Indexing Dataspaces

Xin Dong
University of Washington
Seattle, WA 98195
lunadong@cs.washington.edu

Alon Halevy
Google Inc.
Mountain View, CA 94043
halevy@google.com

ABSTRACT

Dataspaces are collections of heterogeneous and partially unstructured data. Unlike data-integration systems that also offer uniform access to heterogeneous data sources, dataspaces do not assume that all the semantic relationships between sources are known and specified. Much of the user interaction with dataspaces involves exploring the data, and users do not have a single schema to which they can pose queries. Consequently, it is important that queries are allowed to specify varying degrees of structure, spanning keyword queries to more structure-aware queries.

This paper considers indexing support for queries that combine keywords and structure. We describe several extensions to inverted lists to capture structure when it is present. In particular, our extensions incorporate attribute labels, relationships between data items, hierarchies of schema elements, and synonyms among schema elements. We describe experiments showing that our indexing techniques improve query efficiency by an order of magnitude compared with alternative approaches, and scale well with the size of the data.

Categories and Subject Descriptors: H.3.1: Content Analysis and Indexing

General Terms: Performance, Experimentation

Keywords: Dataspace, Indexing, Heterogeneity

1. INTRODUCTION

Dataspaces are large collections of heterogeneous and partially unstructured data [24]. Unlike data integration systems that also offer uniform access to heterogeneous data sources, dataspaces do not assume that all the semantic relationships between sources are known and have been specified. In some cases, semantic relationships are unknown because of the sheer number of sources involved or the lack of people skilled in specifying such relationships. In other cases, not all semantic relationships are necessary in order to offer the services of interest to users. A dataspace system typically employs automatic methods that try to extract some of the semantic relationships, but these results are approximate at best. The goal of dataspaces is to provide *useful* services whenever possible, and assist users in creating tighter semantic relationships when they see fit. Scenarios in which we want to manage dataspaces include personal data on one's desktop, collections of data sources in enterprises, government agencies, collaborative scientific projects, digital libraries, and large collections of structured sources on the Web.

We are building a system that enables users to interact with dataspaces through a search and query interface. In doing so, we are keeping two goals in mind. First, much of the interaction with such a system is of exploratory nature—the user is getting to know the data and its structure. Second, since there are many disparate data sources, the user cannot query the data using a particular schema. To support these two goals, it is important that users are able to use varying degrees of structure in their queries, ranging from keyword queries to structure-aware queries. Furthermore, it is beneficial that the system returns *possibly related* data in answers to queries and not only the data that strictly satisfy the query.

We capture these novel querying needs with two types of queries: *predicate queries* and *neighborhood keyword queries*. A predicate query allows the user to specify both keywords and simple structural requirements, such as "a paper with title 'Birch', authored by 'Raghu', and published in 'Sigmod 1996'". A neighborhood keyword query is specified by a set of keywords, but differs from traditional keyword search in that it also explores *associations* between data items, and so it leverages additional structure that may exist in the data or may have been automatically discovered. For example, searching for "Birch" returns not only the papers and presentations that mention the BIRCH project, but also people working on BIRCH and conferences in which BIRCH papers have been published.

This paper considers indexing support for predicate queries and neighborhood keyword queries. Broadly speaking, existing methods either build a separate index for each attribute in each data source to support structured queries on structured data, or create an *inverted list* to support keyword search on unstructured data. Consequently, as we shall show, they fall short in the context of queries that combine keywords and structure. The area in which indexing structure and keywords has received most attention is in the context of XML. However, the techniques proposed for XML indexing fall short in our context for two reasons. First, the

XML techniques typically rely on encoding the parent-child and ancestor-descendant relationships in an XML tree; however, the relationships in a dataspace do not fit this model. Furthermore, most XML indexing methods build multiple indexes; as we show in our experiments, visiting multiple indexes to answer a predicate query or a neighborhood keyword query can be quite time-consuming.

We propose to capture both text values and structural information using an extended inverted list. Our index augments the text terms in the inverted list with labels denoting the structural aspects of the data such as (but not limited to) attribute tags and associations between data items. When an attribute tag is attached to a keyword, it means that this keyword appears as a value for that attribute. When an association tag is attached to a keyword, it means that this keyword appears in an associated instance. We explore several methods for extending inverted lists and describe experiments that validate the utility of our extensions.

Our ultimate goal is to support robust indexing of loosely-coupled collections of data in the presence of varying degrees of heterogeneity in schema and data, such that we can efficiently answer queries that combine keywords and simple structural requirements. This paper makes the following contributions towards this goal:

- We introduce a framework that indexes heterogeneous data from multiple sources through a (virtual) central triple store, so as to support queries that combine keywords and structural specification.
- We describe extensions to inverted lists that capture attribute information and associations between data items.
- We show how our techniques can be extended to incorporate various types of heterogeneity, including synonyms and hierarchies of attributes and associations.
- We describe experimental results showing that our techniques improve search efficiency by an order of magnitude and perform better than competing alternatives. In addition, the experiments show that our technique scales well and supports efficient index updates.

Section 2 overviews our indexing framework and formally defines our problem. Section 3 describes how to extend inverted lists to support attribute and association information. Section 4 shows further extensions for attribute hierarchies and synonyms. Section 5 presents experimental results. Section 6 discusses related work and Section 7 concludes.

2. PROBLEM DEFINITION

We begin by describing our problem setting and the types of queries we aim to support. We also give a brief overview of inverted lists.

2.1 Indexing Heterogeneous Data

Our goal is to support efficient queries over collections of heterogeneous data that are *not* necessarily semantically integrated as in data-integration systems. The scenario we use throughout our discussion is representative of data that may be extracted from multiple sources, some of which are only partially structured or not structured at all (e.g., DBLife [18], Semex [19], iMeMex [8]). The scenario includes a collection of files of various types (e.g., LATEX and BIBTEX files, Word documents, Powerpoint presentations, emails and contacts,

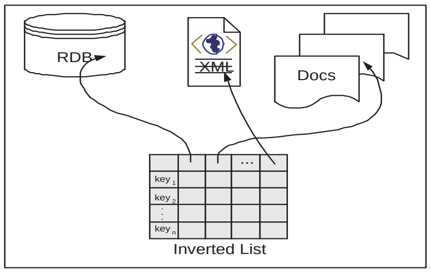


Figure 1: We build an index over a collection of heterogeneous data. Our index is an inverted list where each row represents a keyword and each column represents a data item from the data sources.

and webpages in the web cache), as well as some structured sources such as spreadsheets, XML files and databases. We extract associations between disparate items in the unstructured data and also from structured data sources.

Answers to queries are data items from the sources, such as files, rows in spreadsheets, tuples in relational databases or elements in XML data. Hence, our goal is to build an index whose leaves are references to data items in the individual sources (see Figure 1).

To build such an index, we model the data from different data sources universally as a set of triples, which we refer to as a *triple base*. Each *triple* is either of the form (instance, attribute, value) or of the form (instance, association, instance). Thus, a triple base describes a set of instances and associations. An instance corresponds to a real-world object and is described by a set of *attributes*, for each of which there might be multiple values. An association is a relationship between two instances. We assume that associations are directional, and in particular, the two directions of a particular association may be named differently (e.g., author and authoredPaper).

The items in our index correspond to instances extracted from the data sources using a variety of methods. For example, we extract instances and associations from tuples in a relational database by trying to guess the E/R model that may lead to the schema. For example, if the key of a table consists of multiple attributes and each is a foreign key to another table, we consider tuples in the table as representing associations. As other examples, we adapt techniques from [23, 5, 19] to extract relationships between structured and unstructured data. Note that these extractions are imprecise in nature, and so our querying mechanisms and indexing techniques need to allow more flexibility. Our paper omits the details of extraction and focuses on the indexing aspect.

