A Relational Approach to Incrementally Extracting and Querying Structure in Unstructured Data

Eric Chu Akanksha Baid Ting Chen AnHai Doan Jeffrey Naughton
Computer Sciences Department
University of Wisconsin-Madison
{ericc, baid, tchen, anhai, naughton} @cs.wisc.edu

Abstract

There is a growing consensus that it is desirable to query over the structure implicit in unstructured documents, and that ideally this capability should be provided incrementally. However, there is no consensus about what kind of system should be used to support this kind of incremental capability. We explore using a relational system as the basis for a workbench for extracting and querying structure from unstructured data. As a proof of concept, we applied our relational approach to support structured queries over Wikipedia. We show that the data set is always available for some form of querying, and that as it is processed, users can pose a richer set of structured queries. We also provide examples of how we can incrementally evolve our understanding of the data in the context of the relational workbench.

1. Introduction

Currently, to find information from the vast amount of unstructured data (i.e., text) on the Web, users have to rely on a combination of keyword search, browsing, and possibly predefined search options. Although these mechanisms are easy to use and often lead to what users are looking for eventually, they cannot leverage the potentially rich set of structures embedded in text. For example, consider the page about the city Madison in Wikipedia [36]. It contains sections of text titled "History," "Geography," "Demographics," and so on. From the text we might find relationships that we want to extract (e.g., "technology companies" such as "Raven Software" and "Human Head Studios" have headquarters located in "Madison"). Furthermore, the page has "wiki tables" that capture explicit structured data, such as one that records the city's monthly average low, average high, and mean temperature, and mean precipitation (a portion of the table is shown in Figure 1). The ability to query this set of structures is highly desirable. For example, we may want to know how cold Madison gets during winter, by averaging the minimum temperature of December, January, and February; and if there is also a temperature table for the city Seattle, we may want do the same for Seattle and compute the difference between the two averages. However, to do so, we need to extract these tables and pose structured queries.

	Jan	**Feb**	**...**	**Dec**
Avg High Temp °F (° C)	23 (-5)	29 (-2)	...	29 (-2)
Avg Low Temp °F (° C)	6 (-14)	12 (-11)	...	13 (-11)
Mean Temp °F (° C)	15 (-9)	20 (-7)	...	21 (-6)
Avg precipitation in (cm)	1.14 (2.9)	1.14 (2.9)	...	1.32 (3.35)

Figure 1. A portion of the temperature table from the page about Madison in Wikipedia. Though the content has a clear structure, users cannot query it using structured queries.

Since unstructured data can practically contain any structure, extracting and maintaining a set of structures for querying are a constant work in progress for a system. Therefore, at any point in time, a user should be able to query using as much or little structure as is currently known, and should be able to perform increasingly sophisticated queries as the system incrementally evolves its understanding of the data.

Unfortunately, there is little consensus about what kind of system should be used to support this kind of incremental capability. In this paper, we explore using a relational system as the basis for a workbench for extracting and querying structure from unstructured data. This approach is perhaps surprising because relational database systems are often regarded as one of the most

VLDB '07, September 23-28, 2007, Vienna, Austria.
Copyright 2007 VLDB Endowment, ACM 978-1-59593-649-3/07/09.

rigidly structured alternatives, and their design goals seem diametrically opposed to the kind of flexibility required for incremental discovery and exploitation of structure. However, we show that this is not the case with our workbench model, which provides 1) a way to store the evolving set of documents and structures, 2) tools that can be used to query and to incrementally process the data, and 3) a way to handle changes in our understanding of the data set as it is processed.

The system we envision allows users to load a set of documents without any pre-processing, and begin querying the documents immediately using keyword searches. Of course, at this point, there is no benefit above that provided by a traditional Information Retrieval (IR) system. To extract more information from the collection of documents, we need to somehow process the documents, store the result of that processing, and make it available for querying. For example, it may be useful to run one or more clustering tools over the documents, and to record the results of these by labelling the documents with cluster identifiers. It may be useful to extract structure from the documents, perhaps in the form of attribute-value pairs, and to record the attribute-value pairs found in each document. It may also be useful to integrate these attribute-value pairs; that is, record that attribute *i* of one document corresponds to attribute *j* of another document.

