uncompressed index structure, and then present various techniques for compressing this index structure. Please note that all the compression techniques explored in this paper are complementary to existing database and inverted list compression techniques such as [2][5][10][11][14][15]. Such compression techniques can be applied on top of our indices to achieve further compression.

3.1 Uncompressed Index (UI)

An Uncompressed Index (UI) can be created as follows. For each distinct value v of the indexing attribute, we can store the list of data sources that contain at least one tuple that has the value v for the indexing attribute. A small example of such an index is shown in Figure 2. Data sources d6, d7, and d8 have at least one tuple that has value 1 for the indexing attribute, data sources d2, d3, d4, and d6 have at least one tuple that has value 2 for the indexing attribute, and so on. Building a B+-tree index on the value column will enable equality and range queries to be efficiently handled. For handling equality queries, we can simply lookup the list of databases that contain the desired value. For handling range queries, we can union the set of data sources for all the values in the desired range.

In realistic deep web scenarios, the number of distinct values of the indexing attribute and the number of data sources are both likely to be large. Since the index structure

value	data sources
1	d6,d7,d8
2	d2,d3,d4,d6
3	d4
4	d1,d4,d5
5	d2,d3,d6
6	d1,d7,d8

Figure 2: Uncompressed Index

value	Cluster id	Cluster id	data sources
1	c1	c1	d1,d6,d7,d8
2	c2		
3	c3	c2	d2,d3,d4,d6
4	c3		
5	c2	c3	d1,d4,d5
6	c1		

Figure 3: Value Clustered Index

has to be built for *each* indexing attribute, the overall size of the index is also likely to be very large, and in the worst case, could approach the entire size of the indexed deep web. Consequently, we need to explore techniques for effectively compressing the index structure.

Note that we cannot simply use a compression utility such as gzip to compress each index structure. If we used gzip to compress the index, then the entire index would have to be uncompressed for *every query* over the indexing attribute, which would make the index lookup time very expensive.

We can, however, use database compression techniques that compress individual fields of data [2][5][11][15]. We use such techniques, specifically [11], for compressing the lists of data sources. We could also use inverted list compression techniques such as [10][14]. However, we go significantly beyond this and provide an *additional* compression factor of 10 in space while still efficiently supporting equality and range queries.

3.2 Value Clustered Index (VCI)

The key idea behind the Value Clustered Index (VCI) is the following. In UI, the lists of data sources associated with different values are often similar (although not necessarily the same). This captures the intuition that “closely related” values are often stored in “closely related” data sources. For example, if the indexing attribute is the ISBN number of a book, then the ISBN numbers of out-of-print books are likely to have a very similar list of data sources, consisting mostly of online retailers specializing in out-of-print books. In the small example in Figure 2, values 1 and 6 have a similar list of data sources, as do values 2 and 5, and values 3 and 4.

Given that different values can have a similar list of data sources, we can save space by “grouping” or “clustering” such values together, and storing the associated list of data sources just once for each cluster (instead of repeating the list for each value in the cluster as in UI). As an illustration, consider the UI in Figure 2. Assume that the values are divided into three clusters, with cluster c1 containing values 1 and 6, cluster c2 containing values 2 and 5, and cluster c3 containing values 3 and 4. The resulting VCI is shown in Figure 3. Note that each value has a cluster id associated with it. Each cluster id in turn is associated with

value	Cluster id
1	c1,c3
2	c1,c2
3	c2
4	c2,c3
5	c1
6	c3

Cluster id	data sources
c1	d2,d3,d6
c2	d4,d5
c3	d1,d7,d8

Figure 4: DataSource Clustered

a list of data sources. Thus, the list of data sources associated with a value can be determined by first looking up the cluster id for the value, and then looking up the list of databases associated with the cluster id (in reality, cluster id can be a physical disk pointer making the look up for the list of databases very efficient). A B+-tree index on the value will again enable equality and range queries.

It is important to note that in VCI, the list of data sources associated with each cluster is the *union* of the list of data sources associated with each value in the cluster. For example, for cluster c1, the list of data sources (d1,d6,d7,d8) is the union of the list of data sources associated with value 1 (d6,d7,d8) and value 6 (d1,d7,d8). By using the union, we ensure that no relevant data sources are missed for a query over a given value. However, some data sources that are not relevant to a query can also be returned. For example, a query for value 1 will have to contact d1 even though d1 does not contain that index attribute value. VCI attempts to minimize such “false positives” by clustering together only values that have a similar list of data sources.

Another interesting feature to note about VCI is that it allows space to be traded off for precision. At one extreme, if each distinct value is mapped to a separate cluster, VCI is the same as UI, and there are no false positives. At the other extreme, all values are mapped to the same cluster resulting in a very small index, but every data source will have to be contacted for every query resulting in a large number of false positives. In practice, a space-precision tradeoff between the two extremes will likely be most desirable, and this can be tuned to the needs of the application.

Thus far, we have described VCI assuming that we have some way of clustering values with similar lists of data sources. Thus, a practical issue is determining an efficient way to do this clustering that (a) scales to large data sets, and (b) attempts to minimize the number of false positives in each cluster. To handle the scalability issue, we rely on existing scalable clustering techniques such as Birch [17]. Birch is a general algorithm for clustering that can handle any clustering domain, as long as the notions of a centroid (the “mid-point” of a cluster), radius (a measure of the quality of a cluster) and cluster distance are defined for that domain. We now define centroid, radius and distance for VCI so that it attempts to minimize the number of false

value	Cluster id
1	c3
2	c1,c2
3	c2
4	c2
5	c1
6	c3

Cluster id	data sources
c1	d2,d3,d6
c2	d1,d4,d5
c3	d1,d6,d7,d8

Figure 5: Value-DataSource Clustered

value	Cluster id
1	c1
2	c2
3	c2
4	c2
5	c3
6	c3

Cluster id	data sources
c1	d6, d7,d8
c2	d1,d2,d3,d4, d5,d6
c3	d1,d2,d3,d6, d7,d8

Figure 6: Histogram Based

positives for queries. We use $ds(v)$ to denote the set of data sources associated with value v . For a cluster having the set of values V :

$$centroid(V) = \bigcup_{v \in V} ds(v)$$

$$radius(V) = \frac{\sum_{v \in V} |centroid(V) - ds(v)|}{|V|}$$

The centroid of a cluster is the union of the data sources associated with the values in the cluster, or simply the data source list associated with a cluster. The radius of a cluster is the sum of the number of false positives for each value in the cluster, normalized by the number of values in the cluster. Thus, as desired, a cluster with small radius is of high quality, while a cluster with large radius is of low quality. The distance between two clusters $V1$ and $V2$ is:

$$distance(V1, V2) = |V1| \times |centroid(V2) - centroid(V1)| + |V2| \times |centroid(V1) - centroid(V2)|$$

The distance between two clusters is the additional number of false positives that would occur if the two clusters were merged together. This is computed as the sum of the following two quantities (a) the number of values in the first cluster multiplied by the number of data sources that only occur in the second cluster, and (b) the number of values in the second cluster multiplied by the number of data sources that only occur in the first cluster.

Using the above definitions of centroid, radius, and distance, we used Birch to generate VCI clusters that attempt to minimize the number of false positives. Further, by setting a threshold on the maximum allowable radius, we can control the space-precision tradeoff by choosing high quality clusters with the associated space overhead, or choosing lower quality clusters and getting more space compression.

3.3 DataSource Clustered Index (DCI)

VCI achieves space compression by clustering values that have similar data source lists. An alternative means of compression is possible by clustering data sources that have similar values. The intuition here is that closely related data sources (such as amazon.com and barnesandnoble.com) often have closely related sets of values (such as book ISBN numbers). Thus, instead of storing these data sources separately in the data source list,

they can be clustered and a single cluster id can be stored in the data source list.

In our example in Figure 2, data sources 1, 7, and 8, have a similar (though not identical) set of values and can be clustered together. Similarly, data sources 2, 3 and 6 have a similar set of values, and so do data sources 4 and 5. The DCI index structure resulting from clustering these related data sources together is shown in Figure 4. Note that a value is associated with a cluster id if *at least* one data source in the cluster contains the value – this is to ensure that no relevant data sources are missed during querying. This also means that DCI could have false positives because if a value is associated with a cluster id, then not every data source in that cluster necessarily contains that value. For example, in Figure 4, the value 3 is associated with cluster c2, and cluster c2 contains data source d5 even though d5 does not contain the value 3.

DCI offers a space-precision tradeoff similar to VCI. If the number of clusters equals the number of data sources, the index structure is identical to UI (with no false positives and associated space overhead); if the number of clusters is 1, then there are many false positives but little space overhead; intermediate tradeoff points can be got by adjusting the number of clusters. Like in VCI, the Birch algorithm is used for clustering databases. The formulae for centroid, radius, and distance are symmetric to VCI (using values in place of data sources and vice versa), and are presented in the Appendix.

3.4 Value-DataSource Clustered Index (VDCI)

Consider a scenario of online book data sources, where there is a natural clustering of data sources based on the ISBN numbers of the books the data sources carry (the ISBN numbers are in the categories popular books, out of print books, collector’s books, etc.). Now consider a data source that carries both popular books and out of print books. Under both VCI and DCI, the data source will have to be clustered along with *either* the popular books cluster, *or* the out of print books cluster. There is no way to specify that the data source is part of one cluster for popular books, and another cluster for out of print books. VDCI attempts to address this limitation of VCI and DCI. In VDCI, a cluster is a set of values V and a set of data sources D , and implies that the data sources D are related with respect to the values V (or equivalently, the values V are related with respect to the data sources D). Thus, VDCI generalizes both VCI and DCI.

As an illustration of VDCI, consider again the example in Figure 2. Consider the data sources d2, d3, and d6. These data sources are similar in that they all contain values 2 and 5. Thus, VDCI allows the cluster $(\{2,5\},\{d2,d3,d6\})$ to be formed. In addition, d6 can also be part of the cluster with d1, d7 and d8 with respect to the values 1, resulting in the cluster $(\{1,6\},\{d1,d6,d7,d8\})$. Let us also assume that the cluster $(\{2,3,4\},\{d1,d4,d5\})$ is created. Then the resulting VDCI index structure is shown in Figure 5.

Since VDCI clusters with respect to each value-data source combination, a cluster is defined as a set of (value, data source) pairs. The definition of the centroid, radius and distance is a combination of the formulae for VCI and DCI and is presented in the Appendix.

Although VDCI generalizes both VCI and DCI, it suffers from the drawback that it needs to cluster each (value, data source) pair separately in order to find the right clusters. This is in contrast to VCI/DCI, which only need to cluster each value/data source separately.

3.5 Histogram Based Index (HBI)

VCI and VDCI do not consider the ordering among values when constructing clusters. Consequently, two adjacent values in the value space are no more likely to be clustered together (based on the data source list) than distant values in the value space. However, since range queries select adjacent values, it may be beneficial to cluster adjacent values together. HBI is based on this intuition, and clusters adjacent values into the same cluster. This is similar to grouping attribute values into “buckets” in histograms in relational systems [12]. Figure 6 shows the HBI structure for the UI in Figure 2 where value 1 is in one cluster, values 2, 3 and 4 are in a second cluster, and values 5 and 6 are in a third cluster.

Since HBI is limited to clustering only adjacent values in the same cluster, it loses some of the flexibility of VCI and VDCI. To minimize the effects of this loss of flexibility, it is important to determine the cluster boundaries effectively. For this, we introduce a threshold parameter for constructing histograms. All the values for the indexing attribute are sorted first and inserted into the clusters (histogram buckets) in ascending order. Whenever the inclusion of a new value causes average number of false positives of the current cluster to exceed the threshold, we create a new cluster for the new value. Thus we can control the index size by specifying the proper threshold for clusters.

4. EXPERIMENTAL EVALUATION

We now experimentally evaluate our index structures. There are three metrics that we use in our evaluation. The first is *index creation time*. The second is *compression factor*, which is the ratio of the size of the uncompressed index (UI) to the size of the compressed index. The third is *false positives*, which is the average number of false positives for equality/range queries.

4.1 Experimental Setup

We use synthetic data for our evaluation. We chose to use synthetic data for the following reasons. First, we can vary parameters to illustrate the tradeoffs between the different approaches. Second, we were not able to easily obtain real data from hundreds/thousands of deep web data sources, although this is something we are currently pursuing.

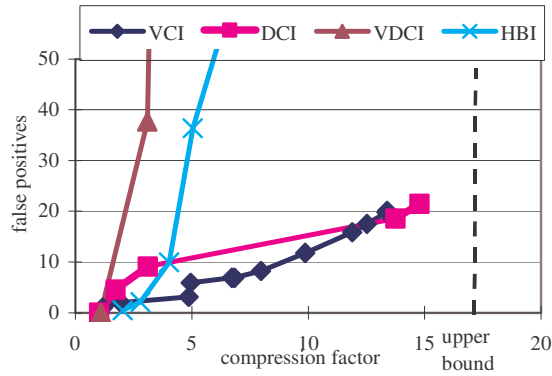


Figure 7 Equality Queries

The synthetic data set was generated using the following parameters. The **number of values** is the number of distinct values for the indexing attributes (default value is 100,000). The **number of data sources** is the number of deep web data sources (default is 1000). The **number of pairs** is the number of unique (value, data source) pairs (default is 4,000,000). This is a measure of the data size, but is usually smaller than the number of tuples because duplicate values in the same data source are only counted once.

The **number of groups** is the number of logical groups among the data sources (default is 20). A logical group captures the notion that a set of data sources can have a similar distribution of values for the indexing attribute. Thus, the data values for each data source in a group are picked from the same Gaussian distribution characteristic of that group. The mean and standard deviation of the Gaussian distribution for a group is chosen so that there is at most a 5% overlap between the distributions of any two distinct groups. The **group mode** is the number of logical groups that each data source belongs to (default is 1). The group mode captures the intuition that a given data source can be similar to one set of data sources with respect to one set of values (such as popular books), and similar to another set of data sources with respect to another set of values (such as out-of-print books).

Finally, the **value correlation** is a measure of how the ordering in the value space maps to the ordering of values over which Gaussians are defined – this is especially important for range queries. The value correlation is defined in terms of the normalized number of permutations to go from the value space ordering to the Gaussian ordering. A value correlation of 1 implies full correlation, while a value correlation of 0 implies no correlation (default is 0.2). Using the default settings, the size of the uncompressed index (UI) is 8MB.

False positives for equality queries are measured by averaging the false positives for all distinct values. False positives for range queries are measured by averaging the false positives for all possible range queries of a given size. The **size of range** specifies the number of distinct values selected by the range query (default is 500).

We implemented all the index structures using C++, and our experiments were performed using a 2.8 GHz Pentium

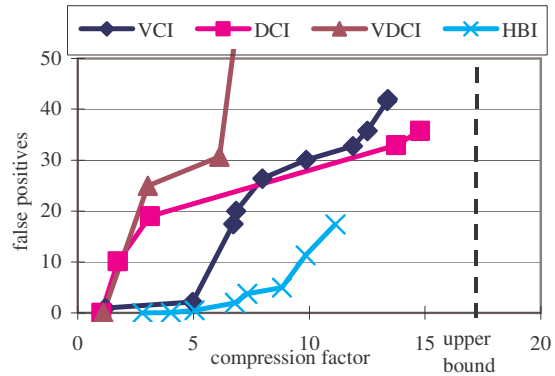


Figure 8 Range Queries

IV processor running Windows XP, with 1GB of main memory and 80GB of disk space.

4.2 Experimental Results

We now present some of our experimental results (more results are presented in the Appendix).

4.2.1 Index creation time

The index creation time for UI using default parameter values was 15 seconds. The index creation time for compressed index structures using the default parameter values and a compression factor of 10 was: VCI – 15 minutes, DCI – 3 minutes, VDCI – 3 hours, HBI – 2.5 minutes. VDCI has a much larger index creation time because it clusters (value, data source) pairs, as opposed to just the values (VDCI and HBI) or just the data sources (DCI). Since the number of (value, data source) pairs is much larger than the number of values or data sources, the clustering time for VDCI increases correspondingly. Although clustering is done offline, this overhead for VDCI could potentially become a bottleneck.

4.2.2 Compression factor and false positives

Figure 7 shows how the number of false positives varies with the compression factor for equality queries, assuming default parameter values. The standard deviation is less than 10, and is not shown. VCI performs the best, with only 10 false positives for a compression factor of 10. DCI performs slightly worse than VCI – this is because there are fewer data sources than there are values, and thus there is less flexibility in forming clusters in DCI. The false positives for HBI increase rapidly with compression because HBI only clusters adjacent values, even if they have very different data source lists.

The most surprising result is the bad performance of VDCI, even though it theoretically generalizes both VCI and DCI. VDCI performs badly for the following reasons. First, VDCI has to deal with a much larger number of data points, which increases the chance of creating bad clusters using scalable one or two pass clustering algorithms. Second, VDCI appears to have too many degrees of freedom with respect to choosing sets of values and data sources. We experimented with different distance functions for VDCI clustering, but could not overcome this fundamental problem. Although we are still exploring other ways to solve this issue, this coupled with the clustering time argues against using VDCI.