To further accommodate heterogeneity, our triple base also models *synonyms* among attribute and association names (as well as an association being synonymous with an attribute) and *hierarchies*. Hierarchies can be of *sub-property* type (e.g., father being a sub-property of parent) or *sub-field* type (e.g., city being a sub-field of address), and we do not distinguish between them. Heterogeneity often arises in the way data sources model structure hierarchies (e.g., different ways of modeling addresses and people); therefore, it is important for our indexing mechanisms to be hierarchy-aware.

EXAMPLE 1. *Consider the triple base depicted in Figure 2. It contains three* Person *instances* p_1, p_2, p_3*, one* Article *in-*

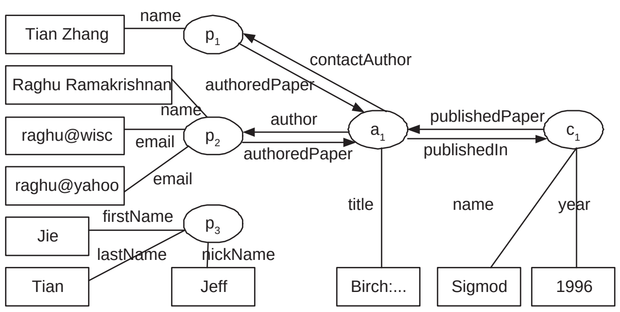


Figure 2: An example triple base. The ellipses represent instances and the rectangles represent attribute values. Undirectional edges represent attributes and directional edges represent associations.

stance a_1, and one Conference *instance c_1. For each instance we list the attribute values and associated instances. For example,* Paper *a_1 has title "Birch:..."; it is associated with* Person *instances p_1 and p_2, and* Conference *instance c_1. Here we assume that the attributes* firstName, lastName *and* nickName *are sub-attributes of* name*, and the association* contactAuthor *is a sub-association of* author. □

2.2 Querying Heterogeneous Data

We aim to support queries over heterogeneous data sources where users are not aware of the existing structure. Hence, we support queries that enable the user to specify as much structure as she can, including none at all. The first type of queries, called *predicate queries*, describes the desired instances by a set of predicates, each specifying an attribute value or an associated instance.

Definition 2.1. *A* predicate query *contains a set of* predicates. *Each predicate is of the form $(v, \{K_1, \ldots, K_n\})$, where v is called a* verb *and is either an attribute name or an association name, and $K_1, \ldots, K_n$ are keywords.*

The predicate is called an attribute predicate *if v is an attribute, and an* association predicate *if v is an association.*

The semantics of predicate queries is as follows. The returned instances need to satisfy at least one predicate in the query. An instance satisfies an attribute predicate if it contains at least one of $\{K_1, \ldots, K_n\}$ in the values of attribute v or sub-attributes of v. An instance o satisfies an association predicate if there exists $i, 1 \leq i \leq n$, such that o has an association v or sub-association of v with an instance o' that has an attribute value K_i. □

We note that we can also express conjunctions of predicates in our language, but the details are irrelevant to our discussion.

Example 2. *The query "Raghu's Birch paper in Sigmod 1996" can be described with the following three predicates. The query is satisfied by instance a_1 in our example.*

```
(title 'Birch'), (author 'Raghu'),
(publishedIn '1996 Sigmod')
```
□

In practice, users can specify predicate queries in two ways. First, they can specify a query through a user interface featuring drop-down menus that show all existing attribute or association labels. Second, they can compose the query in a certain syntax (such as the one shown in Example 2), specifying attribute or association labels that they know (such as those in data sources familiar to them). In general, our querying is aimed to be more forgiving in cases where users do not know the schema. For example, we support synonym terms, and we don't require knowledge of attribute hierarchies—users can specify terms anywhere in a hierarchy.

The second type of queries, called *neighborhood keyword queries*, extends keyword search by taking associations into account.

Definition 2.2. *A* neighborhood keyword query *is a set of keywords, $K_1, \ldots, K_n$. An instance satisfies a neighborhood keyword query if either of the following holds:*

- *The instance contains at least one of $\{K_1, \ldots, K_n\}$ in attribute values. In this case we call it a* relevant instance.
- *The instance is associated (in either direction) with a relevant instance. In this case we call it an* associated instance. □

Example 3. *Consider the query "Birch". Instance a_1 is a relevant instance as it contains "Birch" in the* title *attribute, and p_1, p_2, and c_1 are associated instances.* □

Predicate queries and neighborhood keyword queries are different from traditional structured queries in that the user can specify keywords instead of precise values, and provide only approximate structure information. For example, the query in Example 2 does not specify if "Raghu" should occur in an author attribute, or in an author sub-element, or in the attribute of another tuple that can be joined with the returned instance. These types of queries are also different from keyword search in that query answering explores the structure of the data to return associated relevant instances. In the rest of the paper, we introduce an index designed to support such queries.

Clearly, a significant part of answering the above queries is intelligent ranking of the results. Our system uses a combination of methods, including ranking results that match *several* keywords in a predicate more highly, weighting associations differently when ranking associated instances, applying PageRank on the association network. The rest of the paper focuses on the indexing aspects of query answering.

2.3 Inverted Lists

Our index is based on extending inverted lists, a technique widely used in Information Retrieval. In Section 6 we explain why we chose inverted lists over more database-oriented techniques. We now quickly review how an inverted list indexes a set of instances in a triple base by keyword.

Conceptually, an inverted list is a two-dimensional table, where the i-th row represents *indexed keyword K_i* and the j-th column represents instance I_j. The cell at the i-th row and j-th column, denoted (K_i, I_j), records the number of occurrences, called *occurrence count*, of keyword K_i in the attributes of instance I_j. If the cell (K_i, I_j) is not zero, we say instance I_j is *indexed on K_i*. The keywords are ordered in alphabetic order, and the instances are ordered by their identifiers. Table 1 shows the inverted list for our example triple base.

Table 1: **The inverted list for the example triple base.**

	a_1	c_1	p_1	p_2	p_3
1996	0	1	0	0	0
birch	1	0	0	0	0
jeff	0	0	0	0	1
jie	0	0	0	0	1
raghu	0	0	0	3	0
ramakrishnan	0	0	0	1	0
sigmod	0	1	0	0	0
tian	0	0	1	0	1
wisc	0	0	0	1	0
yahoo	0	0	0	1	0
zhang	0	0	1	0	0

Note that inverted lists, as described above, do not capture any structure information: in the example inverted list, we cannot tell that "tian" occurs as p_1's name (actually, first name) and p_3's lastName. In the subsequent sections we describe extensions to inverted lists that enable efficiently answering predicate queries and neighborhood keyword queries.

In practice, an inverted list is seldom stored as a matrix. There are multiple ways to store an inverted list [4], such as a sorted array, a prefix B-tree or a Patricia trie. In addition, [42] describes techniques for compression of inverted lists. The extensions we describe are orthogonal to these physical implementations.

3. INDEXING STRUCTURE

This section describes how we index attributes and associations along with keywords to support predicate queries. We consider hierarchies in the next section.

3.1 Indexing Attributes

Consider an attribute predicate $(A, \{K_1, \ldots, K_n\})$ in a predicate query. Instances satisfy the predicate if they contain some of the keywords $K_1, \ldots, K_n$ in their A attribute. To handle attribute predicates efficiently, our index should tell us which attributes contain a given keyword.

There are several ways to capture attribute types in indexing. One option is to build an index for each attribute, but as we shall show in the experiments, it can introduce a significant overhead to the index structure. Another option is to specify the attribute name in the cells of the inverted list. For example, the cell ("tian", p_1) in Table 1 could be modified to record "name:1". However, this method would considerably complicate query answering. The solution we propose captures attribute names with the indexed keywords to save both index space and lookup time.

Attribute inverted lists (ATIL): We create *an attribute inverted list* (see Table 2) as follows. Whenever the keyword k appears in a value of the a attribute, there is a row in the inverted list for $k//a//$. For each instance I, there is a column for I. The cell ($k//a//$, I) records the number of occurrences of k in I's a attributes. (We assume // is a string reserved for indexing purposes only, and any other delimiter that never occurs in the indexed keywords works too.)