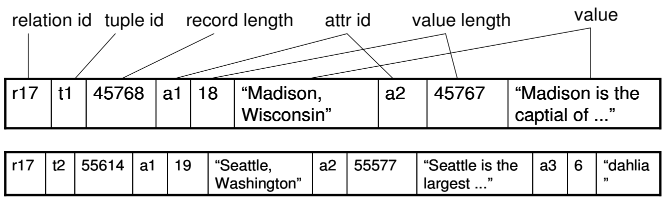
It is important that all this information is available for querying, in any combination, at any time. For example, users should be able to run keyword searches over documents belonging to any combination of clusters and restrict the search to documents that satisfy predicates on specified attribute values. Users should also be able to run SQL-like queries over the extracted attributes, perhaps again limited to documents belonging to specified clusters. Also, as more attributes are correlated by integration, the results of queries over these attributes (possibly extracted with different names in different documents) should improve in quality.

Certainly one could build such a system using many different approaches. One could start from scratch and write a stand-alone system that works over file-system resident data. One could adopt an XML-centric approach, and place one's hope in the growing capabilities of XML query engines. These and other approaches could certainly be successful. However, it is our argument that existing relational database technology can go a long way toward supporting such a system. Our basic idea is simple: at the start, the documents are loaded into the RDBMS in a table with only two attributes: id and text. As clustering tools and extracting tools are run over the data, we add attributes, and store the results of those tools in the new attributes. Of course, most attributes will be null for most documents; however, recent work on managing sparse data sets provides evidence that such extremely sparse data sets can be efficiently managed by an RDBMS. As integration tools relate sets of attributes, we record these relationships (and the lack thereof) in "mapping tables."

The workbench we envision supports three basic operators – Extract, Integrate, and Cluster – for

name	id	type	size
DocTitle	a1	VARCHAR(100)	100
DocContent	a2	TEXT	unlimited
official flower	a3	VARCHAR(50)	50
headquarter.city	a4	VARCHAR(50)	50
headquarter.company	a5	VARCHAR(50)	50

Figure 2. Attribute catalog.



r17	t1	45768	a1	18	"Madison, Wisconsin"	a2	45767	"Madison is the captial of ..."

r17	t2	55614	a1	19	"Seattle, Washington"	a2	55577	"Seattle is the largest ..."	a3	6	"dahlia "

Figure 3. Records in interpreted storage format.

processing the data set incrementally. It uses a modified "wide table" to store the data set, and a mapping table and a relationship table to store the schematic relationships within the set of structures.

A workbench based on a relational system can offer many benefits for supporting structured queries over unstructured documents. First, the data is always available for querying. With full-text indexes, users can start posing keyword queries over the set of documents as soon as it is loaded into the workbench. As we obtain more structure over time, the data's utility increases and users can pose increasingly sophisticated queries. Second, the operators can be combined and applied repeatedly to keep evolving our understanding of the data set. To handle the evolving set of structures, the wide table provides a simple, but scalable alternative that does not require complicated schema design. Lastly, in addition to support for querying, the workbench can take advantage of other strengths of a database, such as concurrency control, recovery, and query optimization.

The reason we call our approach a workbench is that it does not do anything by itself – it only provides the tools to process, manage, and query the data. Database administrators (DBAs), and perhaps even the users, need to decide the parameters of the operators, the choice of specific clustering, extraction, and integration algorithms, what to do with the output, when they are finished with processing, and other relevant issues. Our goal is to provide an environment in which to run these tools, record the results, and make the results available for querying.

As a proof of concept, we applied our approach to support structured queries over Wikipedia. Wikipedia is a database of unstructured documents that contain a lot of structured data. However, currently, the only way to query documents in Wikipedia is by doing "vanilla" keyword search (i.e., with no advanced search options), and browsing. In the case study, we first describe our simulation of the workbench. Next, we provide examples of how we combine the three basic operators to incrementally evolve our understanding of the data, and show that users can benefit from each stage of this process. For example, with just a little effort in extraction, we can allow users to specify a scope for their keyword queries to improve precision; with more effort, we can allow them to build more complex structured queries that include arithmetic comparison, aggregation, and even joins.

The rest of our paper is organized as follows. Section 2 presents the data and schema representation adopted in the workbench. Section 3 defines the three operators and describes how we use them to evolve our understanding of the data in the context of the workbench. Section 4 presents our case study on Wikipedia. Section 5 discusses related work. Section 6 concludes the paper and suggests future work.

2. Schema and Data Representations

2.1 A Wide, Sparse Table with Complex Attributes

In considering a storage model for the documents and their extracted structures, an important observation is that the continuing extraction of heterogeneous structures will gradually lead to a sparse data set. A data set is considered sparse when it comprises a large number of attributes, but most entities (or documents in our case) have non-null values for only a small fraction of all these attributes.