Note that false positives increase rapidly for all indices beyond a compression factor of 10. This increase is because the indices are approaching their theoretical upper bound for compression. This upper bound comes about because each index has to store the mapping from the value to the cluster id, for each value. The size of this mapping is thus the upper bound for compression, which is 480KB for the default settings.

Figure 8 shows the space-precision tradeoff for range queries. The results are similar to equality queries, except that HBI now outperforms even VCI. The good performance of HBI is attributable to the fact that it chooses clusters based on the value ordering. Since range queries go over the same value ordering, this reduces the number of false positives.

5. RELATED WORK

This work is related to the body of work on distributed databases [8] and information integration [3][7]. However, our focus is on *identifying relevant data sources* efficiently, while prior work has mostly focused on query processing *once the relevant data sources are identified*.

The GLOSS system [6] supports data source discovery for text documents. The Niagara system [9] uses an index structure that identifies a superset of data sources relevant to a user query. However, these systems do not address the issue of compression, and do not support structured queries such as range queries.

There has been work on field-level lossless compression in databases/indices [2][5][11][15] and inverted list [10][17]. Our work builds upon this work in compressing individual fields (see Section 3.1), but goes significantly beyond and achieves up to a factor of 10 further compression by trading off space for a little loss in precision. The relationship of our work to traditional compression techniques such as gzip is described in Section 3.1.

6. CONCLUSION AND FUTURE WORK

We have presented new space-efficient index structures for querying the deep web. The index structures support structured queries such as equality and range queries, and can be compressed by up to a factor of 10. However, this compression implies that extra data sources may have to be contacted during query processing. Our preliminary experimental results indicate that this overhead is minimal even for large compression ratios.

Our index structures also illustrate another tradeoff in handling equality vs. range queries. The VCI index based on clustering significantly outperforms the HBI index based on histograms for equality queries, and vice versa for range queries. We are thus exploring ways to combine the benefits of cluster-based and histogram-based indices so that they are effective for both types of queries.

Other avenues for future work include extensions to multiple attribute queries and joins, incremental index maintenance, using structured vocabularies in the index structure, incorporating the notion of ranking, and evaluating the index structures using real data sets.

7. REFERENCE

- [1] M. Bergman, "The Deep Web: Surfacing Hidden Value", Technical White Paper, <http://www.brightplanet.com/deepcontent/tutorials/DeepWeb>.
- [2] Z. Chen, J. Gehrke, F. Korn, "Query Optimization In Compressed Database Systems", SIGMOD 2001.
- [3] D. Florescu, et al., "Database Techniques for the World Wide Web: A Survey", SIGMOD Record 27(3), 1998.
- [4] Froogle: <http://www.froogle.com>
- [5] J. Goldstein et al, "Compressing Relations and Indexes", ICDE 1998.
- [6] L. Gravano, H. Garcia-Molina, A. Tomasic, "GLOSS: Text-Source Discovery over Internet", TODS 24(2), 1999.
- [7] A. Gupta, V. Harinarayanan, A. Rajaraman, "Virtual Database Technology", ICDE 1998.
- [8] D. Kossmann, "The State of the Art in Distributed Query Processing", ACM Computing Surveys 32(4), 2000.
- [9] J. Naughton et al, "The Niagara Internet Query System", IEEE Data Eng. Bulletin, 24(2), 2001.
- [10] G. Navarro, et al., "Adding Compression to Block Addressing Inverted Indexes", Information Retrieval, 3:49-77, 2000
- [11] P. O'Neill, D. Quass, "Improved Query Performance with Variant Indices", SIGMOD 1997.
- [12] V. Poosala, et al., "Improved Histograms for Selectivity Estimation of Range Predicates", SIGMOD 1996.
- [13] S. Raghavan, H. Garcia-Molina, "Crawling the Hidden Web", VLDB 2001.
- [14] P. Weiss, Size Reduction of Inverted Files Using Data Compression and Data Structure Reorganization. PhD thesis, George Washington University, 1990.
- [15] T. Westmann, et al., "Implementation and Performance of Compressed Databases", SIGMOD Record, 2000.
- [16] G. Wiederhold, "Mediators in the Architecture of Future Information Systems", IEEE Computer 25(3), 1992.
- [17] T. Zhang, et al., "BIRCH: Efficient Data Clustering Method for Very Large Databases", SIGMOD 1999.

APPENDIX A FORMULAE FOR DCI

We present the formulae for *centroid*, *radius*, and *distance* in DCI. Let $dv(d)$ be the set of data values associated with a data source d and let a cluster be a set of data sources D , then

$$\text{centroid}(D) = \bigcup_{d \in D} dv(d)$$

$$\text{radius}(D) = \frac{\sum_{d \in D} |\text{centroid}(D) - dv(d)|}{|D|}$$

The centroid of a cluster is the union of the values associated with the data sources in the cluster. The radius of a cluster is the sum of the difference between the centroid and the values associated with each data source, normalized by the number of data sources in the cluster. The distance between two clusters $D1$ and $D2$ is:

$$\text{distance}(D1, D2) = |D1| \times |\text{centroid}(D2) - \text{centroid}(D1)| + |D2| \times |\text{centroid}(D1) - \text{centroid}(D2)|$$

The distance is computed as the sum of the following two quantities: (a) the number of data sources in the first cluster multiplied by the number of values that only occur in the second cluster, and (b) the number of data sources in the second cluster multiplied by the number of values that only occur in the first cluster.

APPENDIX B FORMULAE FOR VDCI

We present the formulae for *centroid*, *radius*, and *distance* in VDCI. A cluster C in VDCI is a set of (value, data source) pairs, i.e., $C = \{(v, d)\}$ where v is a value and d is a data source. Let

$$D = \bigcup_{(v,d) \in C} \{d\}, V = \bigcup_{(v,d) \in C} \{v\}, ds(v) = \bigcup_{(v,d) \in C} \{d\}$$

In other words, D is the set of data sources in C , V is the set of values in C and $ds(v)$ is the set of data sources associated with value v in C . The *centroid* of C is,

$$\text{centroid}(C) = (V, D)$$

$$\text{radius}(C) = \frac{\sum_{v \in V} |D - ds(v)|}{|V|}$$

The centroid of a cluster is a pair in which the first component is the set of values in the cluster and the second component is the set of data sources in the cluster. The radius of a cluster is the sum of the number of false positives for each value in the cluster, normalized by the number of values in the cluster. The distance between two clusters $C1$ and $C2$ is:

$$\text{distance}(C1, C2) = |V1 - V2| \times |D2 - D1| + |V2 - V1| \times |D1 - D2|$$

where $\text{centroid}(C1) = (V1, D1)$ and $\text{centroid}(C2) = (V2, D2)$

The distance between two clusters is the additional number of false positives that would occur if the two clusters were merged together. This is computed as the sum of the following two quantities (a) the number of values that only occur in the first cluster multiplied by the number of data sources that only occur in the second cluster, and (b) the number of values that only occur in the second cluster multiplied by the number of data sources that only occur in the first cluster. We do not take the

common values of the two clusters into account since the number of false positives associated with them is not an additional quantity.

APPENDIX C ADDITIONAL EXPERIMENTAL RESULTS

We now present additional experimental results obtained by varying the *number of data sources*, *number of pairs*, *size of range* and *value correlation* parameters. For each experiment, we varied one parameter and used default values for the rest. We set the compression factor to be 10 for these experiments. We do not show the performance numbers for VDCI in these experiments because its performance is consistently bad.

Figure 9 shows how the number of false positives varies with the number of data sources for equality queries. The number of false positives increases gradually with the number of data sources for VCI and DCI. The number of false positives for HBI, however, increases more rapidly because the data source lists of adjacent values become increasingly different as the number of data sources increases. Figure 10 shows how the number of false positives varies with the number of data sources for range queries. The trend is similar to Figure 9 for VCI and DCI. HBI, on the other hand, performs much better for range queries because adjacent values are clustered together.

Figure 11 shows the results when the number of pairs is varied for equality queries. For this graph, the index size was fixed to be 800KB. The graph shows that the false positives for all three approaches increase with the number of pairs. This is attributable to the fact that each value now occurs in a larger number of data sources. Figure 12 shows a similar graph for range queries. In this graph, however, the number of false positives decreases when the number of pairs increases. This can be explained by the fact that the data source lists for adjacent values become more similar because they each contain more data sources.

Figure 13 shows the results when the size of range is varied. For small range sizes, the number of false positives for HBI increases rapidly because its performance approaches that for equality queries (equality queries are simply range queries with a range size of 1). However, for relatively large range sizes, HBI performs well as expected. The false positives for VCI and DCI increases when the range size increases from 1 to 100; this is due to the fact that more than one cluster in VCI/DCI has to be contacted for a range query, and the error in each cluster adds to the overall error. The number of false positives decreases beyond a range size of 100 because the number of data sources containing values in the selected range increases faster than the number of newly selected clusters, thereby reducing the total number of false positives.

Figure 14 shows the results when the value correlation is varied for equality queries. VCI and DCI perform consistently well since they do not consider value ordering when constructing the clusters. In contrast, the number of false positives of HBI increases rapidly when the value correlation decreases – this is because HBI has to group adjacent values in the same cluster, and when the correlation decreases, adjacent values are likely to have increasingly dissimilar lists of data sources. We do not show the effects of varying value correlation for range queries.

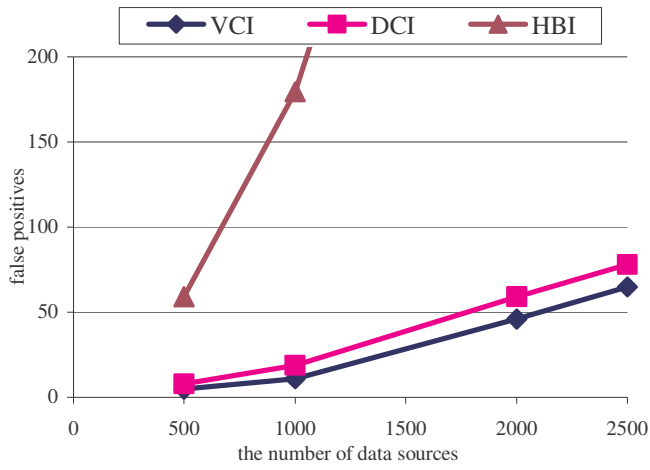


Figure 9 Effects of the number of data sources on equality queries

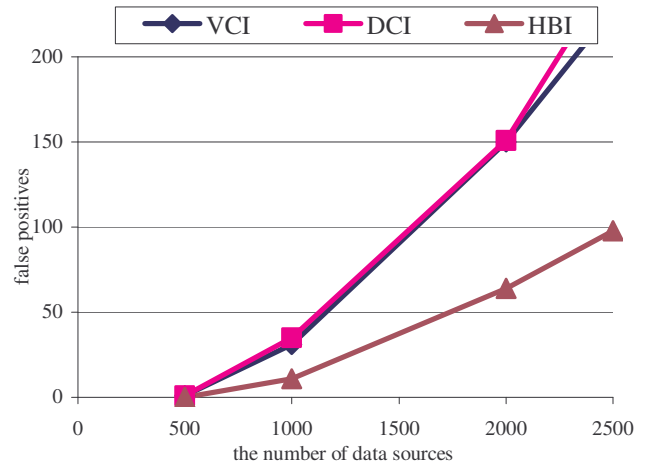


Figure 10 Effects of the number of data sources on range queries

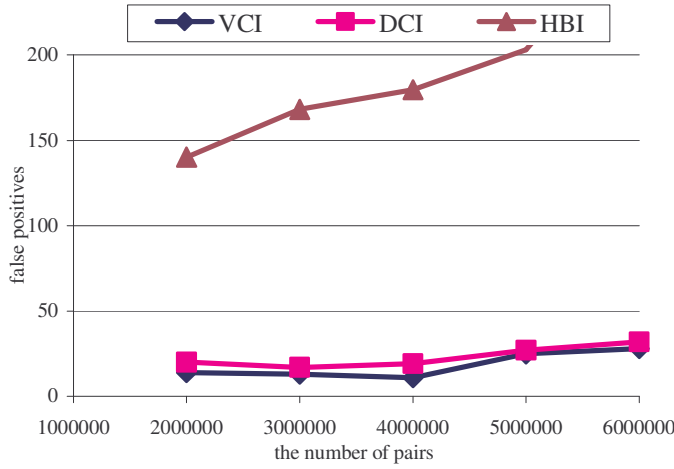


Figure 11 Effects of the number of pairs on equality queries

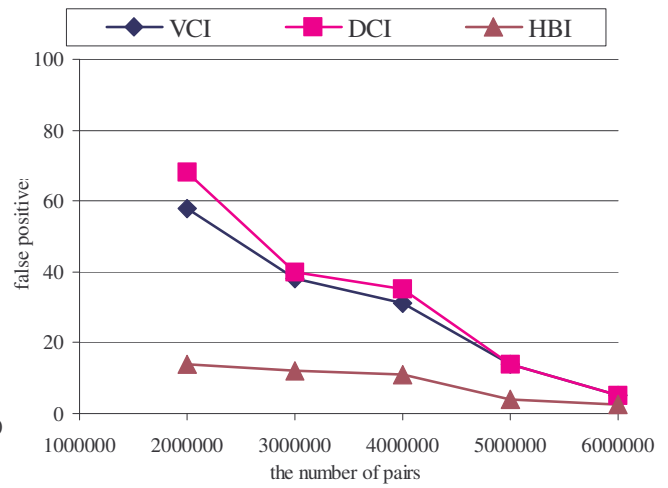


Figure 12 Effects of the number of pairs on range queries

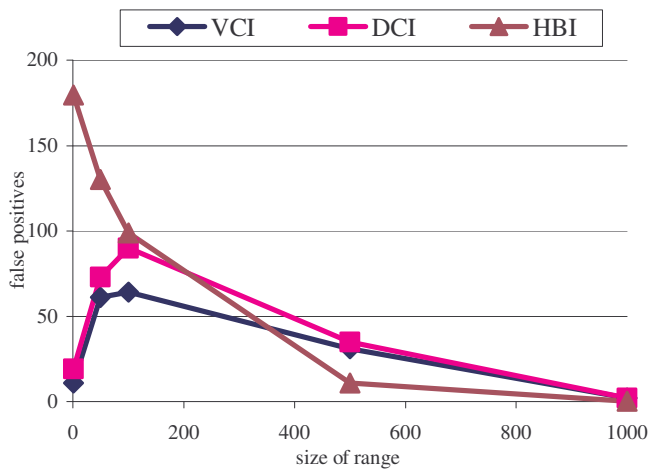


Figure 13 Effects of size of range on range queries

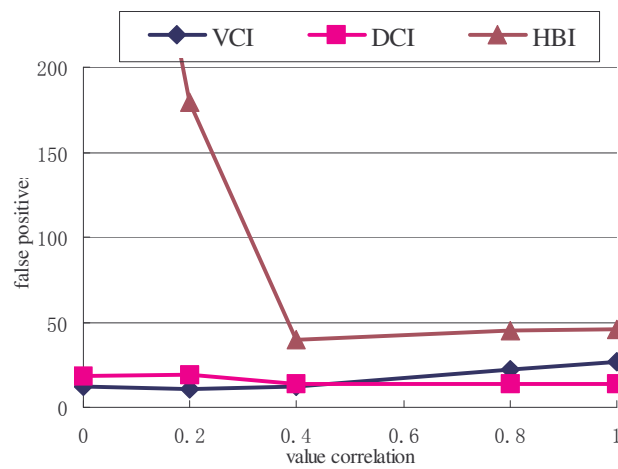


Figure 14 Effects of value correlation on equality queries

Modeling Query-Based Access to Text Databases

Eugene Agichtein Panagiotis Ipeirotis Luis Gravano

Columbia University

{eugene,pirot,gravano}@cs.columbia.edu

ABSTRACT

Searchable text databases abound on the web. Applications that require access to such databases often resort to querying to extract relevant documents because of two main reasons. First, some text databases on the web are not “crawlable,” and hence the only way to retrieve their documents is via querying. Second, applications often require only a small fraction of a database’s contents, so retrieving relevant documents via querying is an attractive choice from an efficiency viewpoint, even for crawlable databases. Often an application’s query-based strategy starts with a small number of user-provided queries. Then, new queries are extracted—in an application-dependent way—from the documents in the initial query results, and the process iterates. The success of this common type of strategy relies on retrieved documents “contributing” new queries. If new documents fail to produce new queries, then the process might stall before all relevant documents are retrieved. In this paper, we develop a graph-based “reachability” metric that allows to characterize when an application’s query-based strategy will successfully “reach” all documents that the application needs. We complement our metric with an efficient sampling-based technique that accurately estimates the reachability associated with a text database and an application’s query-based strategy. We report preliminary experiments backing the usefulness of our metric and the accuracy of the associated estimation technique over real text databases and for two applications.

1. INTRODUCTION

Searchable text databases abound on the web. Applications that require access to such databases often resort to querying to extract relevant documents because of two main reasons. First, some text databases on the web are not “crawlable,” and hence the only way to retrieve their documents is via querying. Second, applications often require only a small fraction of a database’s contents, so retrieving relevant documents via querying is an attractive choice from an efficiency viewpoint, even for crawlable databases. Various query-based methods (e.g., [4, 1]) have been proposed in the past for retrieving and extracting the information stored in databases via querying. These algorithms share the general approach of starting with a small set of queries, retrieving some documents from the database, extracting some information from them, and potentially augmenting the set of queries using the newly extracted information.