To answer a predicate query with attribute predicate (A, $\{K_1, \ldots, K_n\}$), we only need to do keyword search for $\{K_1//A//, \ldots, K_n//A//\}$. For example, to answer the attribute predicate "lastName, 'Tian'", we transform it into a keyword query "tian//lastName//". In Table 2, the search will yield p_3 but not p_1.

Table 2: **The ATIL: each indexed keyword is a concatenation of a keyword and an attribute.**

	a_1	c_1	p_1	p_2	p_3
1996//year//	0	1	0	0	0
birch//title//	1	0	0	0	0
jeff//nickName//	0	0	0	0	1
jie//firstName//	0	0	0	0	1
raghu//email//	0	0	0	2	0
raghu//name//	0	0	0	1	0
ramakrishnan//name//	0	0	0	1	0
sigmod//name//	0	1	0	0	0
tian//lastName//	0	0	0	0	1
tian//name//	0	0	1	0	0
wisc//email//	0	0	0	1	0
yahoo//email//	0	0	0	1	0
zhang//name//	0	0	1	0	0

3.2 Indexing Associations

Consider the association predicate $(R, \{K_1, \ldots, K_n\})$. Instances satisfy the predicate if they have associations of type R with instances that contain some of the keywords $K_1, \ldots, K_n$ in attribute values.

One naive solution here would be to perform a keyword search on keywords $\{K_1, \ldots, K_n\}$ and find a set of instances $\{I_1, \ldots, I_m\}$ that contain these keywords. Then, for each instance $I_k, k \in [1, m]$, we find associated instances. This approach can be very expensive for two reasons. First, when m is large, iteratively finding associated instances for each I_k can be expensive. Second, a returned instance can be associated with one or more instances in $\{I_1, \ldots, I_m\}$. Ranking the returned results requires counting the number of associated instances for each result, which can be expensive. We offer a solution that extends inverted lists to also capture association information, thereby avoiding the expensive traversal of the triple base.

Attribute-association inverted lists (AAIL): We index association information as follows. Suppose the instance I has an association r with instances $I_1, \ldots, I_n$ in the triple base, and each of $I_1, \ldots, I_n$ has the keyword k in one of its attribute values. The inverted list will have a row for $k//r//$ and a column I. The cell ($k//r//$, I) has the value n.

An inverted list that captures both attribute and association information is called an *attribute-association inverted list (AAIL)* (see Table 3). Given an association predicate $(R, \{K_1, \ldots, K_n\})$, we can answer it by posing the keyword query $\{K_1//R//, \ldots, K_n//R//\}$ over the AAIL. For example, when searching for "Raghu's papers", the query contains an association predicate "author 'Raghu'" and so we look up keyword "raghu//author//". Based on the AAIL in Table 3, we return instance a_1.

Integrating association information in the inverted list increases the size of the index. However, in most applications when the size of the indexed data increases, the average number of associated instances for each instance increases only slightly or even remains the same, so the index typically grows linearly with the size of the data. Our experiments show that adding association information into an ATIL (to obtain AAIL) slows down answering attribute predicates only slightly, but it speeds up answering associ-

Table 3: The AAIL: each index keyword is a concatenation of a keyword and an attribute name, or a keyword and an association name.

	a_1	c_1	p_1	p_2	p_3
1996//publishedIn//	1	0	0	0	0
1996//year//	0	1	0	0	0
birch//authoredPaper//	0	0	1	1	0
birch//publishedPaper//	0	1	0	0	0
birch//title//	1	0	0	0	0
jeff//nickName//	0	0	0	0	1
jie//firstName//	0	0	0	0	1
raghu//author//	1	0	0	0	0
raghu//email//	0	0	0	2	0
raghu//name//	0	0	0	1	0
ramakrishnan//author//	1	0	0	0	0
ramakrishnan//name//	0	0	0	1	0
sigmod//name//	0	1	0	0	0
sigmod//publishedIn//	1	0	0	0	0
tian//contactAuthor//	1	0	0	0	0
tian//lastName//	0	0	0	0	1
tian//name//	0	0	1	0	0
wisc//author//	1	0	0	0	0
wisc//email//	0	0	0	1	0
yahoo//author//	1	0	0	0	0
yahoo//email//	0	0	0	1	0
zhang//contactAuthor//	1	0	0	0	0
zhang//name//	0	0	1	0	0

ation predicates by an order of magnitude compared with the naive method.

It is interesting to distinguish our association index from join indexes [39], where a precomputed join $R \bowtie S$ is materialized as a separate table and two copies of the table are maintained, one clustered on R's key columns and the other clustered on S's key columns. Our association index can be viewed as a union of the original data and multiple join results. This index structure enables us to count the occurrences of keywords and the numbers of associated instances with one scanning of the index.

Finally, note that a k-ary association can be modeled as an instance that is related to the k instances involved in the association. Our indexing method can be easily extended to this case and we omit the details for space considerations.

4. INDEXING HIERARCHIES

We now consider answering predicate queries in the presence of hierarchies. For example, for the query "name 'Tian'", we wish to return instances p_1 and p_3, rather than only p_1.

A simple method to incorporate hierarchies would be to first find all descendants of the name attribute (in this example, they are firstName, lastName and nickName), then expand the keyword query by considering also descendant attributes (so search "tian//name// OR tian//firstName// OR tian//lastName// OR tian//nickName//"). However, this method requires multiple index lookups and thus can be expensive.

Our solution is based on integrating the hierarchy information into the index structure. We begin by describing two possible solutions, and then combine their features and introduce a hybrid indexing scheme. For ease of explanation, we consider only attribute hierarchies, but the same principle applies to association hierarchies. We assume that each attribute has at most a single parent attribute. This covers most cases in practice and the approach can be easily extended to multiple-inheritance cases.

Table 4: The Dup-ATIL: the difference from Table 2 is highlighted using bold font.

	a_1	c_1	p_1	p_2	p_3
1996//year//	0	1	0	0	0
birch//title//	1	0	0	0	0
jeff//name//	**0**	**0**	**0**	**0**	**1**
jeff//nickName//	0	0	0	0	1
jie//firstName//	0	0	0	0	1
jie//name//	**0**	**0**	**0**	**0**	**1**
raghu//email//	0	0	0	2	0
raghu//name//	0	0	0	1	0
ramakrishnan//name//	0	0	0	1	0
sigmod//name//	0	1	0	0	0
tian//lastName//	0	0	0	0	1
tian//name//	0	0	1	0	**1**
wisc//email//	0	0	0	1	0
yahoo//email//	0	0	0	1	0
zhang//name//	0	0	1	0	0

4.1 Index with Duplication

Our first solution duplicates a row that includes an attribute name for each of its ancestors in the hierarchy.

Attribute inverted lists with duplication (Dup-ATIL): We construct a Dup-ATIL as follows. If the keyword k appears in the value of attribute a_0, and a is an ancestor of a_0 in the hierarchy (a could also be a_0), then there is a row $k//a//$. The cell $(k//a//, I)$ records the number of occurrences of k in values of the a attribute and a's sub-attributes of I. We answer a predicate query with the Dup-ATIL in the same way as we use the ATIL.

EXAMPLE 4. *Table 4 shows the Dup-ATIL for our example. Consider instance p_3. We index p_3 not only on* "jie//firstName//", "tian//lastName//", *and* "jeff// nickName//"*, but also on* "jie//name//", "tian//name//", *and* "jeff//name//"*. Thus, in the inverted list, row* "tian// name//" *also records one occurrence for instance p_3, and there are new rows* "jie//name//" *and* "jeff//name//"*, each recording one occurrence for instance p_3.*

*Now consider searching a person with name "Tian". We transform the query "*name *'Tian'" into keyword search* "tian//name//" *and return instances p_1 and p_3.* □

On the one hand, a Dup-ATIL has the benefit of simple query answering, but on the other hand, it may considerably expand the size of the index because of the duplication. The size of the Dup-ATIL will be especially affected if the attribute hierarchy contains long paths from the root attribute to the leaf attributes and most values in the triple base belong to leaf attributes.