For storing a sparse data set, Agrawal et al. [4] discussed using vertical tables as an alternative to horizontal tables with positional storage. However, Beckmann et al. [7] later showed that vertical tables generally suffer from complex queries and poor performance, and that horizontal tables with interpreted storage outperform both vertical tables and positionally stored horizontal tables. More recently, we argued that using a multi-table schema to store a sparse data set often creates more problems than it solves, and that the right approach is to use a wide table [14]. That is, forego schema design and store all objects in a single horizontal table using the interpreted storage format. We use this same storage model for our workbench.

Unlike the predominant positional storage format, which would cause a huge storage space blow-up by storing the null values in a sparse data set, the interpreted storage format [13] avoids storing the null values. Specifically, the system uses an attribute catalog to record for each attribute its name, id, type, and size. A tuple in the interpreted format starts with a header, which contains fields such as relation-id, tuple-id, and record length; then, for each of its non-null attributes, the tuple stores the attribute's identifier, length field (if the type is of variable length), and value. Attributes that appear in the catalog, but not in the tuple, are implicitly null for that tuple. The interpreted storage format is highly flexible for schema evolution – we only need to update the system catalog and the tuples that have non-null values for these attributes.

Let us consider how an empty wide table in our workbench evolves when we insert the two pages about Madison and Seattle from Wikipedia. Each page corresponds to a row in the table. We declare two attributes in the catalog. The first one is the title of the page (DocTitle), which we use as a unique identifier for a page for the purpose of demonstrating. The second is the content of the page (DocContent). Clearly, these two columns in the table will be dense because every page must have non-null values for them.

DocTitle	**DocContent**	**official flower**	**headquarter(city, company)**
Madison, Wisconsin	"Madison is the capital of the U.S. state of Wisconsin ..."		[(Madison, Raven Software), (Madison, Human Head Studios), ...]
Seattle, Washington	"Seattle is the largest city in the Pacific Northwest ..."	dahlia	[(Seattle, Starbucks), (Seattle, Amazon.com), ...]

Figure 4. The wide table after extracting the attribute "official flower" and then the relationship "headquarter(city, company)."

The schema grows when we apply an extractor and find at least one page containing the target attribute. For example, suppose we run an extractor on the two documents and it extracts the value "dahlia" for the attribute "official flower" (we allow attribute names to be keywords or phrases, just as the elements in a "malleable" schema [19]) from the Seattle page. To reflect this knowledge, we add "official flower" to the catalog (Figure 2), and update the record for Seattle by appending information about the new attribute to the end of the record (the second record in Figure 3). Because the extractor does not find a value for this attribute from the Madison page, we leave the corresponding record unaffected. In Figure 4, the first three columns represent the current state of the wide table. Similarly, if for some reason we decide we need to remove the attribute "official flower," we will just need to remove the entry in the attribute catalog and update only the records that have a non-null value for "official flower."

Unfortunately, the conventional practice of first normal form – the requirement that each field in the database holds an atomic value – would not work well for our workbench for two reasons. First, it is common for an extractor to extract multiple instances of the same structure. For example, an extractor for the attribute "lake" will extract instances such as "Lake Mendota" and "Lake Monona" from the Madison page, and "Lake Washington" and "Lake Union" from the Seattle page. Storing each instance in a separate column in the wide table is unreasonable, especially when there are many instances (e.g., Seattle has 22 instances of "sister city"). However, if we create a new table for each distinct structure, the number of tables will be prohibitively large, and a SQL query will likely involve many joins. Second, even if the document contains only one instance of a particular structure, the structure can be complex, such as a hierarchy or an n-ary relationship. A concrete example is the weather table in Figure 1. Each temperature value is

host id	host name	mappings
a6	temp (°F)	{a6 = a7 * 9/5 + 32}
a7	temperature (°C)	{a7 = 5/9 * (a6 – 32)}

Figure 5. The mapping table to reconcile the two attributes that have the same meaning (albeit in different units).

relationship id	definition
r1	{a4, a5}

Figure 6. A structure table with only one entry for the headquarter(city, company) relation.

associated with three attributes: the month, the semantics (minimum or maximum), and the unit (Fahrenheit or Celsius). Although in this case the DBAs could transform the table to make it relational and store each attribute in the wide table, this solution is inconvenient in the long run. Ideally, having extracted a structure, we would like to just store it in a column with as little administration as possible.

To fix these problems, we allow a table in the workbench to contain complex attributes (e.g., attributes whose values can be lists, arrays, tables, sets of tuples, etc.), in keeping with the complex attribute support found in object-relational database systems. Note that we do not require that the RDBMS-provided query language know how to operate on all of these structures – instead, we envision users and/or administrators writing user defined functions (UDFs) that "know about" these structures and are invoked in users' queries. Figure 4 shows the wide table after we extract instances of the relation headquarter(city, company) and store them under one column.