More specifically, there are two tasks that have been successfully addressed through querying: extracting information from text databases, and building text database summaries.

Task 1: Information Extraction: This is the task of extracting structured relations from unstructured (i.e., natural language) text.

The structured relations can be used for answering queries or for data mining tasks. For example, a user may be interested in automatically building a structured table of reported disease outbreaks *DiseaseOutbreaks(diseaseName, country, date)* from news articles. Unfortunately, exhaustively scanning every news article for potential events of interest is (unnecessarily) slow. It has been shown that querying can be used to significantly improve the efficiency of information extraction [1]. The goal is to extract all tuples for a target relation from a database, starting by using a few seed tuples as queries and then successively querying for each newly extracted tuple. Intuitively, the documents that contain a query tuple are expected to contain other previously unseen tuples as well:

- (0) while *seed* has an unprocessed tuple *t*
- (1) retrieve up to *MaxResults* documents matching *t*
- (2) extract new tuples *t_e* from these documents
- (3) augment *seed* with *t_e*

Thus, new tuples are discovered by querying for the known tuples, and when the algorithm terminates, all of the tuples “reachable” from the *seed* tuples will be discovered. In [1] it was observed that while in some cases this simple “bootstrapping” strategy—referred to as *Tuples* in [1]¹—succeeded in reaching most of the useful documents in the database (and thereby in extracting most of the tuples in the target relation), in other cases this algorithm only discovered a small fraction of the available tuples. As another example, Shah [10] uses a query strategy related to *Tuples* to extract people’s names and build networks of “experts.” In this work we identify and analyze the intrinsic property of text databases with respect to an extraction task that determines the success of this general approach. But first, we present a related task that can also be modeled using our techniques.

Task 2: Text Database Summary Construction: Many valuable text databases on the web have non-crawlable contents that are “hidden” behind search interfaces. Metasearchers are helpful tools for searching over many such databases at once through a unified query interface. A critical task for a metasearcher to process a query efficiently and effectively is the selection of the most promising databases for a query, a task that typically relies on statistical summaries of the database contents. Unfortunately, web-accessible text databases do not generally export summaries of their contents. In the past, query-based algorithms have been proposed to automatically build such summaries for web-accessible databases [4, 8]. The goal is to construct an augmented dictionary of all words that appear in the database, and their frequency. One of the algorithms described in [4] automatically discovers the content of a text database by first querying the database with some seed words, and then extracting new words from the retrieved documents to construct new queries. A somewhat simplified version of this algorithm (e.g., the stopping condition in [4] is different, for execution efficiency) is as follows:

¹Reference [1] introduces alternative querying strategies—notably *QXtract*—that do not follow this general structure.

- (0) while *seed* has an unprocessed word t
- (1) retrieve up to *MaxResults* documents matching t
- (2) extract new words t_e from these documents
- (3) augment *seed* with t_e

The output of this algorithm is a set of words and their approximate frequency in the database. If the extracted summary is incomplete, the accuracy of the database selection step might suffer, and so will the overall effectiveness of the metasearching process.

We observe that the querying algorithms for both *Task 1* and *Task 2* share a key characteristic: only the documents that are “reachable” from the initial queries will be discovered by these algorithms. In this paper, we present a querying model that describes the general querying approaches used for *Tasks 1* and *2*. In Section 2 we model the two tasks using our “reachability” graph formalism, and derive a single *reachability* metric based on the connectivity of this graph, to predict the success or failure of a general class of algorithms for *Tasks 1* and *2*. We illustrate the construction of the reachability graph for real text databases in Section 3. Then, in Section 4, we provide an efficient sampling-based technique for estimating the reachability of a given database. We evaluate our estimation techniques in Section 5 over real text databases for the tasks described above, showing that we can successfully estimate the reachability of a database by examining only a small document sample from the database. We conclude the paper in Section 6 with discussion of future work and potential applications of our techniques.

2. MODEL

In this section we present our model of query-based access to text databases. We first provide the intuition behind our model using *Tasks 1* and *2* (Section 2.1), and develop this intuition into a model using the “reachability” graph formalism (Section 2.2). We then present a key observation about the general structure of the reachability graphs that emerge when querying text databases (Section 2.3), which will lay a foundation for our estimation techniques of Section 4.

2.1 Querying Text Databases Revisited

The common characteristic of the “bootstrapping” approaches that have been applied to *Tasks 1* and *2* is that they start with a small set of *seed* queries, and use the information extracted from the retrieved documents for additional querying.

For example, consider the *Task 1* scenario, illustrated in Figure 1, and assume that the initial set of seed tuples consists of one tuple, t_1 . The database is queried using t_1 as described in Section 1. As a result, the document d_1 is retrieved. From this document, we extract the new tuple t_2 . After adding t_2 to the *seed*, and sending it as a query to the database, we extract the new tuple t_3 , which in turn retrieves the document d_4 that contains the tuple t_4 . Note that t_3 also retrieves the document d_2 , which “rediscovered” the tuple t_2 . As we can see, the tuples t_2 , t_3 , and t_4 are all reachable from t_1 , as shown in the resulting “reachability” graph on the right side of Figure 1. In contrast, tuple t_5 is not reachable from any of these tuples. As a result, t_5 will not be discovered by the algorithm if t_1 is the only “seed” tuple.

The procedure described above can be summarized using the graphs in Figure 1. By analyzing the structure of these graphs we can get a better understanding of the process and predict whether a particular query-based access method can succeed in reaching all (or a significant fraction of) the content of interest stored in the database. With this goal in mind, we now formally present our query-based reachability model.

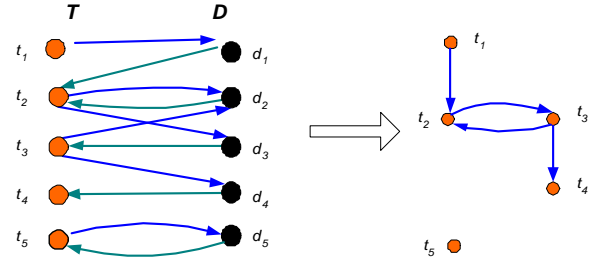


Figure 1: Portion of the querying and reachability graphs of a database.

2.2 Querying and Reachability Graphs

First, we define the “queries” more formally and then we define our graph-based representation of querying. Recall that for *Task 1* the queries consisted of the conjunctions of the extracted *tuple attributes*, while for *Task 2* the queries were the *words* extracted from the retrieved documents. Both types of queries can be modeled as instances of a general unit of information, a *token*, which is extracted from a document—in an application-specific way—and can be used for querying.

DEFINITION 1.: We define a token as a unit of information that can be extracted from a document and can be converted into a Boolean query, perhaps involving phrases. \diamond

The actual choice of what is considered a *token* is application-specific. The tokens might be the *words*, or the *named entities* (e.g., “Microsoft Corp.”) that appear in the documents, or even tuples (such as $\langle flu, aspirin \rangle$), which can be transformed to the query “flu” AND “aspirin” for an automatically extracted relation (such as *Treats(Disease, Drug)*). Using this definition of a token, we can now define the *querying graph* more formally.

DEFINITION 2.: We define the querying graph $QG(T, D, E)$ of a database with respect to some task’s querying strategy as a bipartite graph containing tokens T and documents D as nodes, and a set of edges E between T and D nodes. A directed edge from a document node d to a token node t means that t occurs in d . A directed edge from a token node t to document node d means that d is returned from the database as a result to a query that consists of the token t . \diamond

For example, suppose the token $t_1 = \langle flu, aspirin \rangle$ retrieves a document d that also contains another token $t_2 = \langle cold, tylenol \rangle$. Then, we insert an edge into QG from t_1 to d , and also an edge from d to t_2 . We consider an edge $d \rightarrow t$, originating from a document node d and pointing to a token node t , as a “contains” edge and an edge $t \rightarrow d$, originating from a token node t and pointing to a document node d , as a “retrieves” edge. Notice that the existence of the edge $t \rightarrow d$ in the graph does not imply the existence of the edge $d \rightarrow t$, and the existence of the edge $d \rightarrow t$ in the graph does not imply the existence of the edge $t \rightarrow d$.

The *querying graph* representation can accommodate constraints that appear while querying real text databases. For example, a database might have an upper bound *MaxResults* on the number of documents returned as a result to a query. This is modeled by constraining any token vertex in the querying graph to have out-degree no larger than *MaxResults*. Another real-life constraint for a query-based algorithm might be an upper limit *MaxDocs* on the total number of documents retrieved. In this case, the algorithm to find all tokens that are reachable from an initial seed set (which can be considered as a walk on the graph) should not cross more than *MaxDocs* edges of the type $t \rightarrow d$.

While the querying graph thoroughly describes the querying process, what we are really interested in is the *reachability graph* of the database, which is derived directly from the querying graph.

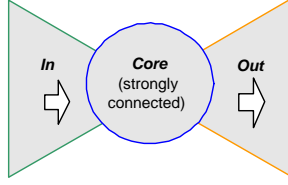


Figure 2: Structure of connected components in directed graphs.

DEFINITION 3.: We define the reachability graph $RG(T, E)$ of a database with respect to some task’s querying strategy as a graph whose nodes are the tokens T that occur in the database, and whose edge set E is such that a directed edge $t_i \rightarrow t_j$ means that t_j occurs in a document that is retrieved by t_i . \diamond

In Figure 1(b) we show the reachability graph derived from the underlying querying graph, illustrating how its edges are added. Since token t_2 retrieves document d_3 that contains token t_3 , the reachability graph contains the edge $t_2 \rightarrow t_3$. Intuitively, a path in the reachability graph from a token t_i to a token t_j means that there is a set of queries that start with t_i and lead to the retrieval of a document that contains the token t_j . In the example in Figure 1, there is a path from t_2 to t_4 , through t_3 . This means that query t_2 can help discover token t_3 , which in turn helps discover token t_4 . The absence of a path from a token t_i to a token t_j in the complete reachability graph means that we cannot discover t_j starting from t_i . This is the case, for example, for t_2 and t_5 in Figure 1.

The reachability graph is a directed graph, and its connected components can be described using the “bowtie” structure in [3]. Consider a strongly connected component *Core* in a reachability graph. By definition, every token in the *Core* is reachable via a directed path from every other token in the *Core*. Additionally, the tokens in the *Core* are reachable via a directed path from other tokens not in the *Core*, to which we refer as the *In* component (Figure 2). Finally, other nodes not in the *Core* or the *In* components are reachable via a directed path from the *Core* tokens. We refer to these nodes as the *Out* component (Figure 2). For example, consider the strongly connected component that consists of nodes t_2 and t_3 in Figure 1(b). The *In* component for this *Core* consists of the node t_1 , while the *Out* component contains the token t_4 .²

Having described the general shape of connected components in the reachability graph, we now turn to a quantitative analysis of the relative sizes of the different parts of the graph. We conjecture that the reachability graph of a database for tasks such as *Tasks 1* and *2* tends to belong to the well-studied family of power-law graphs. Power-law distributions have been known to arise in text domains [12]; additionally, power-law graphs have recently been observed to be a good model for graphs in related domains such as the web [3] and the Internet [7] graphs. One property of interest of power-law graphs is that the size of their connected components can be estimated using only a small number of parameters, as we describe next.

2.3 Reachability of Power-Law Graphs

A power-law graph [2] is a graph that has vertices with degrees that follow a *power-law* distribution. The power-law distribution states that the expected number of vertices y with degree k is:

$$y = e^\alpha \cdot k^{-\beta} \quad \Rightarrow \quad \ln(y) = \alpha - \beta \cdot \ln(k) \quad (1)$$

where the parameters α and β are the intercept and the slope of the

²Additionally, other nodes not in the *In*, *Out*, or *Core* might still be connected to these components (e.g., the nodes in the “tendrils” of the component [3]). These additional nodes do not help in our reachability analysis, and hence we do not consider them further.

line with best fit to the degree distribution plotted on the log-log scale.

Power-law graphs are actively studied in the graph theory community. Recent results [2, 5] allow to efficiently estimate properties of a given power-law graph. Specifically, Aiello, Chung, and Lu [2, 5] show that by using the average degree of the vertices in a random undirected power-law graph and the parameters α and β of the power-law distribution, it is possible to predict the size of the biggest connected component C_G of a graph, also called the “giant component”. If the giant component emerges, then the remaining connected components are expected to be *small*, with a size distribution that also follows the power law.

We conjecture (and we study this experimentally in Section 3) that the reachability graphs in real text databases for our retrieval tasks can be modeled as *directed* power-law graphs. We use the giant *strongly connected* component *Core* to define the giant component C_{RG} in the reachability graph RG . More specifically, we define C_{RG} as the biggest strongly connected component *Core*, with its associated *In* and *Out* components in the “bowtie” structure described above. According to the power-law, if a giant strongly connected *Core* component exists, then the remaining strongly connected components (and their associated *In* and *Out* components) are expected to be small.

We can now define the *reachability* of a text database with respect to some task’s querying strategy. As we discussed, in order for tasks such as *Tasks 1* and *2* to succeed, the extracted tokens must help discover other new tokens, which by definition are reachable from the previously discovered tokens. As we discussed above, if a large component C_{RG} exists in the reachability graph RG , then a token not in C_{RG} necessarily belongs in a *small* component and can help discover only a small number of new tokens. On the other hand, any token in the *In* or *Core* portions of C_{RG} will allow the querying strategy to discover all of the tokens in the *Core* and *Out* portions of C_{RG} . In other words, the relative size of the *Core* and the *Out* portions of C_{RG} can be used to predict the performance of query-based algorithms for tasks such as *Tasks 1* and *2* for executions where at least one initial seed token happens to be part of the *In* or *Core* portions of C_{RG} . Thus, we define the *reachability* of a text database as the fraction of the nodes T of the reachability graph that belong to the *Core* and *Out* portions of the giant component C_{RG} :

$$reachability = \frac{|Core(C_{RG})| + |Out(C_{RG})|}{|T|} \quad (2)$$

In the rest of the paper, we turn to the problem of how to efficiently approximate the reachability of a database for some task’s querying strategy.

3. REACHABILITY OF REAL DATABASES

In this section we show that the reachability graphs constructed over real text databases for *Tasks 1* and *2* have an approximate power-law degree distribution. Hence, we conjecture that power-law graphs can model the structure of reachability graphs for these retrieval tasks. We illustrate our observations using two real text databases, one for each of our retrieval tasks.

NYT: This database is a collection of 135,000 newspaper articles from The New York Times, published in 1995. We use this database for an instance of *Task 1*: to retrieve all of the *tuples* describing disease outbreaks (e.g., *⟨Typhus, Belize, June 1995⟩*), extracted from the NYT database using *Proteus* [11], a sophisticated information extraction system developed at New York University. A total of 8,859 tuples were extracted from the collection using an exhaustive scan of the database (which required over two weeks to complete). In this case, the *tokens* correspond to the *tuples* of the target relation, and the queries are constructed using the conjunction of the

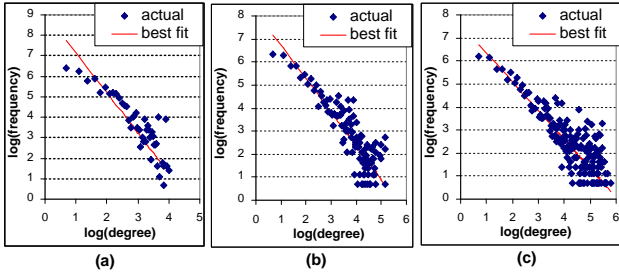


Figure 3: The outdegree distribution of the NYT reachability graph for Task 1 when (a) $MaxResults = 10$, (b) $MaxResults = 50$, and (c) $MaxResults = 200$.

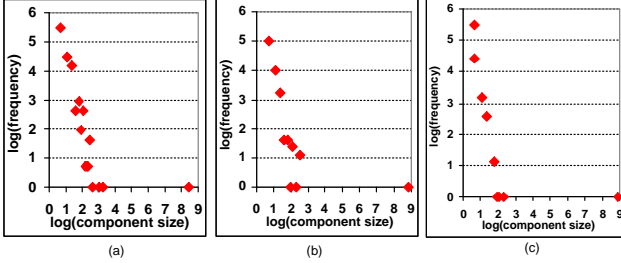


Figure 4: The component size distribution of the NYT reachability graph for Task 1 when (a) $MaxResults = 10$, (b) $MaxResults = 50$, and (c) $MaxResults = 200$.

first two attributes of each tuple (e.g., “Typhus” AND “Belize”). The complete reachability graph RG is computed by querying the database with all the 8,859 tuples extracted beforehand from the collection. To simulate constraints that search engines might impose, we limit the maximum number of documents, $MaxResults$, retrieved for each query.

We report the degree distribution of the resulting reachability graphs of the NYT database with respect to Task 1 in Figures 3(a)-(c) when $MaxResults$ is set to 10, 50, and 200 respectively. We show the power-law distribution that best fits the data. As we can see, the outdegree distribution is closely related to the fitted distribution. We report the size distribution of *connected components* in Figure 4, which also agrees with the size distribution expected for power-law graphs.