4.2 Index with Hierarchy Path

We now introduce a second solution, which does not affect the number of rows in the inverted list. Instead, the keyword in every row includes the entire hierarchy path.

Attribute inverted lists with hierarchies (Hier-ATIL): We construct a Hier-ATIL by extending the attribute inverted list as follows (see Figure 5). Let $a_0, \ldots, a_n$ be attributes such that for each $i \in [0, n-1]$, attribute a_i is the super-attribute of a_{i+1}, and a_0 does not have super-attribute. We call $a_0// \ldots //a_n//$ a *hierarchy path* for attribute a_n. For each keyword k in the value of attribute a_n, there is a row for $k//a_0// \ldots //a_n//$. For each instance

Table 5: The Hier-ATIL: each row represents a concatenation of a keyword and a *hierarchy path*. The difference from Table 2 is highlighted using bold font.

	a_1	c_1	p_1	p_2	p_3
1996//year//	0	1	0	0	0
birch//title//	1	0	0	0	0
jeff//name//nickName//	0	0	0	0	1
jie//name//firstName//	0	0	0	0	1
raghu//email//	0	0	0	2	0
raghu//name//	0	0	0	1	0
ramakrishnan//name//	0	0	0	1	0
sigmod//name//	0	1	0	0	0
tian//name//	0	0	1	0	0
tian//name//lastName//	0	0	0	0	1
wisc//email//	0	0	0	1	0
yahoo//email//	0	0	0	1	0
zhang//name//	0	0	1	0	0

I, there is a column for I. The cell ($k//a_0//\dots//a_n//$, I) records the number of occurrences of k in I's a_n attributes.

A Hier-ATIL captures the hierarchy information using hierarchy paths, which have a nice feature: the hierarchy path of an attribute A is a prefix of the hierarchy paths of A's descendant attributes. Thus, we can transform an attribute predicate into a prefix search. Specifically, consider a query predicate $(A, \{K_1, \dots, K_n\})$. We transform it into a prefix search: $K_1//A//*, \dots, K_n//A//*$. For example, we can transform the query predicate "name 'Tian'" into a prefix search "tian//name//*" and so return both p_1 and p_3.

Since the indexed keywords in an inverted list are ordered, we can answer a prefix query easily. To look up prefix $P*$, we first locate the first row where the keyword is P or starts with P. We then scan the succeeding rows until reaching an indexed keyword that does not start with P, and we accumulate the occurrence counts in these rows for each instance.

Unlike Dup-ATIL, building a Hier-ATIL does *not* increase the number of indexed keywords. Although it can lengthen many of the indexed keywords, real indexing systems typically record a keyword only by the difference from its previous keyword (for example, given a keyword k_1 and a succeeding keyword k_2, where the maximal common prefix of k_1 and k_2 is p, an index can record k_2 by the length of p and k_2's suffix that differs from k_1). Thus, building a Hier-ATIL introduces only a small overhead. However, with Hier-ATIL we need to answer a predicate query by transforming it into a prefix search, which can be more expensive than a keyword search. Answering a prefix search is especially expensive when a keyword occurs in many different attributes with common ancestors.

It is interesting to compare our approach with the one proposed in [16] in the context of indexing XML data, where the focus was on answering queries with path expressions. Whereas we index a keyword *followed by* the hierarchy path, [16] indexes an XPath with the keyword *in the end*. Our approach has two advantages in our context. First, attribute keywords have much higher variety than attribute names and thus are more selective. Second, in the presence of attribute hierarchies, using our index we can transform a query predicate into a prefix search (e.g., "tian//name//*"), but using their index we need to transform it into a general regular-expression query (e.g., "name/*/tian//"), which can be much more expensive to answer.

Table 6: The Hybrid-ATIL with threshold t=1: the difference from Table 5 is the row for "tian//name////".

	a_1	c_1	p_1	p_2	p_3
1996//year//	0	1	0	0	0
birch//title//	1	0	0	0	0
jeff//name//nickName//	0	0	0	0	1
jie//name//firstName//	0	0	0	0	1
raghu//name//	0	0	0	1	0
raghu//email//	0	0	0	2	0
ramakrishnan//name//	0	0	0	1	0
sigmod//name//	0	1	0	0	0
tian//name////	**0**	**0**	**1**	**0**	**1**
tian//name//lastName//	0	0	0	0	1
wisc//email//	0	0	0	1	0
yahoo//email//	0	0	0	1	0
zhang//name//	0	0	1	0	0

```
procedure Lookup($L, P$) return $S$
//$L$ is a Hybrid-ATIL; $P$ is a prefix to look up;
//Return $S$, an array summarizing for each instance the
      occurrences of keywords with prefix $P$;
  Initialize each value of $S[]$ to 0;
  Locate the first keyword $\bar{K} = K +'' //''$ with prefix $P$;
  while $P$ is the prefix of $\bar{K}$
    Update $S$ according to the row for $\bar{K}$;
    if ($\bar{K}$ ends with "////")
      if $K = P$ return $S$;
      else Skip all succeeding keywords with prefix $K$;
    Read the next keyword $\bar{K} = K +'' //''$;
  return $S$;
```

Figure 3: The algorithm for looking up a prefix in a Hybrid-ATIL.

4.3 Hybrid Index

The two solutions we have proposed have complimentary benefits: Dup-ATIL is more suitable for the cases where a keyword occurs in many attributes with common ancestors, and Hier-ATIL is more suitable for the cases where a keyword occurs in only a few attributes with common ancestors. We now describe a hybrid indexing scheme that combines the strengths of both methods.

Hybrid attribute inverted list (Hybrid-ATIL): The goal of a Hybrid-ATIL is to build an inverted list that can answer any prefix search (ending with "//") by reading no more than t rows, where t is a threshold given as input to the algorithm.

We build the Hybrid-ATIL by starting with the Hier-ATIL and successively adding *summary rows*, using a strategy we shall describe shortly. The indexed keyword in a summary row is of the form $p//$, where $p = k//a_0//\dots//a_l//$, k is a keyword, and $a_0//\dots//a_l//$ is a hierarchy path for attribute a_l. Rows whose indexed keywords start with p are said to be *shadowed* by the summary row $p//$. Note that keywords in summary rows end with an additional // to be distinguished from ordinary rows. The cell ($p//$, I) has the sum of the occurrence counts of I in $p//$'s shadowed rows. The Hybrid-ATIL with threshold $t = 1$ for the example triple base is shown in Table 6.

To answer a prefix query of the form $k//a_1//\dots//a_m//*$, we look at all the rows with prefix $k//a_1//\dots//a_m//$ *except* those shadowed by summary rows. Figure 3 shows the algorithm for prefix lookup in a Hybrid-ATIL.

EXAMPLE 5. *Consider two queries on the example triple* *base. The query predicate "*name *'Jeff' " is transformed into prefix search* "jeff//name//*"*. In Table 6, only keyword* "jeff//name//nickName//" *contains this prefix, so we return instance* p_3.

*The query predicate "*name *'Tian' " is transformed into prefix search* "tian//name//*"*. As the Hybrid-ATIL contains a summary row with indexed keyword* "tian//name////"*, we can directly return instances* p_1 *and* p_3 *without considering other keywords.*

In both cases, we read no more than one row (recall that $t = 1$ *for the index) to answer a prefix search.* □

Creating the Hybrid-ATIL: We begin with the Hier-ATIL and add summary rows until none can be added. We denote by $Ans(p)$ the number of rows we need to examine to answer a prefix query p. We create a summary row for a prefix p if $Ans(p) > t$ and there is no p', such that p is a prefix of p' and $Ans(p') > t$. If we add a summary row for $p//$, we remove the p row from the inverted list if one exists.

In Table 5, $Ans(\text{"}tian//name//'') = 2$. Therefore, with threshold $t = 1$, the row "tian//name//" would be replaced by a summary row, as shown in Table 6. We can actually show that we can construct the Hybrid-ATIL from the Hier-ATIL with a single pass over the keyword entries.