In summary, the wide table provides a simple, but flexible way to store the evolving set of structures extracted from the documents. Each document corresponds to a row in the wide table. New structure discovered from a document is appended at the end of the corresponding row. Due to the diversity of structure, the table can have many attributes, be very sparse, and some attributes can have internal structure.

Some might doubt the scalability of this wide-table approach. One problem is that the number of attributes that this wide table can contain is limited by the number of bits allocated for the attribute identifier – we currently use a 16-bit attribute identifier, which limits the number of attributes to 65,536. As we will see in our case study with Wikipedia, depending on the extraction approach, the number of attributes can easily exceed this number even with just one data source. For this problem, we note that nothing prevents us from allocating, for instance, 32 bits for the attr-id. But more importantly, the main purpose of the wide table is to provide convenient and flexible storage to cope with the evolution of data. At the beginning, we know nothing about the structure, so everything is stored in one table. However, as we gain a better understanding of the data, we may identify subsets of structures that are logically different. At this point, we could optionally create views or new tables for these subsets, or "split" the wide table. Deciding whether and when to go beyond the "single table" view of the data is an interesting topic for future work.

2.2 Mapping Table and Relationship Table

The mapping table is a data structure to store mappings for different attributes that correspond to the same real-world concept. It is similar in spirit to the mapping tables described by Kementsietsidis et al. [27]. In the logical view, each row in the mapping table describes a set of mappings to a distinct "host" attribute. A mapping to the host is an expression that may include just the identifier of another attribute for a simple 1-1 correspondence, or an expression if the mapping involves some form of conversion or involves more than one attributes (i.e., n-1 correspondence to the host). For example, if the weather table from the Madison page records the temperature only in Fahrenheit with the attribute "temp (°F)," and the weather table from the Vienna page records its temperature only in Celsius with the attribute "temperature (°C)," then we will want to map the two attributes to support any query that involves these two measures. Figure 5 shows the mapping table after the update. The column "host name" is just for demonstration.

When a query includes an attribute, the system looks up the attribute in the mapping table. If there is a mapping between this attribute and some other attribute, the query may need to rewrite the query to also include the matching attribute for evaluation.

The purpose of the relationship table is to record complex structures that comprise multiple attributes, such as the headquarter(city, company) relation, for possible future updates. In the logical view of this table, each row describes a distinct structure with two attributes: a relationship identifier and the set of attributes that belong to the structure. Figure 6 shows the entry that defines the headquarter relation, whose attributes "city" and "company" are also added to the attribute catalog shown in Figure 2. Note that this is not the only way to keep track of which attributes belong to which complex structure. For example, instead of using a relationship table, we could add a column that stores the relationship id of each attribute, if applicable, in the attribute catalog.

3. Operators for Incremental Processing of Data

Two types of evolution can occur in the workbench. The first one is due to the system's evolving understanding of the data. The second is due to changes to the documents such as inserting and deleting documents, and updating the contents of existing documents. In the latter case, deleting and updating document contents may cause changes that need to be propagated to the set of structures, the mapping table, and the relationship table. We do not consider this case in this paper. Instead, we assume that once a document is in the workbench, its content remains unchanged. For the purpose of explanation, in this section, we assume that the DBAs are the only ones using these operators, although that is not necessarily the case as we could potentially allow user participation in improving the structure. However, addressing how to leverage mass collaboration in the context of the workbench is out of the scope of this paper.

We identify three basic operators – Extract, Integrate, and Cluster – that the workbench should support. Just as their names suggest, Extract is for extracting structure, Integrate is for identifying attributes that correspond to the same real-world concept, and Cluster is for clustering a set of different attributes based on some similarity function. These operators are the basic building blocks that can be combined and applied repeatedly to keep evolving the system's understanding of the data.

The operators should satisfy three requirements. First, each operator should be able to use different algorithms. Second, the DBAs should be able to specify a scope for the input on which a chosen method operates. Third, given the output, the DBAs should be able to specify what they want to do with it. We can consider each of these operators as a procedure in which a DBA specifies the scope of an operator through a query, and the other parameters – methods, output, and action with the output – in a UDF. In Section 4.1, we discuss some possible parameters for each operator, then in Section 4.2, we describe how the operators can be combined to improve performance.

3.1 Basic Operators

Extract:

We classify extraction methods into two types. The first one detects structure such as entity and relationship from natural language. Most IE systems fall into this category, such as DIPRE [8], Snowball [2], and KnowItAll [21]. The second type extracts structured data embedded in text of known format, such as LaTex, XML, and wiki markup text [6]. It is a common practice to write ad hoc scripts to extract structured data from a specific format. Both types of extractors should be considered for use in the workbench.