20NG: This database is the “20 newsgroups” collection of approximately 20,000 Usenet articles, from the UCI Machine Learning Repository. We now describe the construction of the reachability graph with respect to Task 2 (building a comprehensive summary of word document frequencies in the database). In this task, the tokens are the words in the documents. Such summaries typically ignore stopwords, and therefore we do not include stopwords in our reachability graph. Figures 5 (a) and (b) report the outdegree distribution of the reachability graph constructed for Task 2, for $MaxResults$ equal to 1 and 10, respectively. The outdegree distribution follows a power-law form for a large part of the distribution. We can model the distribution more accurately using extensions of the pure power-law model (e.g., see [9]), but a full discussion of other candidate distributions is beyond the scope of this paper. For $MaxResults=1$ and 10 we have just one connected component, which includes all the tokens of the reachability graph.

We conjecture that these observations will hold for the reachability graphs constructed over other text databases as well. We will explore this further experimentally in our future work. For the text databases where our conjecture holds, we will be able to predict the reachability of the databases (and consequently the performance of algorithms for Tasks 1 and 2) without constructing the complete reachability graph. Instead, we can simply estimate the param-

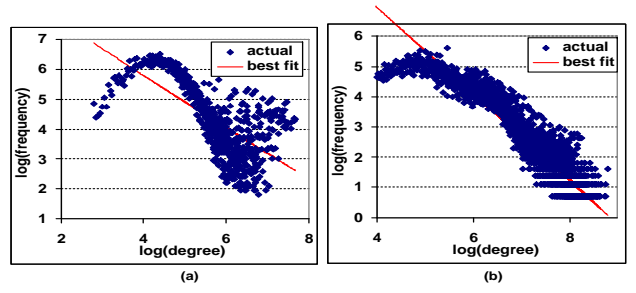


Figure 5: The outdegree distribution of the 20NG reachability graph for Task 2 when (a) $MaxResults = 1$, and (b) $MaxResults = 10$.

eters of the outdegree distribution of the corresponding reachability graphs, as discussed next.

4. ESTIMATING GRAPH PROPERTIES

In this section we describe how to estimate the reachability of a database for a query strategy. Specifically, Section 4.1 shows that we can base our estimates on the average node outdegree d in the reachability graph, which in turn we can estimate using a small document sample from the database. Section 4.2 describes our document sampling method.

4.1 Estimating Reachability

As we discussed, in a power-law graph at most one giant connected component is expected to emerge, and the rest of the connected components are expected to be small. Thus, the reachability estimation for power-law random graphs reduces to estimating the size of the giant component, if any.

Chung and Lu [5] showed that for an undirected power-law graph G with values of $\beta < \beta_0$ ($\beta_0 \approx 3.475$)³, a giant component emerges, while for smaller values of β all components are expected to be small. More specifically, [5] estimates the relative size of the giant component C_G when $\beta < \beta_0$ as follows:

$$\frac{|C_G|}{|T|} \geq \begin{cases} 1/d \cdot (1 - \frac{2}{\sqrt{de}}) & \text{if } d \geq e \\ 1/d \cdot (1 - \frac{1+\log d}{d}) & \text{if } 1 < d < e \\ 0 & \text{if } 0 < d \leq 1 \end{cases} \quad (3)$$

where d is the average degree of G . Note that the estimate *depends only on d* .

We believe that Equation 2 can be applied to estimate the size of the giant component C_{RG} in our *directed* reachability graph RG , and consequently to estimate the *reachability* of a database with respect to a given extraction task, where d in the equation above is the average *outdegree* of RG . Note that while *reachability* (Equation 2) is defined in terms of the *Core* and *Out* portions of C_{RG} , the equation above predicts the relative size of the *complete* giant component in the undirected graph. Therefore, the value predicted by Equation 3 will overestimate the *reachability*. Unfortunately, we are not aware of any similarly compact theoretical results to estimate the sizes of the specific portions of the giant component in directed graphs. As such results are developed, we could use them to improve the quality of our estimation.

Recall that by applying the results in [5] we can estimate the *reachability* of a database with respect to a given task by estimating the average outdegree d . We now describe an efficient document sampling technique that can be used to estimate d based on an observed (small) sample of the reachability graph.

³ β is the absolute value of the slope of the linear fit to the degree distribution. See Section 2.3.

4.2 Sampling Text Databases for Estimating Reachability

In order to estimate d , we retrieve a small document sample D_S and construct a reachability graph RG_S for D_S . By definition, RG_S is a subgraph of the complete reachability graph RG . We construct RG_S as follows. We start with a small number (e.g., 50) of seed tokens $T_{seed} \subset T$. We then query the database for each token $t_i \in T_{seed}$, retrieving up to $MaxResults$ documents for each query. For each document d retrieved by a token t_i , we extract each token t_j in d . For each t_j , we insert an edge $t_i \rightarrow t_j$ into RG_S .

Having obtained our sample subgraph RG_S , we can use it to estimate parameters of interest for the complete reachability graph RG . Specifically, we estimate d , the average outdegree of RG , as the average outdegree of the nodes in T_{seed} . Note that the outdegree of each node $t_i \in T_{seed}$ in RG_S is equal to the outdegree of t_i in RG . Since we draw T_{seed} randomly, we expect that the average outdegree of the corresponding vertices in the partial reachability graph will reflect the average outdegree of the complete RG .

The estimate of the average outdegree is used to predict the relative size $|C_{RG}|/|T|$ of the giant component as described in Equation 3, which approximates the reachability of the text database in question (Equation 2).

5. EXPERIMENTS

We now report experimental results for the technique that we described in Section 4. In Section 5.1 we describe the experimental setup, and in Section 5.2 we report the results of our preliminary experiments.

5.1 Experimental Setup

To evaluate the accuracy of our estimation technique, we constructed the reachability graph for the NYT and 20NG databases (Section 3). The reachability graph for NYT was constructed for *Task 1* and the reachability graph for 20NG was constructed for *Task 2*, for different values of $MaxResults$, one of the constraints that may be imposed by the text database. For each reachability graph, we compute the size $|C_{RG}|$ of its giant component using the STRONGLY-CONNECTED-COMPONENTS algorithm from [6] to compute the *Core*, and subsequently the *In* and *Out* components. Using $|Core(C_{RG})|$ and $|Out(C_{RG})|$ we compute the reachability values (Equation 2), which are reported in Figure 6, and serve as the “gold standard” for evaluating the estimation accuracy of our method.

Observe that almost all of the tokens for *Task 2* over 20NG are reachable for all values of $MaxResults$ (i.e., a complete database summary can be constructed with exhaustive querying)⁴. In contrast, a large fraction of the tokens for *Task 1* over the NYT database are not reachable for values of $MaxResults$ below 200 (i.e., by following the approach of *Task 1* we will not be able to reach a large fraction of the tuples in the target relation no matter how many queries we issue to the database).

For each database and each value of $MaxResults$, we apply the sampling technique of Section 4.2 by starting with a different number of seed tokens. Specifically, we sample the corresponding database using S randomly chosen *seed* tokens⁵ from T to construct RG_S . We experimented with $S = 10, 50, 100$, and 200, which means that we would send 10, 50, 100, and 200 token queries to the database to estimate its reachability. The maximum number of documents retrieved during sampling has a strict upper bound equal

⁴The C_{RG} for $MaxResults = 1$ consists of *Core*, which contains 95.5% of all tokens in C_{RG} , and *In*, which contains the rest.

⁵We assume that we can somehow obtain (e.g., as user input) an initial seed set of the appropriate size S . When this is not possible, we can repeatedly obtain random *document* samples until S seed tokens are extracted.

$MaxResults$	NYT				20NG
	<i>Core</i>	<i>In</i>	<i>Out</i>	<i>reachability</i>	<i>reachability</i>
1	0.001	0.003	0	0.001	0.955
10	0.078	0.156	0.063	0.141	1
50	0.260	0.159	0.200	0.460	1
100	0.334	0.132	0.274	0.608	1
200	0.388	0.102	0.334	0.722	1
1000	0.477	0.036	0.429	0.906	1

Figure 6: The relative size of the subcomponents of C_{RG} for the NYT and 20NG databases and for *Tasks 1* and *2* respectively, for different values of $MaxResults$.

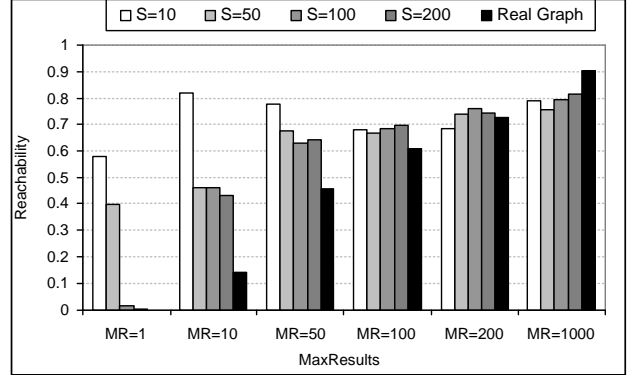


Figure 7: The reachability estimates for the NYT database for *Task 1*, for different values of $MaxResults$ and seed sample size S .

to $MaxResults \cdot S$.

5.2 Experimental Results

We report the estimation results for the NYT database for *Task 1* in Figure 7. As we can see, for *Task 1* our technique is able to estimate the reachability of the database with varying success. For example, when $MR = 1$ the actual reachability is 0.001. For sample size S of 100 and 200 queries, we estimated the reachability to be 0.015 and 0.002, respectively. While this estimate has high relative error, it is *good enough*, as it indicates that an algorithm for *Task 1* will not succeed in retrieving all necessary documents if $MaxResults = 1$. For the remaining values of $MaxResults$ we still generally overestimate the reachability of the database, but our estimates are closer to the real value. As we discussed above, the overestimates are partly caused by using the prediction of Equation 3 to estimate the relative size of only the *Core* and *Out* portions of the giant component. Another possible cause of error is the divergence of real reachability graphs from the idealized, pure power-law model, and may be remedied by extending our work to include refinements of the power-law model. While not exact, our estimates are still indicative of the general structure of the reachability graph, and consequently can be used as a rough predictor of expected performance of algorithms for *Task 1*, as we will discuss in the next section.

Figure 8 reports the results for estimating reachability of the 20NG database with respect to *Task 2*. The results show that our estimation technique was able to accurately detect that the 20NG database is completely reachable for *Task 2*. For example, while the *Core* for $MR = 10$ includes the complete graph (i.e., the reachability is 1), we estimated the reachability to be 0.95 by submitting only 10 queries, which allows us to predict that general approaches for *Task 2* will be successful for this database.

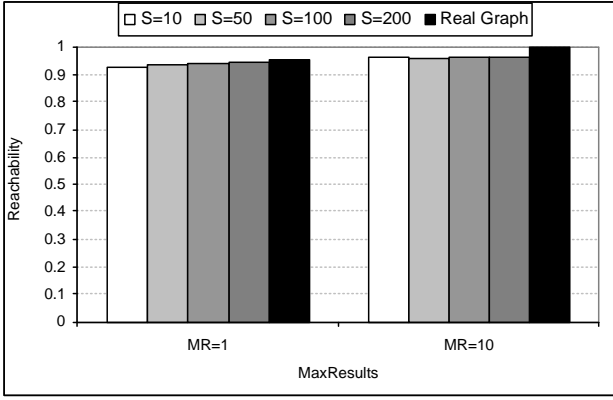


Figure 8: The reachability estimates for the 20NG database for Task 2, for different values of *MaxResults* and seed sample size *S*.

6. DISCUSSION

We presented a model for query-based access to text databases for general information extraction and database summary construction tasks. To the best of our knowledge, this is the first attempt to model query-based access to text databases, and our work parallels the efforts of modeling the web [3] and Internet [7] topologies. Based on this model, we developed the *reachability* metric, which indicates the expected performance of a general class of algorithms for these tasks. To complement our model, we presented an efficient technique for computing approximate values of reachability for a database with respect to the desired task.

We believe that our graph-based abstraction of the querying process can be used to model a variety of query-based access methods. Currently, most query-based algorithms are only empirically tested, with limited theoretical justification. We plan to model other query-based algorithms and use results from graph theory to provide theoretical justification for the observed experimental results.

In this work, our predictions for Tasks 1 and 2 are corroborated by empirical studies presented in [1] and [4] respectively. Reference [1] reports using *Tuples*, an implementation of the strategy for Task 1 for extracting the *DiseaseOutbreaks* relation over the NYT database with *MaxResults*=50. We found that all but a handful of the tuples retrieved by *Tuples* in [1] are in the $Core \cup Out$ portions of the giant component of the corresponding reachability graph, and the resulting recall of the strategy is therefore correctly predicted by the reachability value of 0.46 (Figure 6). If higher recall is desired, our reachability predictions could be used to either increase –if possible– the maximum number of documents, *MaxResults*, returned for each query (at the expense of precision), or choose an alternative querying strategy such as *QXtract* [1], which generates queries automatically by following a different approach. On the other hand, our model predicted that the algorithm proposed by Callan et al. [4] for Task 2 can successfully discover all the words that appear in the text database, as long as no limitation on the number of queries issued is imposed.

Finally, other properties of the reachability graph are also of interest and can be estimated using a small number of parameters. The edge density of the reachability graph is an indication of the rate at which an algorithm can obtain new information by querying a text database. The diameter of the graph shows the minimum required effort to retrieve all the information stored in the database.

Additionally, the *querying* graph is also a useful tool for studying the efficiency of different algorithms. In this paper we have seen that the *reachability* graph can be used to predict whether a method can succeed in retrieving all the tokens stored in a database. However, the reachability graph cannot reveal how many queries

are required to retrieve these tokens. In contrast, this information could be derived from the querying graph. For example, the reachability metric predicts that the algorithm in [4] for Task 2 can retrieve all the tokens from the 20NG database. However, to achieve the goal of retrieving all the tokens, we have to issue thousands of queries and retrieve thousands of documents from the database, which is modeled by crossing thousands of edges in the corresponding querying graph. By issuing only a small number of queries and retrieving up to a total of 300-500 documents, as suggested in [4], it is possible to retrieve only (arguably the most “important”) 15%-20% of the tokens in 20NG, which can be predicted by analyzing the querying graph for this task. In the future, we would like to further study the properties of the querying graph to determine how effective an algorithm can be if there is a limit on the number of queries or on the number of documents that can be retrieved from the database.

We believe that our model can serve as inspiration to develop even more comprehensive models and provides useful abstraction tools for the study of a variety of query-based algorithms.

ACKNOWLEDGEMENTS : This material is based upon work supported by the National Science Foundation under Grants No. IIS-9733880 and No. IIS-9817434. We thank Michael E. Agishtein for many fruitful discussions, and Regina Barzilay for her helpful comments.

7. REFERENCES

- [1] E. Agichtein and L. Gravano. Querying text databases for efficient information extraction. In *Proceedings of the 19th IEEE International Conference on Data Engineering (ICDE 2003)*, 2003.
- [2] W. Aiello, F. Chung, and L. Lu. A random graph model for massive graphs. In *Proceedings of the ACM Symposium on Theory of Computing (STOC 2000)*, 2000.
- [3] A. Z. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, and J. L. Wiener. Graph structure in the web. In *Proceedings of the Ninth International World Wide Web Conference (WWW9)*, pages 309–320, 2000.
- [4] J. Callan and M. Connell. Query-based sampling of text databases. *ACM Transactions on Information Systems*, 19(2):97–130, 2001.
- [5] F. Chung and L. Lu. Connected components in random graphs with given degree sequences. *Annals of Combinatorics*, 2002.
- [6] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. McGraw-Hill Science, Mar. 1990.
- [7] M. Faloutsos, P. Faloutsos, and C. Faloutsos. On power-law relationships of the Internet topology. In *SIGCOMM*, 1999.
- [8] P. G. Ipeirotis and L. Gravano. Distributed search over the hidden web: Hierarchical database sampling and selection. In *Proceedings of the 28th International Conference on Very Large Databases (VLDB 2002)*, 2002.
- [9] M. Mitzenmacher. A brief history of generative models for power law and lognormal distributions. In *First Workshop on Algorithms and Models for the Web-Graph*, 2001.
- [10] M. A. Shah. ReferralWeb: A resource location system guided by personal relations. Master’s thesis, M.I.T., May 1997.
- [11] R. Yangarber and R. Grishman. NYU: Description of the Proteus/PET system as used for MUC-7. In *Proceedings of the Seventh Message Understanding Conference (MUC-7)*, 1998.
- [12] G. K. Zipf. *Human Behavior and the Principle of Least Effort*. Addison-Wesley, 1949.