The construction of the Hybrid-ATIL guarantees $Ans(p) \leq t$ for any prefix p. Note that adding summary rows can increase the size of the index. However, by choosing an appropriate threshold t we can trade-off index size (so the prefix-lookup time) and occurrence-accumulation time.

Note that the Information Retrieval community has proposed other types of indexes for regular-expression matching, such as *suffix tree* [3], which indexes all suffixes of each document, and *multigram index* [14], which creates k-gram indexes for reasonable k values (e.g., $k = 2, 3, \ldots, 10$). In addition, HYB [6] and KISS [30] have recently been proposed for general prefix matching. Compared with these approaches, our index is oriented to prefix matching where we know the exact prefix delimiters ("//" in our case), thus indexing and searching can be more efficient.

4.4 Schema-Level Synonyms

Accommodating different hierarchical structures is already an important step towards supporting data heterogeneity in our indexing mechanism. We now briefly describe how our techniques easily handle two other forms of heterogeneity.

The first form of heterogeneity is where an association in one source is an attribute in another. For example, author can be an attribute of a Paper instance with author names as attribute values, or an association between Paper instances and Person instances. Since our index does not distinguish attributes and associations in the indexed keywords, it naturally incorporates this kind of heterogeneity.

The second type of heterogeneity, *term heterogeneity*, is where different terms represent the same attribute or association. For example, author and authorship can describe the same association.

To accommodate term heterogeneity, we assume we have a synonym table for attribute and association names. If attribute a is referred to as $a_1, \ldots, a_n$ in different data sources, we choose the *canonical* name of a as one of $a_1, \ldots, a_n$. We note that the synonyms are either given to us or are derived using schema-matching techniques, and hence will typically be approximate.

Table 7: The KIL with threshold $t = 1$: to save space, we only show the rows where the indexed keywords start with "birch".

	a_1	c_1	p_1	p_2	p_3
birch////	1	1	1	1	0
birch//authoredPaper//	0	0	1	1	0
birch//publishedPaper//	0	1	0	0	0
birch//title//	1	0	0	0	0

In our index, when a keyword k appears in a value of the a_i attribute, there is a row in the inverted list for $k//a//$. For each instance I, there is a column for I. The cell $(k//a//, I)$ records the number of occurrences of k in I's $a_1, \ldots, a_n$ attributes.

To answer a predicate query with attribute predicate $(a_i, \{K_1, \ldots, K_n\}), i \in [1, n]$, we transform it into a keyword search for $\{K_1//a//, \ldots, K_n//a//\}$. In our example, if we consider author as a canonical name for author and authorship, the attribute predicate "authorship, 'Tian'" will be transformed into "tian//author//" instead of "tian// authorship//".

The other form of heterogeneity that is common in practice is *value heterogeneity*: sources refer to the same real-world objects in different ways (e.g., references to persons, addresses, publications). We leave the handling of value heterogeneity to future work, as it raises some novel challenges.

4.5 Neighborhood Keyword Queries

The indexing methods we described so far lend themselves almost immediately to answering neighborhood keyword queries. We build the *Keyword Inverted List (KIL)*, which is essentially a Hybrid-AAIL. In a KIL we summarize not only prefixes that end with hierarchy paths, but also prefixes that correspond directly to keywords. To answer a neighborhood keyword query with keywords $K_1, \ldots, K_n$, we transform it into a prefix search for $K_1//*, \ldots, K_n//*$.

EXAMPLE 6. *Table 7 shows a fragment of the KIL with threshold* $t = 1$*. Given the neighborhood keyword query "Birch", we look up "birch//*" and return instances* a_1, c_1, p_1 *and* p_2*.* □

Note that if we wish to distinguish between the relevant instances (those for which the keywords occur in attribute values) and the associated instances (those for which the keywords occur in associated instances), we can add two special symbols as the root of all attributes and the root of all associations, and index accordingly.

5. EXPERIMENTAL EVALUATION

We now describe a set of experiments that validate the efficiency of our indexing methods and compare them against several alternatives. Our main result shows that by indexing both structure information and text values, we can considerably improve the performance of answering predicate queries and neighborhood keyword queries. In addition, we examine the efficiency of updating the index and the scalability of our index.

5.1 Experimental Setup

The main data set we use is constructed from a collection of personal data on the desktop and a few external

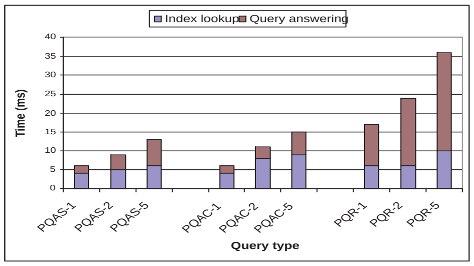


Figure 4: Efficiency of answering predicate queries. In each column, the longer bar shows the overall query-answering time and the shorter bar shows index-lookup time.

sources. We extract associations between disparate items on the desktop (e.g., LATEX and BIBTEX files, Word documents, Powerpoint presentations, emails and contacts, and webpages in the web cache). The instances and associations are stored in an RDF file, managed by the Jena System [28]. The RDF file contains 105,320 object instances, 300,354 attribute values, 468,402 association instances, and the size of the file is 52.4MB. We describe additional data sets we used in our scale-up experiment in Section 5.4.

We considered four types of queries:

- PQAS: Predicate queries with only attribute clauses where the attributes do *not* have sub-attributes;
- PQAC: Predicate queries with only attribute clauses where the attributes *do* have sub-attributes;
- PQR: Predicate queries with only association clauses;
- NKQ: Neighborhood keyword queries (we did not distinguish between relevant and associated instances).

We varied the number of clauses in the first three types of queries from one to five, and each clause had a single keyword. For NKQs, we varied the number of keywords from one to five. The keywords, attributes, and associations were randomly drawn from the data set.

For each query configuration, we randomly generated 100 queries, and executed each three times. We report the average execution time. To further refine our measurements we also consider the index-lookup time, including the time to locate the entries in the inverted list, the time to retrieve the occurrence counts for each returned instance, and also the time to handle succeeding rows in case of prefix lookup.

We implemented the indexing module using the Lucene indexing tool [34], which stores an inverted list as a sorted array. We implemented our algorithm in Java, and conducted all the experiments on a machine with four 3.2GHz and 1024KB-cache CPUs, and 1GB memory.

5.2 Indexing and Searching

We tested the efficiency of the KIL, the hybrid hierarchical index (see Section 4.5). It took 11.6 minutes to build the KIL and its size is 15.2MB. The query-answering time of predicate queries is shown in Figure 4 and that of both predicate queries and neighborhood keyword queries is shown in Table 8.

We make three observations about the results. First, answering predicate queries and neighborhood keyword queries using the KIL was very efficient: on average it took 15.2 milliseconds to answer a predicate query with no more than 5 clauses, and took 224.3 milliseconds to answer a neighborhood keyword query with no more than 5 keywords. Second, answering PQASs and PQACs (where attribute hierarchies were considered) consumed a similar amount of time, showing the effectiveness of our hybrid indexing scheme. Third, though answering PQRs (queries with associations) took longer time than answering PQASs and PQACs, they spent similar amount of time in index lookup. The difference was in the time to retrieve the answers, and there were much more of them for the PQRs than for the other two types of queries. For the same reason, it took much longer time to answer NKQs.

5.2.1 *Comparison of methods*

Next, we compare our index with several alternative approaches. We first compare the efficiency of KIL with two other methods: NAIVE and SEPIL. The NAIVE method is based on the basic inverted list (alluded to in Section 2). Specifically, NAIVE begins by looking up the set of instances $\mathcal{I}$ that contain the given keywords in attribute values, and then does the following:

- PQAS: Select from $\mathcal{I}$ the instances where the keywords appear in the specified attributes;
- PQAC: The same as PQAS, but also consider descendant attributes;
- PQR: Find the instances that are related to the ones in $\mathcal{I}$ with the specified associations;
- NKQ: Union $\mathcal{I}$ with all instances that are associated with those in $\mathcal{I}$.