The output of an extractor is a set of structures. We assume that the schema of a structure is part of the extractor's definition, and that each structure may have one or more instances. The DBAs can either store the output in the wide table, or feed it to the Integrate and Cluster operators. Before storing a structure, the system should check the attribute catalog and the relationship table to see if the structure has been used before. If there is an exact match between an existing structure and the newly discovered structure, we store the instances in the same column in the wide table. Otherwise, we have to catalog the new structure and store its instances in a new column (but in the same row as the corresponding document).

As we will show later, the scope parameter is useful for improving the performance of the extractors as the system processes the data. The DBAs can also apply an extractor on columns other than DocContent, to extract structure of a finer granularity in existing structure (e.g., from "date" to "day," "month," and "year"). Finally, although the scope is usually specified as a SQL query, in the case of Extract, the scope can also be expressed as a keyword query to select a set of documents relevant to a specific topic. This approach can be an efficient way to filter irrelevant documents for domain-specific extractors, as demonstrated by Agichtein et al. [1].

Integrate:

Integrate takes as input a set of structures from the wide table or a previous operator, and returns one or more sets of mappings over attributes that correspond to the same real-world concept. Based on schema-matching techniques, the chosen method can consider schema-based information (e.g., attribute names and clusters of attributes), and instance-based information (e.g., data contents). The DBAs have to decide what to do with each set of mappings. They can store the mappings in the mapping table. Alternatively, if the attributes have not yet been inserted into the wide table (e.g., they have just been generated by Extract), the DBAs may consider collapsing the attributes into one attribute in some cases, such as when the different attributes are just stems of the same word.

Cluster:

Cluster takes in a set of documents or a set of attributes and classifies the input into one or more clusters. Although it sounds like Integrate, Integrate identifies attributes that are semantically the same, whereas Cluster tries to group together different documents or structures based on some predefined notion of similarity. Document clustering is a well-studied area in Information Retrieval and a variety of approaches can be used for this operator. For attribute clustering, one method that we explored in previous work is to group together attributes that have non-null values in the same tuples [14], which was shown to be very promising for sparse data sets.

The clustering information is useful in a number of ways. For instance, it helps the DBAs decide what views to build to optimize SQL queries over the wide table. As mentioned earlier, when there are clear clusters and the table approaches the maximum number of attributes it can store, the DBAs may want to physically split the wide table into multiple tables for the clusters. Also, the clusters may reveal undiscovered domain knowledge that may improve the other two operators.

Our workbench does not currently model the results of an operator with their probability of being accurate, as in probabilistic databases. Instead, it relies on the threshold that the particular method uses to determine extraction, integration, and clustering; the DBAs can also verify the output of the operators themselves. Although incorporating probabilities and reasoning about them could be very useful, it is orthogonal to the basic operators of our workbench and out of the scope of this paper.

One assumption we make in this model is that the specific algorithms of the three operators are already coded as programs that can be conveniently applied to the data in the workbench. Unfortunately, the reality is much more complicated. For example, currently, PostgreSQL, which we use for our case study, only allows UDFs written in C/C++, so even if we have extraction scripts written in Perl, we cannot apply them as UDFs. How to facilitate the application of external programs within the workbench is an important and practical problem to address. A possible idea is to create a repository of external programs. The DBAs and users can write these external programs (that may follow some guidelines set by the workbench), and upload them to the repository. The workbench can then use them on the data set.

3.2 Operator Interaction

One powerful feature of our workbench is that the operators can be combined synergistically in a "whole is greater than the sum of the parts" fashion. That is, the operators can be combined to improve each other's performance, in terms of both efficiency and quality. There are six possible pairwise combinations of distinct operators: Integrate-Extract, Cluster-Extract, Extract-Cluster, Integrate-Cluster, Extract-Integrate, and Cluster-Integrate. Of course, they can be extended into a sequence of operators – we could do triples, quadruples, etc. In the following, we explain for each case how the previous operator benefits the subsequent one.

Integrate-Extract:

Integrate can help find new targets for Extract. For example, suppose that the DBAs have used Extract to extract from the attribute "address" the finer-grained attributes "street address", "city," "state," and "zip code." When Integrate identifies a mapping between "address" and another attribute "sent-to" based on the data instances, the DBAs may want to apply the same extractor on "sent-to."

Cluster-Extract:

At the early stage of processing, when the DBAs know nothing about the documents, they can only apply domain-independent extractors on the entire set of documents. However, when Cluster discovers a specific domain, the DBAs can narrow the scope of Extract and apply domain-specific extractors on only the documents in this domain. Because domain-specific extractors are more powerful but often applicable to only a small subset of the documents, domain discovery can greatly improve Extract's efficiency and the quality of its results.