Efficient Dissemination of Aggregate Data over the Wireless Web *

Mohamed A. Sharaf
University of Pittsburgh
msharaf@cs.pitt.edu

Yannis Sismanis
University of Maryland
isis@cs.umd.edu

Alexandros Labrinidis
University of Pittsburgh
labrinid@cs.pitt.edu

Panos K. Chrysanthis
University of Pittsburgh
panos@cs.pitt.edu

Nick Roussopoulos
University of Maryland
nick@cs.umd.edu

ABSTRACT

The proliferation of wireless technologies along with the large volume of data available online are forcing us to rethink existing data dissemination techniques for data over the Web, and in particular for aggregate data. In addition to scalability and response time, data delivery to mobile clients with wireless Web connectivity must also consider energy consumption. In this work, we present a hybrid scheduling algorithm (DV-ES) for broadcast-based data delivery of aggregate data over the wireless Web. Our algorithm efficiently "packs" aggregate data for broadcast delivery and utilizes view subsampling at the mobile client, which allow for faster response times and lower energy consumption.

1. INTRODUCTION

Undoubtedly, the Web has brought massive amounts of information to everyone's fingertips, changing forever the way we learn the news, perform research, do business, etc.. The proliferation of wireless technologies, combined with the emergence of web services standards, will enable a new class of applications that empower mobile users to access great amounts of data in a seamless and efficient manner. We collectively refer to such infrastructure as the "*wireless Web*".

There are many motivating applications for the wireless Web. In this work, we are focusing on applications that utilize aggregate, summarized data. Mobile decision making is one such application, in which executives have access to time-sensitive, business-critical data from their mobile, hand-held devices. Scientists on the field would also rely on similar wireless Web technologies in order to access historic data for their research or to be able to retrieve summarized sensor measurements. We expect that the ease of deployment of wireless Web technologies will also change existing

applications (currently running on stationary, wired environments) to take advantage of the mobility it provides.

The ease of deployment and the high mobility of the wireless Web users will provide new opportunities but also bring forth limitations. On the one hand, using the wireless data communication medium allows the use of broadcast/multicast schemes which have better scalability than traditional unicast communication in disseminating data. On the other hand, being mobile poses energy consumption considerations in addition to the traditional response time improvement goals.

In this paper, we are addressing the issue of time and energy efficient delivery of aggregate data to mobile clients over the wireless Web. In wireless and mobile networks, broadcasting is the primary mode of operation for the physical layer. This means that if multiple clients request the same data at approximately the same time, the server may aggregate these requests, and only broadcast the data once. In this way, the low bandwidth of wireless link is better utilized, clearly improving user perceived performance. Several scheduling algorithms have been proposed that attempt to achieve maximum aggregation of requests [1, 3, 11, 2, 10].

In a decision making environment, sets of facts are analyzed along multiple dimensions. This led to the development of the multidimensional data model that represents a set of facts in a multidimensional space in a way that facilitates the generation of aggregate data. In this model, data is typically stored using a star schema. The star schema consists of a single fact table storing the measures of interest (e.g., *sales*, or *revenue*) and a table for each dimension (e.g., *supplier*, *product*, *customer*, *time* or *region*).

Decision making queries typically operate on aggregate data (i.e. summarized, consolidated data), derived from fact tables. The needed data can be derived using the *data cube* operator [4]. The data cube operator is basically the union of all possible *Group-By* operators applied on the fact table. A data cube for a schema with N dimensional attributes, will have 2^N possible views. Given that the data cube is an expensive operator, often views are pre-computed and stored at the server as *materialized views*. Basically, a view is an aggregation query, where the dimensions for analysis are the *Group-By* attributes and the measures of interest are the aggregation attributes.

In the case of queries to aggregate data, there is an interesting property which may exist between two queries: a

*This work is supported in part by NSF award ANI-0123705 and in part by the DoD-Army Research Office under Award No. DAAD19-01-1-0494. Prepared through collaborative participation in the Communications and Networks Consortium sponsored by the U. S. Army Research Laboratory under the Collaborative Technology Alliance Program, Cooperative Agreement DAAD19-01-2-0011. The U. S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation thereon.

set of aggregate data V^d requested by a client may be used to derive the set of aggregate data V^a requested by another client. This happens when V^d corresponds to a detailed version of V^a , which is more abstract. In such a case, V^a has a derivation dependency on V^d , and V^d subsumes V^a .

In earlier work [7, 8], we have exploited the above *subsumption* property by considering aggregate data as views, to increase sharing among clients and go beyond the exact matching of requests, thus allowing for further reduction in access time and power consumption. In this paper, we investigate another technique which also embodies the subsumption property to reduce the size of the broadcast. Specifically, we utilize the Dwarf technology to compress a view and all its subordinate views in order to achieve aggregation of multiple requests. Further, we combine the Dwarf-based scheme with the View-based aggregation to achieve further broadcast efficiency and scalability.

The rest of this paper is organized as follows. In the next section, we introduce a model to support aggregate data dissemination over the wireless Web. Overviews of the related work on dwarf and subsumption based scheduling are presented in Sections 3 and 4, respectively. In Section 5, we present *DV-ES*, our hybrid on-demand scheduling algorithm. Our simulation testbed and experiments are presented in Sections 6.

2. SYSTEM ARCHITECTURE

Our assumed architecture is based on *broadcast pull*. The Web server is responsible for maintaining and disseminating the aggregate data. A client sends a request for aggregate data on the *uplink channel* and then listens to the downlink channel for a response. A client can be in one of two modes, either in *active* mode, tuning and listening to the broadcast, or in *doze* mode, where the client is idle waiting for the response, with the receiver switched off [5].

A client request G is characterized by the set of its Group-By attributes A and measure attributes M . Hence, we represent a request as $G^{A,M}$ and the corresponding view as $V^{A,M}$. Without loss of generality, in this paper we are assuming only one measure attribute, hence we can drop the M part from the definition and use V^A to fully describe a view. Under this definition, a view V^{A1} *subsumes* view V^{A2} , if $A2 \subseteq A1$. In this case, V^{A2} is *dependent* on V^{A1} . We denote the number of dimensional attributes in the set A as $|A|$ and the size of view V^A in bytes as $|V^A|$.

The smallest logical unit of a broadcast is called a *packet* or *bucket*. A broadcast view is segmented into equal sized packets, where the first one is a *descriptor* packet. Every packet has a header, specifying whether it is data or descriptor packet, the offset (time step) to the beginning of the next descriptor packet, and the offset of the packet from the beginning of its descriptor packets. The descriptor packet contains a view descriptor which semantically identifies the aggregation dimensions, the number of attribute values or tuples in the view and the number of data packets accommodating that view.

We use bit encoding to represent both the semantics of a client request and the descriptor packet identifier. The representation is a string of bits; its length is equal to the number of the complete schema dimensions and each bit position corresponds to one of the dimensions a_1, a_2, \dots, a_n . If view V^A has dimension a_x ($a_x \in A$), then the bit at position x is set to 1, otherwise it is a zero.

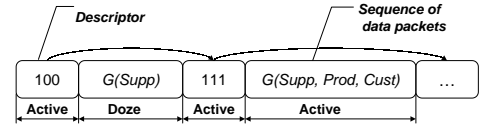


Figure 1: A Client Access to Broadcast

After a client submits a request for view V^R on the uplink channel, it follows a three-phase access protocol: (1) *initial probe*; (2) *semantic matching*; and (3) *view retrieval*.

In the initial probe phase, the client tunes to the downlink channel and uses the nearest packet header to locate the next descriptor packet. The semantic matching phase starts when the client finds the first descriptor packet, say for view V^B . In this phase, the client will “detect” whether view V^B satisfies its request exactly or whether the answer can instead be “inferred” from view V^B .

Depending on the matching result and the scheduling algorithm used (as we will see below), the client will either switch to the final retrieval phase or it will stay in the matching phase. In the former, the client stays in active mode tuning to the next sequence of data packets to read (download) view V^B . In the latter case, it will switch to doze mode reducing power consumption. Using the offset in the packet header, the client wakes up just before the next opportunity to read the next descriptor packet in the broadcast, after which, the semantic matching process is repeated. The access protocol is shown in Figure 1.

2.1 Performance Metrics

The performance of any scheduler in a wireless environment can be expressed in terms of:

- **Access Time:** It is the user perceived latency from the time a request is posed to the time it gets the response. Its two components are the *wait time* and *tune time*.
- **Tune Time:** It is the time spent by the client listening to the downlink channel either reading a descriptor packet or data packets containing the requested view. During tuning, the client is in active mode.
- **Wait Time:** The total time a client spends waiting between descriptor packets until it finds a matching one. A client is in doze mode during the wait time.

In active mode, a client device consumes energy that is orders of magnitude higher than in doze mode. For this reason, tune time has been traditionally used to evaluate the power consumption of a system in a mobile environment. However, the energy dissipated in doze mode becomes more significant when the client has to wait for a long time until its request is satisfied. Hence, in this paper we will adopt a weighted energy consumption cost model that includes the active and doze factors.

3. DWARF TECHNOLOGY

The *Dwarf Structure* [9] is a highly compressed structure for computing, storing and querying data cubes. It efficiently addresses both the storage space problem and the computation cost problem by factoring out prefix and suffix redundancies. Each redundancy is identified *prior* to its computation which results in significant computational

savings during creation. To help clarify the nature of the redundancies let us consider a cube with three dimensions a , b and c .

Prefix redundancy occurs when two Group-Bys share a common prefix (like Group-Bys abc and ab). For example, in the fact table shown in Table 1, store $S1$ appears a total of seven times in the corresponding cube and more specifically in the Group-Bys: $\langle S1, C2, P2 \rangle$, $\langle S1, C3, P1 \rangle$, $\langle S1, C2 \rangle$, $\langle S1, C3 \rangle$, $\langle S1, P1 \rangle$, $\langle S1, P2 \rangle$ and $\langle S1 \rangle$. The same happens with prefixes of size greater than one. This kind of prefix redundancy is factored out in the Dwarf Structure by storing each unique prefix just once. Dense areas of the cube benefit more by eliminating prefix redundancies, since there is no need to repeat the same values of the dimensions over all possible Group-Bys. It has been shown that for very dense cubes the corresponding dwarf is much less than the fact table.

Store	Customer	Product	Price
S1	C2	P2	\$70
S1	C3	P1	\$40
S2	C1	P1	\$90
S2	C1	P2	\$50

Table 1: Fact Table for cube Sales

Suffix redundancy occurs similarly when two Group-Bys share a common suffix (like Group-Bys abc and bc). For example, for the fact table in Table 1, since customer $C1$ shops only from store $S2$ then, for any value x of dimension **Product**, the Group-Bys $\langle S2, C1, x \rangle$ and $\langle C1, x \rangle$ always have the same aggregate values. This happens because the $\langle C1, x \rangle$ Group-By aggregates all the tuples of the fact table with customer $C1$ for any combination of stores. In this case however there is only one store $S2$ and the aggregation degenerates to the aggregation already performed for the Group-By $\langle S2, C1, x \rangle$. Such redundancies are very often in sparse datasets and even more apparent in cases of correlated dimension values – which are very common in real datasets. Suffix redundancies are factored out in the Dwarf Structure by *coalescing* the space of the corresponding suffixes. It has been demonstrated that the space required to store the Dwarf Structure for sparse datasets is many times smaller than that required by alternative techniques such as materialized views.

The Dwarf Structure is self-sufficient since it does need to access or reference the fact table in order to answer for any of the views of the cube and it provides an implicit index mechanism without needing any additional index for querying it. In addition any query can be answered with just one scan/traversal over the dwarf structure.

Sub-Dwarfs The intuition of using the Dwarf as a compressed broadcast unit can be depicted in Figure 2. For a four-dimensional cube of uniform data and a cardinality of one hundred for each dimension, we enumerate all possible views using the binary representation described in Section 2. For each view, the size of the materialized view (in bytes) along with the size of the corresponding *sub-dwarf* is depicted. The sub-dwarf corresponds to the dwarf that is having the corresponding view as fact table (i.e., the original fact table where all the dimensions, that are represented by 0 in the view binary representation, are projected out). The materialized view is an unindexed relation that holds all the

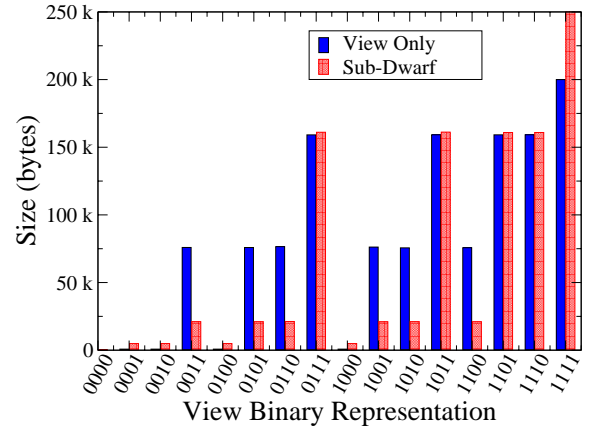


Figure 2: Sub-Dwarfs vs Materialized Views

tuples of the corresponding view. For this specific case we assumed that the values of each dimension are mapped to four-byte integers and that there is one measure represented by a float (four-byte) value.

It is important to point out that while the materialized view contains only the tuples of a single Group-By query, the corresponding sub-dwarf contains *all subordinate*¹ views. For example, for view 0110, the materialized view only holds the tuples of the view 0110, while the sub-dwarf contains all subordinate views 0000, 0010, 0100 and 0110. The subsumption operation in the sub-dwarf case degenerates to just a simple scan over the dwarf without any online aggregation.

The basic idea is that in many cases it is better to broadcast a sub-dwarf instead of the view since its size is much smaller than the size of the sum of the subordinate views. In addition, for dense views, the sub-dwarf is much smaller than the corresponding materialized itself making it “economic” to transmit even if there are no request for the subordinate views.

We observe that for views 0011, 0101, 0110, 1001, 1010 and 1100 the size of the sub-dwarf is much smaller than the size of the corresponding materialized view. The reason is that these views are quite dense since they contain only two dimensions and the savings by factoring out prefix redundancies is tremendous. For views 0111, 1011, 1101 and 1110 (i.e., all views with three dimensions) we observe that the size of the dwarf is slight larger than the corresponding view. In cases where the request workload accesses a lot of their subordinate views it could be beneficial to broadcast the sub-dwarfs instead of the corresponding views. Sub-Dwarfs of just one dimension (i.e., 0001, 0010, 0100 and 1000) are bigger than the corresponding materialized views due to various implementation overheads. The sub-Dwarf that corresponds to 1111 corresponds actually to the full cube since it contains all possible subordinate views.

4. SUBSUMPTION BASED SCHEDULER

In general, accesses for aggregate data have the following properties: *heterogeneity* (views can be of different dimensionality and of various sizes), *skewed access* (requests from clients usually form a hot spot within the data cube), and *subsumption* (it is often possible to use one detailed view to

¹views that can be computed from the fact table.

compute other summarized ones).

In previous work [7], we developed (*SBS- α*), the first on-demand broadcast scheduling algorithm that exploits the subsumption property of summary tables to increase sharing among clients. The *Subsumption-Based Scheduler* (*SBS- α*) consists of two components: A basic selection component, which captures the heterogeneity and skewed access properties and the α -optimizing component that exploits the subsumption property to reduce access time with minimum extra overhead in terms of energy.

SBS- α uses *Longest Total Stretch First* (LTSF) algorithm [1] as its basic selection component, where the stretch of a pending request is the ratio of the time the request has been in the system thus far to its service time. Hence, the stretch of request i for view V^X can be represented as: $\frac{W_i^X}{|V^X|/B}$, where W_i^X is the wait time of request i , $|V^X|$ is the size of V^X , and B is the bandwidth. Since the bandwidth is constant we can re-write the stretch as $\frac{W_i^X}{|V^X|}$.

In *SBS- α* , the server queues up the clients' requests as they arrive. When it is time for the server to make a decision which view to broadcast next, it computes the total stretch value for each view that has at least one outstanding request. The view with the highest total stretch value (say V^b) is selected to be broadcast, as in [1].

Ignoring the subsumption semantic of data cubes, a client that requested V^r , which is derivable from V^b , will wait until V^r is broadcast even if V^b is available sooner. In the extreme case of utilizing the subsumption property, the client will use V^b to derive V^r regardless of the relative cardinality of both tables and potentially incurring unnecessary extra energy. This increase energy is due to the extra time a client has to spend tuning to a detailed version of the aggregate rather than a summarized one and the accompanying high energy consumed in the active mode. For that reason, parameter α is used to define the degree of flexibility in using the subsumption property (that captures the degree of sharing) and it works as follows.

At the server side, upon deciding the broadcast of view V^b , the server discards every pending request for a view V^r that can be derived from V^b and satisfies the following property (α rule): The ratio of the difference in size between views V^b and V^r to view V^b is less than the α value. Formally, V^r can be discarded and is not broadcast, iff V^b is broadcast, $r \subset b$ and $\frac{|V^b| - |V^r|}{|V^b|} \leq \alpha$, where $\alpha \in [0, 1]$.

At $\alpha = 0$ there is no flexibility in using views; the client access is restricted to exact match and *SBS-0* is equivalent to LTSF. At $\alpha = 1$, the case of extreme flexibility, a client can use any subsuming matching view. Picking a reasonable value for α will balance the trade-off between reducing the wait time (doze energy consumption) and increasing the tune time (active energy consumption).