The SEPIL method is an adaptation of the approach proposed in [32] to our context (originally it was designed for complex XML queries). Specifically, it builds three separate indexes: the *inverted list* indexes each attribute value on its text, the *structured index* indexes each attribute value on the labels of the attribute and its ancestor attributes, and the *relationship index* indexes each instance on its associated instances. SEPIL begins by looking up the inverted list for a set of attribute values $\mathcal{A}$ that contain the query keywords, and meanwhile getting their owner instance set $\mathcal{I}$. Then SEPIL does the following:

- PQAS and PQAC: Look up the *structured index* for values of the specified attributes and intersect the results with $\mathcal{A}$, then return the owner instances;
- PQR: Look up the *relationship index* for the instances that are related to the ones in $\mathcal{I}$ with the specified associations;
- NKQ: Look up the *relationship index* for the instances associated with the ones in $\mathcal{I}$, and union the results with $\mathcal{I}$.

Note that unlike our approach, NAIVE and SEPIL return the instances without counting keyword occurrences or the number of associated instances. Performing the count would add a significant overhead to both of these techniques.

Table 8 shows the query-answering time using these three different indexes. It took 1.7 minutes to build the NAIVE index, whose size was 10.6MB. Query answering was inefficient using the NAIVE index, because we had to find the involved attributes and extract associated instances at run time. It was especially inefficient when we had to extract associated instances for a large number of instances that

Table 8: Comparison of search efficiency using the KIL, using separate indexes as proposed in [32], and using a simple inverted list. The KIL improved search efficiency significantly.

(ms)		1 clause		2 clauses		5 clauses	
		Lookup	Query-answer	Lookup	Query-answer	Lookup	Query-answer
PQAS (Predicate queries with simple attributes)	Naive	2	22	3	53	4	129
	SepIL	7	9	8	11	10	15
	KIL	4	6	5	7	6	13
PQAC (Predicate queries with complex attributes)	Naive	3	43	3	119	4	583
	SepIL	7	11	23	28	31	38
	KIL	4	6	8	11	9	15
PQR (Predicate queries with associations)	Naive	3	88	7	147	12	368
	SepIL	301	415	559	749	1397	1871
	KIL	6	17	6	24	10	36
NKQ (Neighborhood keyword queries)	Naive	18	4174	28	5244	50	8407
	SepIL	365	488	717	1052	1662	2376
	KIL	48	97	103	182	232	394

Table 9: Comparison of indexing efficiency for different types of inverted lists. Building a Hybrid-ATIL introduces little overhead compared with building an ATIL.

Index type	Indexing time for shallow-hierarchy triple base (s)	Indexing time for deep-hierarchy triple base (s)
ATIL	118	118
Dup-ATIL	125	418
Hier-ATIL	119	140
Hybrid-ATIL	125	144

contain the given keywords. Indeed, compared with KIL, query-answering time on average increased by a factor of 15.9 and for 1-clause NKQs increased by a factor of 43. We also observed that although KIL spent longer time in index lookup (because the index was 1.4 times as large and more instances were returned in each index lookup), the overall payoff in query-answering time significantly outweighed this additional cost.

It took 5.7 minutes to build the SepIL index. The total size of the inverted list and the structured index was 28.1MB, and the size of the relationship index was 14.2MB. For PQAS and PQAC queries, query-answering time was reduced on average by a factor of 6.6 compared with Naive; however, because the inverted list is large, it still took about twice time as much as KIL. For other queries that require looking up the relationship index, query answering took much longer time than KIL (by a factor of 20.7). This is because the index is large, and for each instance that contains the keyword we need to look up the index and accumulate its associated instances. Even when compared with Naive, there was only a benefit to building the relationship index for answering NKQ queries, which typically returned a large number of instances.

We performed several other experiments to validate different aspects of our indexing methods. For example, we considered only attributes and compared the efficiency of the ATIL with a technique that creates a separate index for each attribute. We observed that ATIL reduced indexing time by 63% and reduced keyword-lookup time by 33%.

5.2.2 Indexing hierarchies

We now compare different methods for indexing attribute hierarchies that were described in Section 4:

- ATIL: use the ATIL but expand a query by issuing a query for every descendant attribute (without accumulating keyword occurrences for result instances)
- Dup-ATIL: duplicate keywords for ancestors
- Hier-ATIL: attach the ancestor path
- Hybrid-ATIL: the hybrid index

In the data set we experimented on, the depth of each attribute hierarchy is no more than 3. To examine the effect of hierarchy depth on search efficiency, we also experimented on a triple base where depths of attribute hierarchies are all over 16. We call the former a *shallow-hierarchy* triple base and the latter a *deep-hierarchy* triple base. The two triple bases have exactly the same data but different schemas: if an attribute does not have any parent attribute in the shallow-hierarchy triple base, in the deep-hierarchy triple base it has a parent attribute $attr_0$, a grand-parent attribute $attr_1$, and so on, till the upmost ancestor $attr_{15}$.

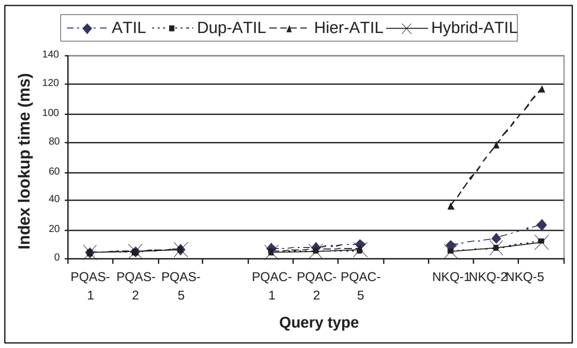
Table 9 shows the index-building time for these inverted lists on both triple bases. Note that a higher hierarchy depth had only significant effect for the construction of the Dup-ATIL (increasing indexing time for a factor of 3.34 and the size of the index for a factor of 4); building a Hier-ATIL or a Hybrid-ATIL took similar amount of time as building an ATIL on both triple bases and the sizes of the indexes are similar. Also note that building a Hybrid-ATIL took only slightly longer time than building a Hier-ATIL, which shows that our algorithm for adding summary rows in Hybrid-ATILs is efficient.

Figure 5 shows the index-lookup time (we omit the query-answering time as it adds the same amount of time over the index-lookup time for all alternative methods). We observe that (1) Hier-ATIL performed poorly on NKQs, as prefix lookup became extremely expensive; (2) Dup-ATIL performed poorly on the deep-hierarchy triple base, as the index size was increased a lot, and (3) Hybrid-ATIL performed better than or equal to any other inverted lists for all types of queries on both data sets.

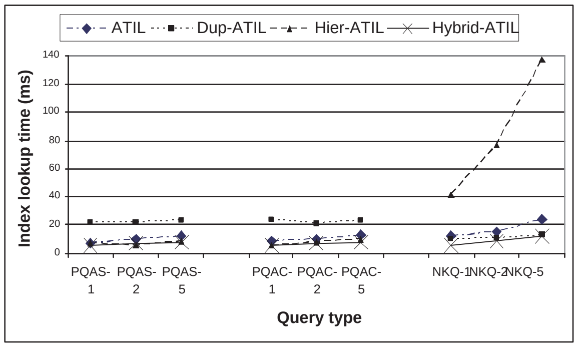
5.3 Index Updates

Our next experiment was designed to measure the efficiency of updating the KIL, both for instance updates and for updates to the schema.