Extract-Cluster:

The extracted set of structures may provide more information that Cluster can use to group together documents or attributes. For example, we try to cluster pages about cities in Wikipedia (see Section 6) based on the section names they contain. Although we are fairly successful in finding most city pages, the short pages are left out because they do not have a section name (just a title). However, after we extract the "city info-box" structure in some of these pages, Cluster will recognize them and put them in the city cluster.

Integrate-Cluster:

Integrate can prevent Cluster from creating multiple clusters where logically a single cluster would be better. For example, given a data set with attributes {C#, Company, FirstName, LastName, CustID, Contact, CName}, Cluster may find two clusters – the tuples either have non-null values for the set of attributes {C#, CName, FirstName, LastName}, or the set of attributes {CustID, Company, Contact}. However, if we have run Integrate first and find the set of mapping {C# = CustID, CName = Company, FirstName + LastName = Contact}, then Cluster will end up with only one cluster.

Extract-Integrate:

This pair is more a necessity than an option – obviously we need to extract a set of structures before integrating them.

Cluster-Integrate:

Cluster can narrow the scope for Integrate in two ways. First, when Cluster identifies a domain for a set of structures, the DBAs may want to apply domain-specific schema matchers on this set of structures. Second, when Cluster identifies two overlapping sets of attributes (e.g., {CustID, CName} and {CustID, Company}), DBAs may want to look for possible mappings in the difference between the two sets of attributes because they may be semantically the same (e.g., CName = Company).

We conclude this section with two thoughts. First, incremental processing is a flexible scheme to support structured queries over unstructured data. On the one hand, the interaction of operators with different methods supports robust evolution of structure. On the other hand, DBAs can also process the data lazily – that is, they do not try to process the data unless they determine that getting the structure will significantly improve searching experience. A lazy approach is more appropriate when resources are limited, as it avoids over-processing structure that users do not care. Second, the correctness of structured queries over this data set is limited by how much, and how well, the DBAs have processed the data. In other words, the results obtained via structured queries can have less-than-perfect recall and precision.

4. Case Study: Wikipedia

4.1 Preliminaries

As a proof of concept, we conducted a preliminary case study on applying our workbench model on Wikipedia, an online encyclopedia written collaboratively by volunteers. Coincidentally, in addressing DB and IR integration, Weikum recently suggested turning Wikipedia into a database that can answer advanced queries, as a first, smaller-scale step to turn the entire Web into a gigantic knowledge base [35]. The pages in Wikipedia are actually stored in a relational database; however, they are stored as blobs of text and the only way to find information from them is via browsing and "vanilla" keyword search, with no advanced search options such as those found in the "Advanced Search" page of Google [24]. The purpose of this case study is to illustrate how our relational workbench can incrementally evolve structure from the contents of Wikipedia, and how users can pose increasingly sophisticated queries at each stage of the processing.

Wikipedia has many qualities that make it an ideal subject of this case study. The contents are embedded in wiki markup text. There are guidelines on how to create and edit a wiki page. As a result, even though many users can make changes to the same page, in general the pages have a consistent structural organization. For example, they all have a title; many of them comprise text organized into a hierarchy of sections, and contain structured data in the forms of "wiki table" and "info box." Moreover, it encourages the use of templates for wiki tables and info boxes in the same domain, so the same structure is often used across many pages.

For our case study, we downloaded a database dump of Wikipedia that includes only the current revisions, as of December 5, 2005. The dump has more than 4 million XML files in 8.5 GB. Each XML file contains a blob that is the content of a wiki page, and metadata about the page such as page-id, title, revision-id, contributor-user-name, last-modification-date, etc. To more easily track the evolution progress and the results of our queries, we also selected a small subset of pages as a control data set. The control set comprises pages from three domains: major American cities (254 files), major universities from the states of Wisconsin, New York, and California (255 files), and top male tennis players on the ATP tour in the "Open Era" (373 files). For the rest of this section, we refer to these domains as "City," "University," and "TennisPlayer," respectively. We ran our experiments in PostgreSQL.

4.2 Incremental Processing

Stage 1: Initial Loading

In the first stage, we parsed the XML files and loaded them into a single table, which initially had five columns: PageId, PageText, RevisionId, ContributorUserName, and LastModificationDate. Each page corresponds to a single row. We used the page title as the PageId of a page. PageText contains the content of the page in wiki text. The other attributes describe the metadata about the page. With a full-text index on PageText, users can already query the documents using keyword searches, even though we have not begun processing the data.