5. PROPOSED HYBRID ALGORITHM

We propose to integrate the two presented technologies (Dwarf and *SBS- α*) into a hybrid on-demand broadcast scheduling algorithm that employs view subsumption (similarly to *SBS- α*) and uses Dwarf cubes as a compact representation form for aggregate data. We named our hybrid algorithm *DV-ES(α)* to reflect the objects that can be included on the broadcast by the server (V for Views, or D for Dwarfs), and the operations that a client can perform (S for subsumption,

or E for extraction from the Dwarf).

Under *DV-ES(α)*, the server can broadcast either entire dwarf sub-cubes or plain views for a given set of dimensions (i.e., for a given Group-By query). Accordingly, a client may receive one of the following:

- the view that exactly matches its query, or
- a detailed view which subsumes the one it originally requested, or
- a dwarf sub-cube that contains the view.

As an example to illustrate the above cases, consider a client request for view (A, B) . The server can serve this request by either broadcasting a view or a dwarf. In the case of sending a view, it can either be the exact view (i.e., (A, B)), or an ancestor view (e.g., (A, B, C)). The same applies in the case of broadcasting a dwarf. That is, it can be either the dwarf corresponding to (A, B) , or a dwarf corresponding to an ancestor view, say (A, B, C) , which will physically contain view (A, B) as well as all possible combination of views that can be derived from (A, B, C) .

In the case where the server disseminates a dwarf sub-cube D^X , the client will download D^X and extract its requested view. Note that D^X will contain V^X and all the views that are descendant from V^X , hence, a client can use D^X to either extract V^X or any other views that is dependent on V^X . On the other hand, if the server decided to disseminate V^X rather than D^X , then a client will still be able derive any view that is subsumed by V^X .

The server decides whether to broadcast a view or dwarf based on the sizes: if the size of the view is less than the smallest dwarf that contains this view, then the view is selected, otherwise the corresponding dwarf sub-cube is selected. The intuition is to save bandwidth in the cases where the size of the dwarf is larger than the corresponding view.

DV-ES(α) is similar to *SBS- α* , consisting of an LTSF component for scheduling and the α -optimizing component to exploit the dependency between views. However, the implementation of these component is slightly different.

In *DV-ES(α)*, the LTSF scheduling component is modified to consider the two different representations of broadcast objects, namely dwarfs or views, when selecting the object to be broadcast. Specifically, since the object representation that yields the smaller size is selected, the stretch of a request i for view V^X is redefined as $\frac{W_i}{\min(|D^X|, |V^X|)}$ where $|D^X|$ is the size of the dwarf corresponding to V^X .

Also, the α optimizing component is modified to incorporate the fact that a request for V^r can be satisfied either by sending a view or by sending the equivalent dwarf (whichever is smaller in size). The α rule is redefined as follows:

$$\begin{aligned} & \text{if selection} = \text{dwarf then} \\ & \quad \text{if } \frac{|D^b| - \min(|D^r|, |V^r|)}{|D^b|} \leq \alpha \text{ then eliminate } G^r \\ & \text{else if selection} = \text{view} \\ & \quad \text{if } \frac{|V^b| - \min(|D^r|, |V^r|)}{|V^b|} \leq \alpha \text{ then eliminate } G^r \end{aligned}$$

where G^r is the request for V^r .

One way that a client could decide whether or not to use a particular object on the broadcast is by applying the same α -rule used at the server. However, this will require a client to have full information about the sizes of all views and their corresponding dwarfs.

We adopted an alternative way which assumes that clients have no prior knowledge about the sizes of views and dwarfs.

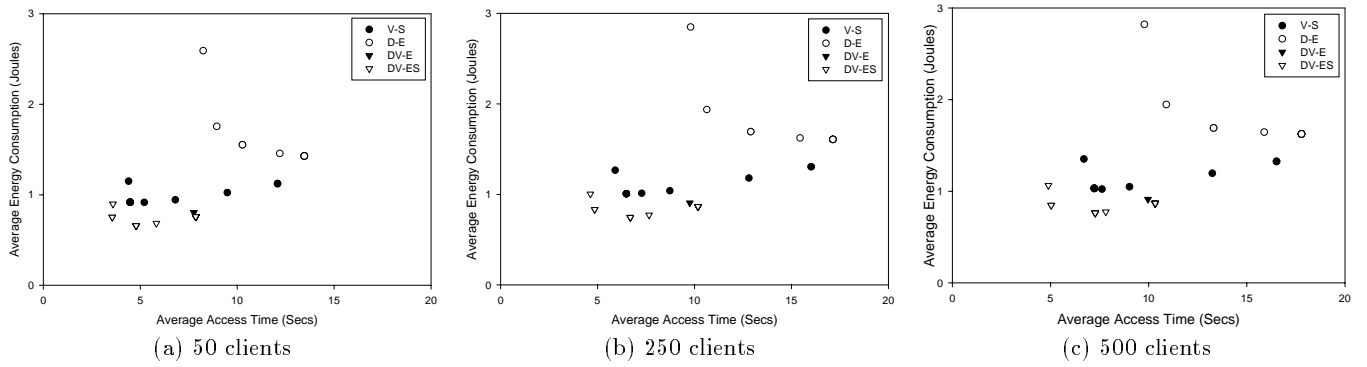


Figure 3: Energy vs Time for all scheduling algorithms

Instead, the server explicitly informs the clients which request can be satisfied by a given view or dwarf on the broadcast. This is achieved by augmenting each descriptor packet with a views encoding bitmap. The length of this bitmap is equal to the number of possible views. If a view can be inferred from the broadcast object and it satisfies the α -rule, the bit corresponding to this view is set to 1, otherwise it is set to 0. This approach has a negligible overhead in terms of bits added to the descriptor packet and it only requires a client’s knowledge of the multidimensional schema at the server which is already part of the meta-data information periodically broadcast by the server.

6. EVALUATION

We implemented a simulator to evaluate the performance of our proposed $DV-ES(\alpha)$.

6.1 Experimental Setup

Algorithms In addition to the SBS- α , in our evaluation we included two “base-case” algorithms in order to better understand the behavior of $DV-ES(\alpha)$. In the first algorithm, the server always broadcasts packed dwarf cubes, whereas in the second algorithm we emulate the hybrid scheduling algorithm, but remove the client’s subsumption ability.

For clarity in presentation, we name these algorithms with the X-Y scheme that we used to name our proposed **DV-ES**(α) algorithm, where X stands for the data objects that can be included in the broadcast (V for Views, or D for Dwarfs) and Y is the operations that clients can perform (S for subsumption, or E for extraction from the Dwarf). Under this scheme, the SBS- α algorithm is named **V-S**(α), the first base case algorithm is named **D-E**(α), and the second base case algorithm is named **DV-E**(α). Parameter α controls the degree of flexibility in deriving views or extracting them from a higher dimensional ancestor.

Workload We present experiments results using synthetic datasets. The base fact table has uniformly distributed data, where the default number of tuples is 100,000 and the default number of dimensions is 6 (we used a cardinality of one hundred for each dimension).

To test the system under a realistic workload, requests are generated by the clients according to Zipf distribution with the Zipf parameter $\theta=0.5$. Queries are sorted according to their size, so that queries to small size views occur with

higher probability than queries to detailed ones.

Time is reported in *Seconds* and the energy consumption is in *Joules*. We considered a wireless LAN where the broadcast channel has a bandwidth of 10 Mbps. Clients are equipped with the ORiNOCO World PC Card [6]. The card operates on a 5V power supply, using 9mA at doze mode and 185 mA at receiver, active mode.

6.2 Experiments

Figure 3 Figure 3 depicts the average access time and energy consumption for the different algorithms for values of α between 0.0 and 1.0 with 0.1 steps. For all algorithms, the access time decreases by increasing the value of α . So, for a certain algorithm, traversing the points from right to left corresponds to increasing the value of α .

Figure 3 shows the poor performance of the D-E algorithm, where only dwarfs are disseminated. DV-ES will always outperform D-E since in the cases where the dwarf size is bigger than the corresponding view, it broadcasts the view. Compared to the other algorithms, D-E will perform better only when the access is skewed toward aggregates whose dwarf representation is smaller than the corresponding view. This clearly is not the case in the given workload as shown in Figure 3.

We also noticed that the performance of DV-E for values of α from 0.0 to 0.9 is similar to that of DV-ES(0) (coinciding points in the graph). This shows that DV-E is almost insensitive to the parameter α . The explanation is that under the DV-E algorithm: 1) a dwarf is selected if its size is smaller than the corresponding view, and 2) a client extracts a view from this dwarf if the view minimum size representation is within an α distance from the selected dwarf. Given our workload, the first condition leads to selecting dwarfs to satisfy medium dimensionality queries, while selecting views to satisfy low and high dimensionality queries. However, most of the time these medium dimensionality dwarfs are orders of magnitude larger in size than the view version of their descendants. Hence, it requires a high value of α in order for extraction to take place (i.e., $\alpha=1$), which is the single point presented in Figure 3.

Finally, it is clear that the DV-ES algorithm outperforms V-S in both time and energy reductions. DV-ES utilizes the cases where the dwarf is smaller than the corresponding view and it allows for both extraction and subsumption. This improvement over V-S is more significant as the request rate increases (i.e., increasing the number of

clients from 50 to 500 as in cases (a), (b), and (c) in Figure 3), which demonstrates the scalability of DV-ES.

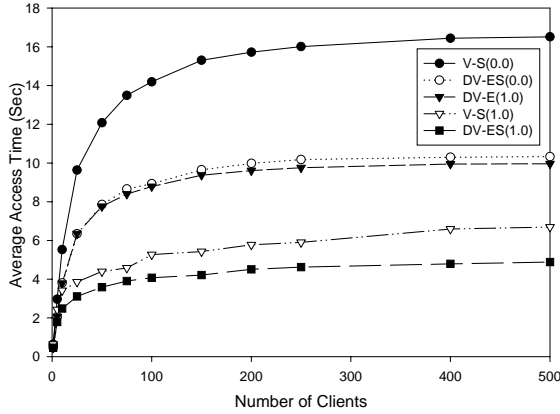


Figure 4: Access Time

Figure 4 We plot the response time of V-S, DV-E, and DV-ES in Figure 4 where the number of clients ranges from 50 to 500 and for values of $\alpha = 0$ and 1. All algorithms exhibit similar behavior: the average access time increases, but ultimately levels as the number of clients is increased. This behavior is normal for broadcast data delivery to clients with shared interests.

Figure 4 shows how the access time decreases with increasing α from 0 to 1: $\alpha = 1$ is the case where flexibility in using ancestors is experienced the most, and $\alpha = 0$ is the case where we do not allow using ancestors. It also shows that a significant reduction in access time is achieved by DV-ES compared to V-S, which is even more significant as the load increases. For instance, consider the cases of 50 and 500 clients where $\alpha = 1$. In the case of 50 clients, the access time decreased by 22% compared to V-S(1), while in the case of 500 clients, the reduction achieved by DV-ES(1) was 37% compared to V-S(1) and it is 3 times less than V-S(0).

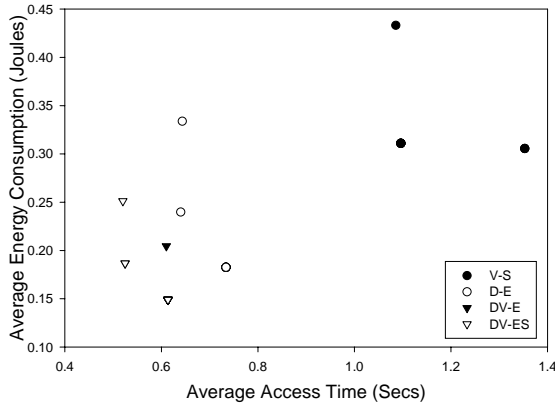


Figure 5: 4-dimension workload

Figure 5 In the previous comparisons we used a fact table of 6 dimensions, whereas in Figure 5 we are experimenting with a fact table that has denser views (by decreasing the number of dimensions to 4, while keeping the cardinality of one hundred for each dimension and the number of

tuples as the default 100K). This has the effect of generating fewer aggregate views than the 6-dimensions case, but they are denser. As shown in Figure 5, for a load of 500 clients, all the dwarf-based algorithms (D-E, DV-E, DV-ES) are performing better than V-S, where only views are disseminated. Comparing the performance of DV-ES(1) and V-S(1), we can see that DV-ES(1) outperforms V-S(1) in both access time and energy consumption. For access time, DV-ES(1) achieved a reduction equal to 50% compared to V-S(1), whereas this reduction was only 37% in the case of 6-dimensional fact table shown in Figure 3(c). For energy consumption, the reduction is 40%, as opposed to only 22% in the case of 6-dimensional fact table.

7. CONCLUSIONS

In this paper we proposed and evaluated *DV-ES*, a new on-demand broadcast scheduling algorithm for disseminating aggregated data over the wireless Web. DV-ES integrates the view-derivation properties of SBS- α that goes beyond the exact match of requests and the Dwarf technology that provides a compact representation for aggregate data. DV-ES achieves reduction both in access time and power consumption by selecting at any given scheduling point to broadcast either a View to be used by the clients to derive their requested data or an entire sub-cube from which clients extract their requested data.

Disclaimer: The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Laboratory or the U. S. Government.

8. REFERENCES

- [1] S. Acharya and S. Muthukrishnan. Scheduling on-demand broadcasts: New metrics and algorithms. *Proc. of 4th ACM/IEEE MobiCom Conf.*, Oct. 1998.
- [2] D. Aksoy and M. Franklin. RxW: A scheduling approach for large-scale on-demand data broadcast. *IEEE/ACM Tran. on Networking*, 7(6):846–860, 1999.
- [3] H. D. Dykeman, M. Ammar, and J. W. Wong. Scheduling algorithms for videotext systems under broadcast delivery. *Proc. of the 1986 Int'l Conf. on Communications*, pp. 1847–1851, June 1986.
- [4] J. Gray, et. al. Data Cube: A Relational Aggregation Operator Generalizing Group-by, Cross-Tab, and Sub Totals. *Proc. of the ICDE Conf.*, Feb. 1996.
- [5] T. Imielinski, S. Viswanathan, and B. R. Badrinath. Energy efficient indexing on air. *Proc. of the ACM SIGMOD Conf.*, pp. 25–36, May 1994.
- [6] ORiNOCO World PC Card. www.orinocowireless.com
- [7] M. A. Sharaf and P. K. Chrysanthis. Semantic-based delivery of OLAP summary tables in wireless environments. *Proc. of the CIKM Conf.*, Nov. 2002.
- [8] M. A. Sharaf and P. K. Chrysanthis. Facilitating Mobile Decision Making. *Proc. of the 2nd ACM Int'l Workshop on Mobile Commerce*, Sep. 2002.
- [9] Y. Sismanis, A. Deligiannakis, N. Roussopoulos, Y. Kotidis Dwarf: shrinking the PetaCube. *Proc. of ACM SIGMOD Conf.*, June 2002.
- [10] K. Stathatos, N. Roussopoulos, and J.S. Baras. Adaptive data broadcast in hybrid networks. *The VLDB Journal*, pp. 326–335, 1997.
- [11] J. W. Wong. Broadcast delivery. *Proc. of the IEEE*, 76:1566–1577, 1988.

Query Set Specification Language (QSSL)

Michalis Petropoulos Alin Deutsch Yannis Papakonstantinou
University of California, San Diego

{mpetropo,deutsch,yannis}@cs.ucsd.edu

ABSTRACT

Applications require access to multiple information sources and the data of other applications. WSDL-based web services are becoming a popular way of making information sources available on the web and, hence, to applications that need to consume them – often via data integration systems that combine the data of multiple sources. We argue that the function signature paradigm that is used today by web services cannot capture the query capabilities provided by structurally rich and functionally powerful information sources, such as relational databases. We propose the Query Set Specification Language (QSSL) that allows the concise description of sets of parameterized XPath queries. A QSS is embedded in a WSDL specification to form a specialized type of web services, called Data Services. Data Services connect the calls that the source accepts with the underlying schema. QSSL will be enhanced to describe subsets of XQuery expressions beyond XPath ones.

1. INTRODUCTION

Web Services Description Language (WSDL) [5] provides an XML format for describing functions offered via web services. The function signatures typically have fixed numbers of input and output parameters. However, the “function” paradigm is not adequate when the software components behind the web services are databases. One typically associates one function with each parameterized query but this is problematic since databases often allow a large or even infinite set of parameterized queries over their schema. For example, the administrator of a product catalog database may want to allow any query that selects products by a combination of selection conditions on the product’s attributes. Assuming the product has, say, 10 attributes, it is obviously impractical to specify 2^{10} function signatures.¹

In addition, the function paradigm does not state explicitly either the relationship between the input

parameters and the output or the semantic connections the available functions have with each other and with the underlying database. We classify such *web services* as *functional* and we argue that they are inappropriate for exporting structurally rich and functionally powerful information sources, such as relational and emerging XML databases.

We present a WSDL extension that enables *Data Services*, which overcome the shortcomings of functional web services. A data service exports the XML Schema [7] of an XML view. The data service also provides a set of parameterized queries that can be executed against the view. Hence the relationship between the input and output parameters is explicit, since the input corresponds to a query and the output to its result. Note that the view typically (but not necessarily) corresponds to a part of the underlying database.