For instance updates, we randomly selected 100 instances, divided them into groups, and interleaved insertion and dele-



(a)



(b)

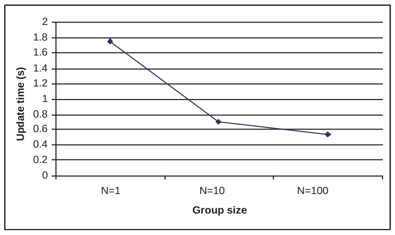
Figure 5: Efficiency of looking up different types of indexes in answering predicate queries with attribute clauses (a) on shallow-hierarchy triple base, and (b) on deep-hierarchy triple base. On both triple bases looking up the Hybrid-ATIL took the shortest time.

tion in each group. We updated a group of instances incrementally; that is, we inserted or deleted the instances in the group, and updated their associated instances in the index. We varied the size of the group: 1, 10 and 100, and compared the average time of updating an instance in KIL.

Figure 6(a) shows the time for inserting or deleting an instance in KIL. The results for insertion-only or deletion-only updates were similar. We observed that when the group size was increased, the update time per instance dramatically dropped. For example, when $N = 100$, updating an instance took on average only 0.5 seconds. In addition, when the size of the group was increased, the speedup of the updates slowed down.

We also observed that index updates in the SepIL method were slower by a factor of 2.25 compared to updates in KIL, but updates in Naive were considerably faster than in both methods. This is because most of the update cost arose from the need to update associated instances in the index. However, as previous experiments have shown, indexing associations significantly sped up query answering at run-time and thus was worthwhile.

For structure updates, we considered four types of operations: renaming an attribute, inserting, updating, and deleting a parent for an attribute. For each operation, we chose three attributes that occur with different frequencies in the data set, and reported the average time for updates. We performed each operation by scanning the inverted list and changing the indexed keywords appropriately. Structure updates were performed very efficiently. As shown in Figure 6(b), it took 2.2 seconds on average to perform each type of structure update.



(a)

Operation	Update time (s)
Rename an attribute	2.2
Insert a parent	2.4
Rename a parent	2.5
Delete a parent	2.0

(b)

Figure 6: Efficiency of KIL updates: (a) instance updates can be performed efficiently in medium-sized groups in an incremental mode; (b) structure updates can be performed efficiently by scanning the index and changing the index keywords.

Table 10: Index-lookup time for answering (a) the original queries and (b) suffix queries on 250MB data sets with perturbed keywords. Index lookup was efficient: on average it took 30.3 milliseconds for predicate queries and 281.6 milliseconds for neighborhood keyword queries. For the purpose of comparison, we also list the index-lookup time on the 25MB data.

(ms)	25MB	f=0	f=0.2	f=0.4	f=0.6	f=0.8
PQAS	5	25	24	23	22	24
PQAC	8	26	27	27	26	26
PQR	6	32	30	35	37	49
NKQ	103	805	628	490	318	139

(a)

(ms)	f=0	f=0.2	f=0.4	f=0.6	f=0.8
PQAS	22	27	28	26	26
PQAC	24	27	28	28	27
PQR	29	43	48	49	46
NKQ	27	46	86	131	146

(b)

5.4 Scalability

Finally, we tested the scalability of KIL with larger data sets. In the first experiment, we created a 250MB data set by adding to the original data set four copies of itself and then perturbing the result data set. Specifically, we chose a perturbation factor $f \in [0, 1]$. When we perturbed the keywords with factor f, we randomly selected a fraction f of the keywords in the data set, and for those words we added one of the suffixes in $\{!, @, \#, \%\}$ with equal probability. (These signs do not occur in the original index.) Hence, when $f = 0$, the keywords are the same as those in the original set, and when $f = 0.8$, for any keyword k the number of its occurrences is about the same as that for $k!$ (or k with any other suffix). We perturbed attribute and association names in the same way.

We experimented on two sets of queries: the original queries and *suffix* queries, where "!" was added to each keyword. We considered randomly generated queries with two clauses.

Table 11: Indexing time and index-lookup time for 10GB data sets.

(ms)	Wikipedia	XMark w/o asso	XMark with asso
Index	4.15hr	6.64hr	12.72hr
PQAS	156	94	116
PQAC	-	67	93
PQR	-	-	217
NKQ	1646	1838	13468

We now describe our experimental results on data sets with perturbed keywords. We observed the same trend for data sets with perturbed attribute and association names. As f was increased, the indexing time went up gradually from 55.3 minutes to 58.2 minutes, and the size of the index went up gradually from 71.2MB to 76.4MB, all roughly 5 times as much as for the original data set. Table 10 shows the index-lookup time. We make three observations.

First, when the number of answers was small, index-lookup time was more related to the size of the index. For all predicate queries, index-lookup time was roughly 5 times as much as that for the original data set. We note that with Lucene, the index look-up time increases linearly with the index size.

Second, when the number of answers was large, index-lookup time was more related to the number of the answers. Although the sizes of the indexes for all different data sets were similar, the index-lookup time for ordinary NKQ queries dropped significantly when f was increased (so the number of answers was decreased), and showed the opposite trend for suffix queries. In particular, when $f = 0.8$, the number of answers for ordinary NKQ queries and that for suffix NKQ queries were similar, and also similar to that for NKQ queries on the original data set. We indeed observed similar index-lookup time for these three cases. This observation implies that our index scales especially well when the number of returned answers is large.

Third, for suffix queries on the non-perturbed data set, the answers are empty, and index lookup was still efficient (on average took 25 milliseconds)

In the second set of experiments, we considered two XML data sets, each of size 10GB. The first data set is from the INEX Wikipedia collection [17] (with duplicates). It contained 1.4 million instances, each with only two attributes. The second data set was generated by XMark [38]. It contained 11.4 million instances, 76.2 million attribute values, and 58.2 million association instances. We indexed the XMark data set in two ways: one indexed only attribute values and one indexed associations in addition. We used these indexes to answer randomly generated queries with two clauses.

The three indexes varied in size: the Wikipedia index was 1.13GB, the XMark index without associations was 3.04GB, and the XMark index with associations was 4.08GB. As shown in Table 11, our indexing technique scales well: on average it took 123.8 milliseconds to look up the index for predicate queries, only 4.1 times as much as for the 250MB data. We also observed that although indexing associations took about twice as much time as indexing only attributes, the increase in keyword-lookup time was not significant (except for neighborhood keyword queries, where considering associations considerably increased the number of returned instances); however, it significantly sped up query answering in presence of association clauses.

6. RELATED WORK

The two bodies of work most close to ours are indexing XML and on keyword queries in relational databases.

There have been many indexing algorithms proposed for answering XML queries. They can be categorized into three classes: indexing on structure, indexing on value, and indexing on both. The first class (e.g., [21, 35, 15, 33, 11, 31, 25]) considers supporting schema-driven queries, such as "list all book authors", and does not index text values. The second class (e.g., [2, 9, 13, 29, 41, 36, 44]) is mainly oriented to XML Twig queries [9]. It indexes text values and at the same time encodes parent-child and ancestor-descendant relationships by numbering the XML elements appropriately. The encoding methods are tuned for tree models, and would not apply in our context where associations between instances form a graph.

The third class combines indexes on structure and on text. We have already compared our approach to that of Cooper et al. [16] (in Section 4). Kaushik et al. [32] and Chen et al. [12] proposed building multiple indexes to capture different aspects of structural information and values of XML data. If we adapt their indexing methods for our context, to answer a query we need to visit several indexes in sequence, looking up an index for each result returned from the previous index. As we show in Section 5.2.1, this process can be quite time-consuming. ViST (Virtual Suffix Tree), proposed in [40], encodes both XML documents and XML queries as suffix sequences and answers the queries by suffix matching. This strategy is again more suitable for tree models and falls short in our context.

Our approach is different from the ones described above in that it does not rely on any specific data model, and it uses one *single* index to capture both structure information and text values. In this way, our method is more oriented to keyword search, can more easily explore associations between data items, and can more efficiently answer keyword queries with simple structure specifications.

Several works have considered keyword queries on relational databases. The DISCOVER [26], DBXplorer [1], and BANKS [7] systems return minimal join-networks that contain all the keywords given in a query. These approaches require building the join-network at run-time and so query answering can be expensive. Queries in form of "SEARCH {instance-type} NEAR {keywords}" are proposed in [20, 10], where the distances between elements are precomputed and indexed, so the index can be quite large and hence costly. Similar ideas were considered in the context of XML data, returning the least common ancestors (LCA) for the elements that contain the given keywords [43, 27].