Stage 2: Extracting SectionName(text)

Next, from each page we extracted the structure SectionName(text), in which SectionName represents the name of a first-level section in the page and the text is the content in that section. For example, the page titled "Madison, Wisconsin" has 16 first-level sections, such as "History," "Geographics," "Demographics," and so on. For each instance of this structure we appended it to the end of the corresponding row. Note that when two instances had different section names, we stored their text in two different columns. (Figure 7)

PageId	PageText	History	Economy	Campus	Personal Life
Madison, Wisconsin	"Madison is the captial of the ..."	"Madison was created..."	"Wisconsin state government. ."		
Seattle, Washington	"Seattle is the largest city in ..."	"What is now Seattle..."	"Five companies on the ..."		
Stanford University	"The Leland Stanford ..."	"Stanford was founded..."		"Stanford University owns ..."	
Roger Federer	"Roger Federer (born August 8, 1981) is a ... "				"Federer was born..."

Figure 7. A portion of the wide table after extracting SectionName(text) over the control data set. The table has a total of 1,258 attributes and 882 rows, but only 1.05% of all cells have non-null values.

One reason to extract this structure is that it allows us to do "focused" keyword search. For example, suppose we want to retrieve from the control set the pages about male tennis players who have been ranked world number one, we can pose the keyword query "World No. 1 tennis player" over the PageText column. For this query, PostgreSQL returns 86 players. It includes all the 23 players who indeed have been ranked number one in the world since the "Open Era." For the other 63 players, most of them are included for reasons such as they have been ranked number one in doubles, they have defeated a top-ranked player, and so on.

Continuing with the example, suppose we know that the pages of the players who have been number one almost always mention that fact in the introduction section. Therefore, we may want to try posing the same keyword query over only the introduction section. This query returns 67 players, 21 of which have been number one. In other words, it gives us better precision but worse recall. Incidentally, for the two players who are excluded, their introduction does mention that they have been ranked number one before, but the fact is expressed as "world number one" instead of "world No. 1."

Although extracting the structure SectionName(text) allows us to do focused keyword search, it leads to inserting 1,253 new attributes into the wide table. Each row has only 13 non-null attributes on average. The entire table has about 1.05% non-null values. Fortunately, using the interpreted storage format can avoid the storage explosion caused by storing the null values.

We checked how many of these attributes are equivalent based on name similarity, and found that even in this small control data set with just 882 files from 3 domains, a significant percentage of attributes are equivalent or highly similar to another attribute. Specifically, more than 350 of the 1,253 attributes belong to one of the 14 most common attribute topics. Figure 8 shows these 14 attribute topics with some examples. We

City	University	TennisPlayer
famous people (37): famous people from abilene, famous people born in akron, famous madisonians, ...	**athletics (7):** athletics and traditions, school athletics, athletics and mascots, athletics highlights, ...	**single_titles (28 total):** singles titles (21), singles titles (33), singles titles (5), ...
demographics (15): demographics and diversity, population and demographics, population history, ...	**greek life (14):** greek social organizations at alfred, greek letter organizations, greek social fraternities, campus greek life, fraternities, ...	**doubles_titles (12 total):** doubles titles (15), mens doubles titles (50), career doubles titles (54), ...
museum (23): museums and cultural organizations, museums and attractions, museum and historical attractions, museums and cultural arts, ...	**campus (73):** facilities & campus construction, hillside campus, main campus, new york city campus, campus and facilities, ...	**personal (6):** personal and family life, personal life, personal information, ...
colleges and universities (33): schools & colleges, colleges & research institutes, schools & universities, ...		**grand_slam_record (16):** grand slam records, grand slam results, grand slam history, ...
culture and entertainment (33): arts and entertainment, entertainers, arts literature humanities, ...		**career overview (35):** career highlights, tennis career, professional career, ...
highways (24): us highways, highways, streets and highways, ...		

Figure 8. The 14 attribute topics with the most aliases found in the control data set. For each topic, we show the most representative attribute in bold, the number of aliases in that topic, and some examples.

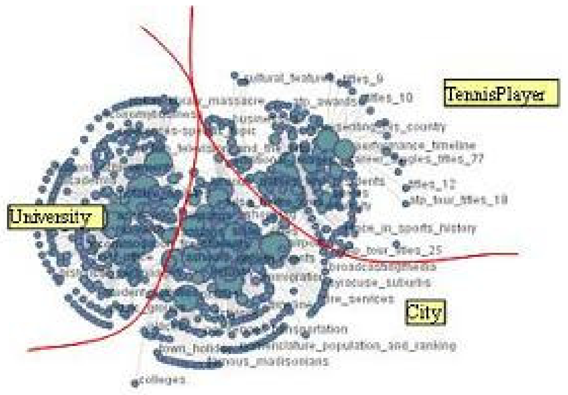


Figure 9. A network diagram by Many Eyes on the section names extracted from the control data set. We can see that the section names form three fairly clear clusters.

could store these mappings in a mapping table as described in Section 3.