The Query Set Specification Language (QSSL) is a WSDL extension that, given an underlying database, allows the concise and semantically meaningful description of set of parameterized queries. The set may be very large or even infinite, since powerful information sources (such as relational databases) support a large number of parameterized queries. Consequently, QSSL must be able to describe sets of parameterized queries without requiring exhaustive enumeration of them.

QSSL concisely describes sets of tree pattern (subset of XPath) queries. We plan to extend to subsets of XQuery. It lends itself to a compact and intuitive visual notation that forms the basis of an under-development GUI that allows the specification of QSSs.

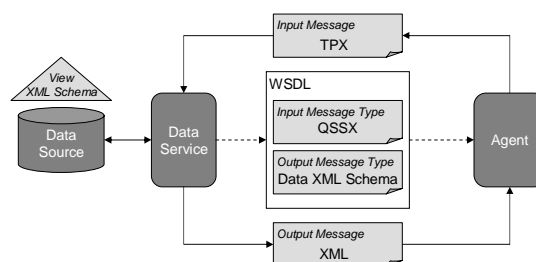


Figure 1. Data Service Architecture.

¹ In the particular example one can resolve the issue simply by allowing some input parameters to be null. This situation is generalized by QSS to capture multiple function signatures in just one WSDL operation [5].

Figure 1 shows the architecture of a data service published by a data source with a given view XML Schema. The query capabilities exported by a data service are published as a WSDL specification [5] that provides an agent with the means to formulate valid and acceptable queries and to be aware of the structure of the result. Notice that we translate QSSs into XML Schemas and we are thus compatible with the WSDL specification. The data service receives an input message from the agent and replies with an output message or a fault. The input message is a tree pattern (TP) query (subset of XPath), defined in Section 2, expressed in the TPX XML format, and the output message is an XML tree. The set of acceptable tree pattern queries (i.e., the set of acceptable input messages) is a Query Set Specification (QSS), defined in Section 2. A QSS describes the possibly infinite set of parameterized queries that are acceptable. The QSS is translated into an XML Schema (QSSX) describing the acceptable TPX messages.

1.1 Example

The running example is based on the XML Schema in Figure 2a that describes the structure of an airline database holding information about flights. The schema describes the flights carried out by one or more airline companies, where each flight has an origin and destination (`from` and `to` elements) and is scheduled at least once per week. In turn, each flight has one or more legs with a code, an origin and a destination and optionally the type of the aircraft used. Note that the schema of the actual airline database may be “richer” but we focus on the part that the database administrator exposes.

The database administrator allows queries that are having any combination of the following conditions:

- The name of the airline company is specified
- The origin and destination of one or more flights is optionally specified
- A day of the week is specified
- The origin of zero or more legs is optionally specified
- The destination of zero or more legs is optionally specified.
- The aircraft used for zero or more legs is optionally specified.

Notice that one may also specify combinations of origin, destination, and aircraft for legs. For the sake of the example, we also allow one to check whether a flight has a leg (existential condition).

The queries may return “airline” or “flight” elements.

This document presents the process of exporting such query capabilities using a data service. More

specifically, Section 2 defines the query language and the query set specification language, and shows how QSSL accommodates recursive schemas. Section 3 describes the XML syntax of TP queries and QSSs. Section 4 presents possible QSSL extensions and related work is discussed in Section 5.

2. SPECIFYING QUERIES AND QUERY SETS

We consider data services that support queries defined by a class of XPath expressions consisting of node tests, navigation along the child axis ‘/’ and the descendant axis ‘//’, and predicates denoted by ‘[]’. The established convention for representing this class of XPath expressions is to use *tree pattern (TP)* queries [2, 14]. We believe that support for tree pattern queries is a minimum data service requirement, since tree patterns are widely used in current applications, and since they are crucial building blocks of more expressive query languages such as XQuery [4]. Moreover, tree patterns provide an excellent visual paradigm which enables graphical user interfaces for constructing applications that produce and consume data services. For example, the XPath expression

```
flights/airline[name='Delta']/flight[from='JFK'][to='LAX'][day='MON'][leg[to='LAS']]
```

is represented by the tree pattern query in Figure 2b.

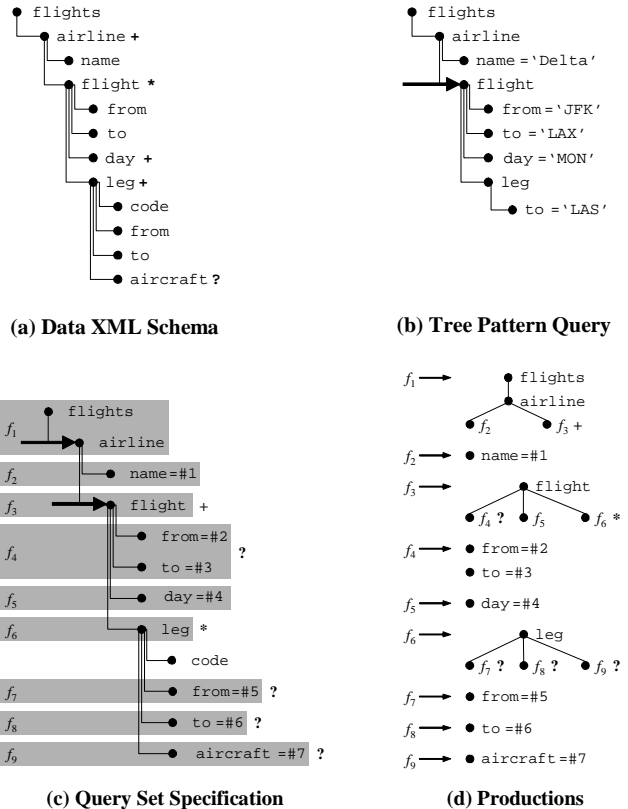


Figure 2. Airline Example.

The arrow pointing to the `flight` element node denotes the *result node* of the tree pattern query.

2.1 Query Set Specifications

We define a data service by specifying the set of tree pattern queries it supports. First, we introduce *parameterized tree patterns (PTPs)*, which are TP queries where the constants are replaced with parameters. A PTP query specifies an infinite set of TP queries, each TP corresponding to a parameter instantiation. A data service exports a possibly infinite set of such parameterized tree pattern queries. This set is succinctly encoded using a *Query Set Specification (QSS)*.

Definition 1 (Query Set Specification). A QSS is a 5-tuple $\langle F, \Sigma, P, S, R \rangle$, where:

- F is a finite set called the *tree fragment names*.
- Σ is a finite set, disjoint from F , called the *element node names*.
- P is a finite set of *productions* of general form $f \rightarrow tf_1, \dots, tf_n$ where $f \in F$ is a tree fragment name and each tf_i is a *tree fragment*. A *tree fragment* is a labeled tree consisting of:
 - *Element nodes* with labels from Σ . Leaf element nodes may be additionally labeled with a parameterized equality predicate of the form $=\#i$, where $\#i$ is a *parameter* and i is an integer.
 - *Tree fragment nodes* n labeled with a name $name(n) \in F$ and an occurrence constraint $occ(n) \in \{1, ?, +, *\}$. Tree fragment nodes can only appear as leaf nodes of a tree fragment. We often omit the occurrence constraint '1'.
 - *Edges* e either of *child* type, denoted by straight lines, or of *descendant* type, denoted by dashed lines.
- $S \in F$ is the start tree fragment name.
- $R \in \Sigma$ is a set called *result node names*. ■

Example. The QSS describing the airline data service from our motivating example is $A = \langle F, \Sigma, P, S, R \rangle$, where $F = \{f_1, \dots, f_9\}$, $\Sigma = \{\text{flights, airline, name, flight, from, to, day, leg, aircraft}\}$, P is the set of productions shown in Figure 2d, $S = f_1$ and $R = \{\text{airline, flight}\}$. ■

A compact visual representation of this QSS is given in Figure 2c, where tree fragments are depicted by shaded boxes with occurrence constraints to their right. This visual representation is the basis of our under-development GUI for specifying QSSs by displaying the XML schema and using drag-and-drop actions.

Given the similarity between QSSs and extended context-free grammars [1], we define the set of parameterized tree pattern queries described by a QSS analogously to the language generated by a grammar. A QSS defines the set of PTPs whose result node is in R and whose pattern is yielded by a sequence of *derivation steps* starting from the start fragment name S . At any step, given a tree fragment node n , the derivation step replaces n with the tree fragments on the right hand side of a production that has n on the left hand side. Depending on the occurrence constraint labeling n , the derivation step might replace it more than once or not at all. More specifically, if $occ(n)=1$, then n is replaced with the corresponding tree fragments exactly once, and they all become children of n 's parent. If $occ(n)=?$, then n is nondeterministically either deleted or relabeled with $occ(n)=1$ before replacement. If $occ(n)=+$, then for a nondeterministically chosen $k \geq 1$, n is replaced with k copies of n , all siblings, with occurrence labels set to 1. If $occ(n)=*$, then n is labeled nondeterministically with $occ(n)=?$ or $occ(n)=+$ first. The parameters introduced in every step are freshly renamed such that their name is unique across the tree fragment obtained so far.

A TP query is *accepted* by a QSS A if and only if it corresponds to an instantiation of the parameters of a PTP query from the set defined by A . We denote with $TP(A)$ the set of TP queries accepted by A .

Example. Figure 3 shows the sequence of derivations steps, denoted by the \Rightarrow symbol, that obtains the corresponding PTP query pq of the TP query q in Figure 2b. Note how the third derivation step replaces f_4 with the corresponding tree fragments, and how the fourth derivation step deletes f_7 and f_9 . After the final derivation step, the node labeled with the `flight` result node name is chosen, thus forming a PTP query pq . When pq 's parameters $[\#1, \#2, \#3, \#4, \#5]$ are instantiated with the constants $['Delta', 'JFK',$

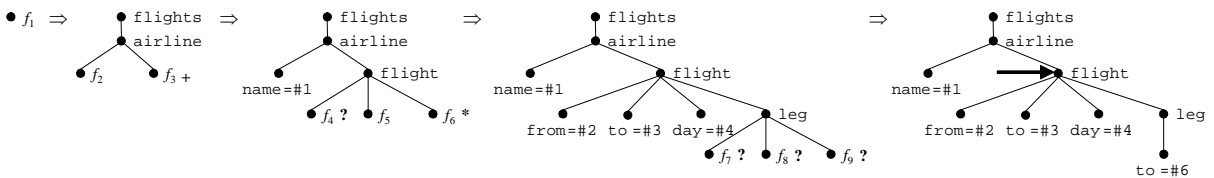


Figure 3. Example Derivation.

'LAX', 'MON', 'LAS'], we obtain the TP query q from Figure 2b. Therefore, q is accepted by A . ■

When the XML Schema is recursive, it describes documents of arbitrary depth. On these documents, there are TP queries of arbitrary pattern height with non-empty answer and it makes sense to export them in a data service.

Despite its fixed size (determined by the XML schema), a QSS can specify such arbitrarily deep TP queries.

Example. The recursive XML Schema in Figure 4a captures the structure of a family tree. Figure 4b shows a TP query that returns the persons found at any depth that are named “Kevin” and were born in “NY” such that at least one of his descendants is married to a person also named “Kevin” and also born in “NY”. Recall that the dotted lines in Figure 4b denote descendant edges.

A QSS that accepts, among others, the corresponding PTP query of the TP query in Figure 4b is shown in Figure 4c. Note the last node, labeled with the tree fragment name f_2 , representing the recursion in the schema. Formally, the above QSS is defined as $B = \langle F, \Sigma, P, S, R \rangle$, where $F = \{f_1, \dots, f_8\}$, $\Sigma = \{\text{familyTree}, \text{person}, \text{name}, \text{place}, \text{spouse}, \text{children}\}$, P is the set of productions shown in Figure 4d, $S = f_1$ and $R = \{\text{person}\}$. Note how the recursion in productions

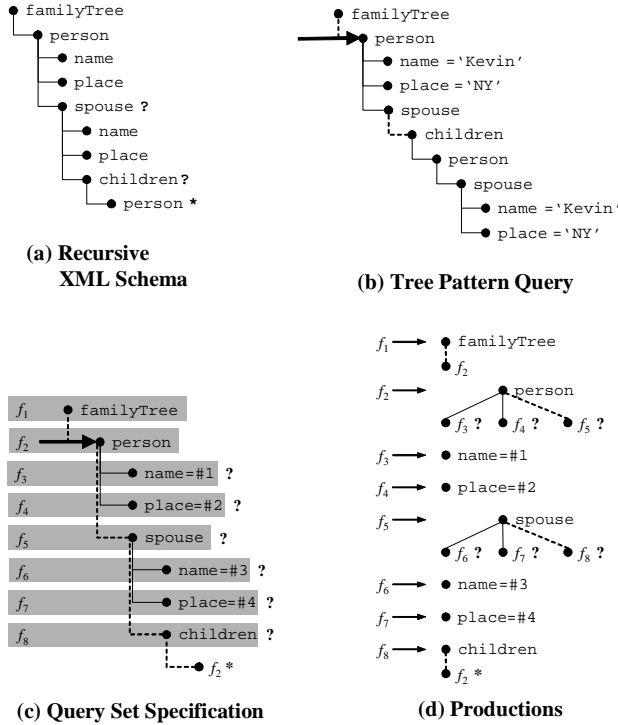


Figure 4. Family Tree Recursive Example.

f_2, f_5 and f_8 allows for derivations of arbitrary length. It is easy to see that the PTP query corresponding to the TP query in Figure 4b is obtained by a sequence of derivation steps using the production associated to f_2 twice. ■

2.2 Reasoning about Data Services

Aside from facilitating the development of applications that are clients of the data service, QSSs allow reasoning about data services. Below are examples of data service properties we would like to verify.

- *Membership of a query in a data service.* The most basic problem is to check if a client TP query q is accepted by a data service described by QSS A , i.e. $q \in \text{TP}(A)$.
- *Subsumption of data services:* given services described by QSSs A_1 and A_2 , check if $\text{TP}(A_1) \subseteq \text{TP}(A_2)$.
- *Totality of a data service:* does the data service described by QSS A accept all possible TP queries?
- *Overlap of data services:* given services described by QSSs A_1 and A_2 , check if $\text{TP}(A_1) \cap \text{TP}(A_2) \neq \emptyset$.

Of course, revisiting the analogy between QSSs and extended context-free grammars, we could reduce these problems to decision problems on grammars. However, while the membership problem can be solved in this way, the other problems in the list reduce to well-known problems that are undecidable even for standard context-free grammars. Fortunately, it turns out that a QSS can be translated to an equivalent top-down nondeterministic unranked tree automaton [3] (the translation is straightforward and omitted due to space limitations). QSSs therefore describe regular tree languages, for which all problems listed above are decidable [3].²

A practically important question is whether a client query can be answered using a finite subset of the queries described by a QSS. This is related to the problem of answering queries using limited query capabilities [18].

2.3 WSDL and XML Syntax

Our proposal for specifying data services is compatible with the standard Web Service Specification Language WSDL [5] in the sense that any QSS can be translated into a WSDL specification.

² This observation should not come as a surprise given the similar result stating that DTDs, who look strikingly similar to extended context-free grammars, actually describe regular tree languages [17].

In general, a WSDL specification describes the format of the messages that a service sends or receives³ using element declarations and type definitions drawn from the XML Schema type system [7]. A WSDL specification describing a data service restricts a general WSDL specification in several ways, since the communication between the agent and the data service is always synchronous and is carried out in a request/response fashion [16]. The input message represents the query received from the service, and the output message the result sent from the service. Both message types are described using the XML Schema type system.

A QSS can be automatically translated into a WSDL specification using the well-known fact that XML Schemas describe regular tree languages themselves. We omit the details of the translation algorithm, but illustrate on an example. This example makes a convincing case for presenting users with a concise and visually intuitive representation such as a QSS instead of the less readable XML syntax of the WSDL.

Example. Appendix A shows the WSDL specification of the QSS from Figure 2c. The schemas that describe the input and the output messages of the data service are imported in the beginning of the specification. The first schema describes the parameterized queries supported by the data service. A QSS is expressed in XML Schema format (QSSX) in order to be contained in a WSDL specification. QSSX is an XML Schema that acceptable TPX queries conform to. The QSSX syntax for the QSS in Figure 2c is shown in Appendix B. The second schema reveals the structure of the underlying database and presents a choice group consisting of the result node names in the result node names set of a QSS. Appendix C shows the XML Schema for the QSS in Figure 2c. As in the case of QSS and QSSX syntax, TP queries need to be expressible in XML format in order to be contained in messages described by a WSDL specification. The XML syntax of TP queries, called TPX, is a subset of XQueryX [12], the XML syntax of XQuery. The TPX query equivalent to the TP query in Figure 2b is given in Appendix D. The XML Schema that defines the TPX language is presented in Appendix E. ■

3. QSSL EXTENSIONS

In the future, QSSL will be enhanced to describe subsets of XQuery expressions beyond XPath ones, as well as additional constraints that restrict the co-occurrence of tree fragments.

³ A WSDL specifies many additional communication details: synchronicity, how sets of messages are grouped into one operation, etc., all of which are orthogonal to our proposal.