Finally, SphereSearch [22] and Kite [37] studied search across heterogeneous data by first conducting data transformation or integration. In contrast, we take a "data-coexistence" approach and index heterogeneous data even if they are only loosely coupled.

7. CONCLUSIONS AND FUTURE WORK

We described a novel indexing method that is designed to support flexible querying over dataspaces. The querying mechanism allows users to specify structure when they can, but also to fall back on keywords otherwise. Answers to keyword queries also include objects associated with the ones that contain the keywords. Our methods extend inverted lists to capture structure when it is present, including attributes of instances, relationships between instances, synonyms on schema elements, and hierarchies of schema elements. We validated our techniques with a set of experiments and showed that incorporating structure into inverted lists can considerably speed up query answering.

In future work, we plan to extend our index to support value heterogeneity and to investigate appropriate ranking algorithms for our context. In particular, since we often have confidence numbers on matching on schema elements or reference reconciliation on data items, we would like to take these confidences into account in the ranking algorithm.

Acknowledgments

We would like to thank Magda Balazinska, Mike Cafarella, Shawn Jeffery, Dan Suciu, Anthony Tung and the anonymous reviewers for many helpful discussions and comments. The work of the first author was funded by NSF Grant #IIS-0415175.

8. REFERENCES

[1] S. Agrawal, S. Chaudhuri, and G. Das. DBXplorer: A system for keyword-based search over relational databases. In *Proc. of ICDE*, 2002.

[2] S. Al-Khalifa, H. Jagadish, N. Koudas, J. M. Patel, D. Srivastava, and Y. Wu. Structural joins: A primitive for efficient XML query pattern matching. In *ICDE*, 2002.

[3] R. Baeza-Yates and G. Gonnet. Fast text searching for regular expressions or automaton simulation over tires. *Journal of the ACM*, 43(6):915–936, 1996.

[4] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. ACM Press, New York, 1999.

[5] M. Banko, M. J. Cafarella, S. Soderland, M. Broadhead, and O. Etzioni. Open information extraction from the web. In *IJCAI*, 2007.

[6] H. Bast and I. Weber. Type less, find more: Fast autocompletion search with a succinct index. In *SigIR*, 2006.

[7] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using BANKS. In *Proc. of ICDE*, 2002.

[8] L. Blunschi, J.-P. Dittrich, O. R. Girard, S. K. Karakashian, and M. A. V. Salles. A dataspace odyssey: The iMeMex personal dataspace management system. In *CIDR*, 2007.

[9] N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: Optimal XML pattern matching. In *Sigmod*, 2002.

[10] S. Chakrabarti, K. Puniyani, and S. Das. Optimizing scoring functions and indexes for proximity search in type-annotated corpora. In *WWW*, 2006.

[11] Q. Chen, A. Lim, and K. W. Ong. D(k)-index: An adaptive structural summary for graph-structured data. In *Proc. of SIGMOD*, 2003.

[12] Z. Chen, J. Gehrke, F. Korn, N. Koudas, J. Shanmugasundaram, and D. Srivastava. Index structures for matching xml twigs using relational query processors. In *ICDE Workshops*, 2005.

[13] S.-Y. Chien, Z. Vagena, D. Zhang, V. J. Tsotras, and C. Zaniolo. Efficient structural joins on indexed XML documents. In *Proc. of VLDB*, 2002.

[14] J. Cho and S. Rajagopalan. A fast regular expression indexing engine. In *Proc. of ICDE*, 2001.

[15] C. Chung, J. Min, and K. Shim. APEX: An adaptive path index for XML data. In *Proc. of SIGMOD*, 2002.

[16] B. F. Cooper, N. Sample, M. J.Franklin, G. R.Hjaltason, and M. Shadmon. A fast index for semistructured data. In *Proc. of VLDB*, 2001.

[17] L. Denoyer and P. Gallinari. The Wikipedia XML Corpus. *SIGIR Forum*, 2006.

[18] P. DeRose, W. Shen, F. Chen, Y. Lee, D. Burdick, A. Doan, and R. Ramakrishnan. DBLife: A community information management platform for the database research community. In *CIDR*, 2007.

[19] X. Dong and A. Halevy. A Platform for Personal Information Management and Integration. In *CIDR*, 2005.

[20] R. Goldman, N. Shivakumar, S. Venkatasubramanian, and H. Garcia-Molina. Proximity search in databases. In *Proc. of VLDB*, 1998.

[21] R. Goldman and J. Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In *Proc. of VLDB*, Athens, Greece, 1997.

[22] J. Graupmann, R. Schenkel, and G. Weikum. The SphereSearch engine for unified ranked retrieval of heterogeneous XML and web documents. In *VLDB*, 2005.

[23] M. Gubanov and P. A. Berstein. Structural text search and comparison using automatically extracted schema. In *WebDB*, 2006.

[24] A. Y. Halevy, M. J. Franklin, and D. Maier. Principles of dataspace systems. In *PODS*, 2006.

[25] H. He and J. Yang. Multiresolution indexing of XML for frequent queries. In *Proc. of ICDE*, 2004.

[26] V. Hristidis and Y. Papakonstantinou. DISCOVER: Keyword search in relational databases. In *VLDB*, 2002.

[27] V. Hristidis, Y. Papakonstantinou, and A. Balmin. Keyword proximity search on XML graphs. In *Proc. of ICDE*, 2003.

[28] Jena. http://jena.sourceforge.net/, 2005.

[29] H. Jiang, H. Lu, W. Wang, and B. C. Ooi. XR-Tree: Indexing XML data for efficient structural joins. In *ICDE*, 2003.

[30] Y.-J. Joung and L.-W. Yang. KISS: A simple prefix search scheme in P2P networks. In *WebDB*, 2006.

[31] R. Kaushik, P. Bohannon, J. F. Naughton, and H. F. Korth. Covering indexes for branching path queries. In *Proc. of SIGMOD*, 2002.

[32] R. Kaushik, R. Krishnamurthy, J. F. Naughton, and R. Ramakrishnan. On the integration of structure indexes and inverted lists. In *Proc. of SIGMOD*, 2004.

[33] R. Kaushik, P. Shenoy, P. Bohannon, and E. Gudes. Exploiting local similarity for indexing paths in graph-structured data. In *Proc. of ICDE*, 2002.

[34] Lucene. http://jakarta.apache.org/lucene/docs/index.html, 2005.

[35] T. Milo and D. Suciu. Index structures for path expressions. In *Proc. of ICDT*, 1999.

[36] P. Rao and B. Moon. PRIX: Indexing and querying XML using Prufer sequences. In *ICDE*, 2004.

[37] M. Sayyadian, H. Lekhac, A. Doan, and L. Gravano. Efficient keyword search across heterogeneous relational databases. In *ICDE*, 2007.

[38] A. Schmidt, F. Waas, M. Kersten, M. J. Carey, I. Manolescu, and R. Busse. XMark: A benchmark for XML data management. In *VLDB*, 2002.

[39] P. Valduriez. Join indices. *ACM transactions on Database Systems*, 12(2), 1987.

[40] H. Wang, S. Park, W. Fan, and P. S. Yu. ViST: A dynamic index method for querying XML data by tree structures. In *Proc. of SIGMOD*, 2003.

[41] W. Wang, H. Jiang, H. Lu, and J. X. Yu. PBiTree coding and efficient processing of containment joins. In *ICDE*, 2003.

[42] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and indexing documents and images*. Morgan Kaufmann Publishers, San Francisco, 1999.

[43] Y. Xu and Y. Papakonstantinou. Efficient keyword search for smallest LCAs in XML databases. In *SIGMOD*, 2005.

[44] N. Zhang, T. Ozsu, I. F. Ilyas, and A. Aboulnaga. Fix: Feature-based indexing technique for XML documents. In *VLDB*, 2006.