In Figure 4c, for example, the QSS only indicates that a parameterized equality predicate on the name and on the birth place of a person can optionally be part of an acceptable PTP query. The QSS does not have the ability to succinctly express that these two predicates are mutually exclusive, or express that at least one of them must be part of an acceptable PTP query. It can achieve the desired effect by explicitly listing all acceptable combinations, but this defeats the purpose of QSSL.

In order to express these constraints, QSS can be enriched with a set of *replacement constrains* including *atLeast*, *atMost* and *xor*.

For example, *atLeast*(1, { f_3 , f_4 }) expresses that in a derivation at least one of f_3 and f_4 must be replaced. *xor*({ f_3 , f_4 }, { f_6 , f_7 }) expresses that either the parameterized predicates on the name and on the place of a person or on the name and on the place of a spouse are part of an acceptable PTP query, but not both.

4. RELATED WORK

In the past, the database community has conducted research on the related problems of answering queries using views [9], capability-based query rewriting [8, 18] and computation of query capabilities [19]. One approach assumes that a source exports a relational view with n attributes, and query capabilities are described as *binding patterns* [9]. Each binding pattern attaches a b (bound) or an f (free) adornment on each attribute of the exported view. Adornment b means that a value for the attribute is required in a query, while f means that a value is optional. The set of adornments can be enriched adding u , where a value for an attribute is not permitted, $c[s]$, where a value for an attribute is required and must be chosen from the set of constants s , and $o[s]$, where a value in s is optional [19]. Note that each binding pattern defines a query template. Query capabilities described as binding patterns are characterized as *negative*, because they restrict the set of all possible queries against the exported view. Wrappers exporting binding patterns are called *thin*, because of their limited functionality to execute the input query against the underlying source.

Another approach describes sets of (parameterized) queries using the expansions of a Datalog program [11, 18]. In this work, it is shown that Datalog is not enough to cover even all yes/no conjunctive queries over a schema. It consequently showed that the RQDL extension can describe large sets, such as the set of all conjunctive queries over a schema. QSSL and data services also attempt to describe the capabilities of sources that support large sets of queries and aim to fuel the research on the problems considered in [8, 9,

18, 19] for the XML data model and the XQuery language [4].

On the industrial level, the effort is focused on turning relational database systems to web services providers by exporting data definition and manipulation operations via web services. These operations are either fixed or parameterized queries expressed in SQL or SQL/XML [6], stored procedures, or functions. Typically, a web service exporting a fixed query takes as input the name of the database operation, and possibly a parameter instantiation, and outputs either an XML document or a serialized object in a given programming language. No schema information of the underlying database is given, either for the input or the output. A list of systems implementing this architecture includes IBM's Document Access Definition Extension (DADx) for DB2 [10], Oracle's Database Web Services specification [13], Microsoft's SQL Server 2000 Web Services Toolkit [21] and BEA's WebLogic Workshop [20]. There is also an effort on consuming web services within the SQL query language, thus integrating relational data with web services.

The W3C Web Services Description Working Group [16] describes usage scenarios that focus on various types of communication using messages and demonstrate how they can be carried out using web services. The technical issues focus on the direction of communication, i.e., request-response, solicit-response or one-way, whether a web service is synchronous or asynchronous and whether it supports conversations, rather than what query capabilities a database exports.

Finally, previous work defines a preliminary and restricted version of QSSL that supports the generation of web-based query forms and reports for semistructured data [15]. Only finite sets of parameterized queries can be encoded no formal semantics is given, and there is not an algorithm that translates a QSS to an XML Schema.

5. REFERENCES

- [1] J. Albert, D. Giammarresi, D. Wood: *Normal Form Algorithms for Extended Context-Free Grammars*, Theoretical Computer Science 267, pp. 35-47, 2001.
- [2] S. Amer-Yahia, S. Cho, L. V. S. Lakshmanan, D. Srivastava: *Minimization of tree pattern queries*, ACM SIGMOD, 2001.
- [3] A. Brüggemann-Klein, M. Murata, and D. Wood: *Regular Tree and Regular Hedge Languages over Unranked Alphabets: Version 1, April 3, 2001*, Technical Report HKUST-TCSC-2001-0, The Hong Kong University of Science and Technology, 2001.
- [4] D. Chamberlin et al.: *XQuery 1.0: An XML Query Language*, W3C Working Draft, 2002.
<http://www.w3.org/TR/xquery/>
- [5] R. Chinnici et al.: *Web Services Description Language (WSDL) v. 1.2*, W3C Working Draft, 2003.
<http://www.w3.org/TR/wsdl12/>
- [6] A. Eisenberg, J. Melton: *SQL/XML is Making Good Progress*, SIGMOD Record 31(2), 2002.
- [7] D. C. Fallside: *XML Schema Part 0: Primer*, W3C Recommendation, 2001.
<http://www.w3.org/TR/xmlschema-0/>
- [8] L. M. Haas, D. Kossmann, E. L. Wimmers, J. Yang: *Optimizing Queries Across Diverse Data Sources*, VLDB, 1997.
- [9] A. Y. Halevy: *Answering Queries Using Views: A Survey*, VLDB Journal 10(4): 270-294, 2001.
- [10] G. Hutchison: *DB2 and Web Services: The Big Picture*, January 2003.
http://www7b.software.ibm.com/dmdd/zones/web_services/
- [11] A. Y. Levy, A. Rajaraman, J. D. Ullman: *Answering Queries Using Limited External Processors*, PODS, 1996.
- [12] A. Malhotra et al.: *XML Syntax for XQuery 1.0 (XQueryX)*, W3C Working Draft 7 June 2001.
<http://www.w3.org/TR/xqueryx>
- [13] K. Mensah, E. Rohwedder: *Oracle9i Database Web Services*, White Paper, November 2002.
http://otn.oracle.com/tech/webservices/htdocs/db_webservices/Database_Web_Services.pdf
- [14] G. Miklau, D. Suciu: *Containment and Equivalence for an XPath Fragment*, PODS, 2002.
- [15] Y. Papakonstantinou, M. Petropoulos, V. Vassalos: *QURSED: Querying and Reporting Semistructured Data*, ACM SIGMOD, 2002.
- [16] W. Sadiq et al.: *Web Service Description Usage Scenarios*, W3C Working Draft, 2002.
<http://www.w3.org/TR/ws-desc-usecases/>
- [17] L. Segoufin, V. Vianu: *Validating Streaming XML Documents*, PODS, 2002.
- [18] V. Vassalos, Y. Papakonstantinou: *Describing and Using Query Capabilities of Heterogeneous Sources*, VLDB, 1997.
- [19] R. Yerneni, C. Li, H. Garcia-Molina, J. Ullman: *Computing Capabilities of Mediators*, ACM SIGMOD, 1999.
- [20] *BEA WebLogic Workshop*
<http://www.bea.com/framework.jsp?CNT=index.htm&FP=/content/products/workshop/>
- [21] *Microsoft SQL Server 2000 Web Services Toolkit*
<http://www.microsoft.com/sql/techinfo/xml/>

APPENDIX

A. WSDL Specification of a Data Service

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="FlightsService"
  targetNamespace="http://airline.wsdl/flights/"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:tns="http://airline.wsdl/flights/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsd1="http://airlineQSSX/"
  xmlns:xsd2="http://airlineSchema/">

  <import location="airlineQSSX.xsd" namespace="http://airlineQSSX/">
  <import location="airlineSchema.xsd" namespace="http://airlineSchema/">

  <message name="queryFlightsRequest">
    <part name="query" type="xsd1:query"/>
    <part name="result" type="xsd2:result"/>
  </message>
  <message name="resultFlightsResponse">
    <part name="result" type="xsd2:result"/>
  </message>

  <portType name="FlightsPortType">
    <operation name="queryFlights" variety="Input-Output">
      <input message="tns:queryFlightsRequest" name="queryFlightsRequest"/>
      <output message="tns:resultFlightsResponse" name="resultFlightsResponse"/>
    </operation>
  </portType>
</definitions>
```

B. QSSX Syntax

```
<?xml version = "1.0" encoding = "UTF-8"?>
<xsd:schema xmlns:xsd = "http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://airlineQSSX/"
  xmlns:qssx = "http://airlineQSSX/">

  <xsd:annotation>
    <xsd:documentation>The root element of a TPX query</xsd:documentation>
  </xsd:annotation>
  <xsd:element name = "query">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:group ref = "qssx:f1"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>

  <xsd:annotation>
    <xsd:documentation>
      The following element groups correspond to the productions of Figure 4d
    </xsd:documentation>
  </xsd:annotation>

  <xsd:group name = "f1">
    <xsd:sequence>
      <xsd:element name = "step">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name = "identifier" fixed = "flights"/>
            <xsd:choice>
              <xsd:sequence>
                <xsd:annotation>
                  <xsd:documentation>
                    The airline element is chosen as the result node
                  </xsd:documentation>
                </xsd:annotation>
                <xsd:element name = "predicatedExpr">
                  <xsd:complexType>
                    <xsd:sequence>
                      <xsd:element name = "identifier" fixed = "airline"/>
                      <xsd:group ref = "qssx:f2"/>
                      <xsd:element name = "predicate" maxOccurs = "unbounded">
                        <xsd:complexType>
                          <xsd:sequence>
```

```

        <xsd:group ref = "qssx:f3"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:sequence>
<xsd:attribute name="axis" use="required" type="xsd:string" fixed = "CHILD"/>
</xsd:complexType>
</xsd:element>
</xsd:sequence>
<xsd:sequence>
  <xsd:annotation>
    <xsd:documentation>
      The flight element is chosen as the result node
    </xsd:documentation>
  </xsd:annotation>
  <xsd:element name = "step">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name = "predicatedExpr">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name = "identifier" fixed = "airline"/>
              <xsd:group ref = "qssx:f2"/>
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
        <xsd:group ref = "qssx:f3" maxOccurs = "unbounded"/>
      </xsd:sequence>
      <xsd:attribute name="axis" use="required" type="xsd:string" fixed = "CHILD"/>
    </xsd:complexType>
  </xsd:element>
</xsd:sequence>
</xsd:choice>
</xsd:sequence>
<xsd:attribute name = "axis" use = "required" type = "xsd:string" fixed = "CHILD"/>
</xsd:complexType>
</xsd:element>
</xsd:sequence>
</xsd:group>

<xsd:group name = "f2">
  <xsd:sequence>
    <xsd:element name = "predicate">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name = "function">
            <xsd:complexType>
              <xsd:sequence>
                <xsd:element name = "identifier" fixed = "name"/>
                <xsd:element name = "constant" type = "xsd:string"/>
              </xsd:sequence>
              <xsd:attribute name="name" fixed="EQUAL"/>
            </xsd:complexType>
          </xsd:element>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
  </xsd:sequence>
</xsd:group>

<xsd:group name = "f3">
  <xsd:sequence>
    <xsd:element name = "predicatedExpr">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name = "identifier" fixed = "flight"/>
          <xsd:group ref = "qssx:f4" minOccurs = "0"/>
          <xsd:group ref = "qssx:f5"/>
          <xsd:group ref = "qssx:f6" minOccurs = "0" maxOccurs = "unbounded"/>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
  </xsd:sequence>
</xsd:group>

<xsd:group name = "f4">

```

```

    <xsd:annotation>
      <xsd:documentation>
        Similar to f2 element group
      </xsd:documentation>
    </xsd:annotation>
    ...
  </xsd:group>

  <xsd:group name = "f5">
    <xsd:annotation>
      <xsd:documentation>
        Similar to f2 element group
      </xsd:documentation>
    </xsd:annotation>
    ...
  </xsd:group>

  <xsd:annotation>
    <xsd:documentation>
      A choice appears in group f6, because leg element might appear as an identifier
      if groups f7, f8 and f9 are not replaced, or as a pedicated expression otherwise
    </xsd:documentation>
  </xsd:annotation>
  <xsd:group name = "f6">
    <xsd:sequence>
      <xsd:element name = "predicate">
        <xsd:complexType>
          <xsd:choice>
            <xsd:element name = "identifier" fixed = "leg"/>
            <xsd:sequence>
              <xsd:element name = "predicatedExpr">
                <xsd:complexType>
                  <xsd:sequence>
                    <xsd:element name = "identifier" fixed = "leg"/>
                    <xsd:group ref = "qssx:f7" minOccurs = "0"/>
                    <xsd:group ref = "qssx:f8" minOccurs = "0"/>
                    <xsd:group ref = "qssx:f9" minOccurs = "0"/>
                  </xsd:sequence>
                </xsd:complexType>
              </xsd:element>
            </xsd:sequence>
          </xsd:choice>
        </xsd:complexType>
      </xsd:element>
    </xsd:sequence>
  </xsd:group>

  <xsd:group name = "f7">
    <xsd:annotation>
      <xsd:documentation>
        Similar to f2 element group
      </xsd:documentation>
    </xsd:annotation>
    ...
  </xsd:group>

  <xsd:group name = "f8">
    <xsd:annotation>
      <xsd:documentation>
        Similar to f2 element group
      </xsd:documentation>
    </xsd:annotation>
    ...
  </xsd:group>

  <xsd:group name = "f9">
    <xsd:annotation>
      <xsd:documentation>
        Similar to f2 element group
      </xsd:documentation>
    </xsd:annotation>
    ...
  </xsd:group>
</xsd:schema>

```


C. Result XML Schema

```
<?xml version = "1.0" encoding = "UTF-8"?>
<xsd:schema xmlns:xsd = "http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://airlineSchema/">
  <xsd:element name = "result">
    <xsd:complexType>
      <xsd:choice>
        <xsd:element ref = "airline"/>
        <xsd:element ref = "flight"/>
      </xsd:choice>
    </xsd:complexType>
  </xsd:element>

  <xsd:element name = "airline">
    <xsd:annotation>
      <xsd:documentation>
        Element declaration as it appears in the data XML Schema
      </xsd:documentation>
    </xsd:annotation>
  </xsd:element>

  <xsd:element name = "flight">...</xsd:element>

  <xsd:element name = "leg">...</xsd:element>
</xsd:schema>
```

D. TPX Query

```
<query xmlns = "http://www.db.ucsd.edu/tpx">
  <step axis = "CHILD">
    <identifier>flights</identifier>
    <step axis = "CHILD">
      <predicatedExpr>
        <identifier>airline</identifier>
        <predicate>
          <function name = "EQUALS">
            <identifier>name</identifier>
            <constant datatype = "CHARSTRING">Delta</constant>
          </function>
        </predicate>
      </predicatedExpr>
      <predicatedExpr>
        <identifier>flight</identifier>
        <predicate>
          <function name = "EQUALS">
            <identifier>from</identifier>
            <constant datatype = "CHARSTRING">JFK</constant>
          </function>
        </predicate>
        <predicate>
          <function name = "EQUALS">
            <identifier>to</identifier>
            <constant datatype = "CHARSTRING">LAX</constant>
          </function>
        </predicate>
        <predicate>
          <function name = "EQUALS">
            <identifier>day</identifier>
            <constant datatype = "CHARSTRING">MON</constant>
          </function>
        </predicate>
        <predicate>
          <predicatedExpr>
            <identifier>leg</identifier>
            <predicate>
              <function name = "EQUALS">
                <identifier>to</identifier>
                <constant datatype = "CHARSTRING">LAS</constant>
              </function>
            </predicate>
          </predicatedExpr>
        </predicate>
      </predicatedExpr>
    </step>
  </step>
</query>
```

E. XML Schema for TPX Syntax

```
<?xml version = "1.0" encoding = "UTF-8"?>
<xsd:schema xmlns:xsd = "http://www.w3.org/2001/XMLSchema"
  targetNamespace = "http://www.db.ucsd.edu/tpx">
  <xsd:group name = "expression">
    <xsd:choice>
      <xsd:element ref = "constant"/>
      <xsd:element ref = "function"/>
      <xsd:element ref = "predicatedExpr"/>
      <xsd:element ref = "step"/>
      <xsd:element ref = "identifier"/>
    </xsd:choice>
  </xsd:group>
  <xsd:element name = "query">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:group ref = "expression"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name = "predicatedExpr">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:group ref = "expression"/>
        <xsd:element ref = "predicate" maxOccurs = "unbounded"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name = "predicate">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:group ref = "expression"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name = "identifier" type = "xsd:string"/>
  <xsd:element name = "constant">
    <xsd:complexType>
      <xsd:simpleContent>
        <xsd:extension base = "xsd:string">
          <xsd:attribute name = "datatype" type = "xsd:string"/>
        </xsd:extension>
      </xsd:simpleContent>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name = "function">
    <xsd:complexType>
      <xsd:choice minOccurs = "0" maxOccurs = "unbounded">
        <xsd:group ref = "expression"/>
      </xsd:choice>
      <xsd:attribute name = "name" use = "required" type = "xsd:string" fixed = "EQUAL"/>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name = "step">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:group ref = "expression"/>
        <xsd:group ref = "expression"/>
      </xsd:sequence>
      <xsd:attribute name = "axis" use = "required">
        <xsd:simpleType>
          <xsd:restriction base = "xsd:NMTOKEN">
            <xsd:enumeration value = "CHILD"/>
            <xsd:enumeration value = "DESCENDANT"/>
            <xsd:enumeration value = "SLASHSLASH"/>
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:attribute>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```