Figure 8 reveals two interesting observations. First, many section names are a mix of attribute names and values. For example, the attribute "singles titles" is frequently followed by a number in parentheses, e.g., "singles titles (13)." Apparently, the many aliases of "singles titles" are due to the convention of putting the total number of titles next to the section name, while the content of that section describes when and where a player has won each of these titles. We also observed this pattern of mixing data with metadata in attributes such as "doubles titles" and "famous people."

This observation indicates an opportunity for extraction. For the "singles titles" example, we can extract the number in parentheses into a new structure, NumberOfSinglesTitlesWon(number), and store it in the wide table. The section names can be changed back to "singles titles," resulting in many fewer aliases. This example illustrates how Integrate and Extract can benefit each other.

The second observation is that given the output of an operator, there is often more than one reasonable action. For example, after we extracted the SectionName(text) structure, we immediately stored them into the wide table, and then stored the mappings in the mapping table. However, a better decision might be to run Integrate and Extract as described above to reduce the attribute explosion, and store only the sections that we believe are likely to be used for focused keyword search. Of course, the first approach has the advantage that we can "undo" any mappings that are later found to be incorrect, simply by updating the mapping table. We leave these decisions to the DBAs.

Suppose we do not know that the documents in the control set come from the domains University, City, and TennisPlayer. We might be motivated to see if the SectionName(text) instances form any clusters because pages in the same domain often have similar section names. We used the following definition of Jaccard coefficient to identify clusters. Given two attributes (section names in this case) A_X and A_Y, let X be the set of rows for which A_X is non-null, and let Y be defined analogously. The Jaccard coefficient for A_X and A_Y is defined as:

$$\text{Jaccard}(A_X, A_Y) = |X \cap Y| / |X \cup Y|$$

The coefficient's value ranges from zero to one. It is zero when no rows have non-null values for both A_X and A_Y, and one when A_X and A_Y are either both null or both non-null for all tuples. We created an adjacency matrix on the attributes with the Jaccard coefficients as the values. Next, we tried to visualize the clusters via a network diagram, a feature in the Many Eyes visualization tool developed by IBM [32]. The tool takes in an input list of distinct attribute pairs that have a Jaccard coefficient greater than

Domain (# files)	# Returned Pages	# Correct Pages	Recall	Precision
City (254)	477	247	.97	.51
University (255)	498	240	.94	.48
TennisPlayer (373)	375	330	.88	.88

Figure 10. The results of posing a keyword query to retrieve a set of documents for each domain. TennisPlayer enjoys a high precision because its domain is relatively disjoint from the other two domains, in which many pages mention both cities and universities.

The University of Wisconsin-Madison	
Motto	Numen Lumen *The divine within the ...*
Established	1848
Type	Public State University
Faculty	2053
Students	41466
...	...
Colors	Cardinal & White
Mascot	Bucky Badger

Figure 11. Info box of the University of Wisconsin-Madison.

0.1, and appear non-null in at least 0.05% of rows. The output is a graph in which the vertices correspond to the attributes, and the edges represent the pairs that satisfy the constraints. Strongly related attributes are kept in close proximity to each other. The size of a vertex is proportional to the number of its outgoing edges. Figure 9 shows this network diagram, which depicts the three clusters quite clearly. Looking at the vertices closely, we were able to identify the domains of the three clusters, as labelled in the figure.

We also tried doing keyword search as a means to approximate clusters. That is, assuming we know that the domains are City, University, and TennisPlayer, we pose three keyword queries: "city," "university," and "tennis player" over the control set to get three sets of documents. Figure 10 shows the results of these keyword queries. All of them have high recall, which is not surprising because the keyword queries describe the domains accurately. However, notice that while TennisPlayer enjoys a relatively high precision, the other two domains have significantly poorer precision. This result also makes sense because many university pages mention the city where the university is located, and many city pages mention major universities that are located in the city. We posed the same keyword queries over the entire Wikipedia, and retrieved 175,193 pages for City, 114,173 pages for University, and 851 pages for TennisPlayer. These numbers are reasonable because they include pages about cities and universities from all over the world, and all kinds of tennis players, including male and female professionals, and possibly amateur and junior players. Nevertheless, these numbers are a huge reduction from the 4 million+ pages in Wikipedia, so keyword search is a very good way of narrowing the scope of documents for further processing.

We can draw an interesting comparison between the two approaches. Clustering section names does not depend on any prior knowledge about the set of documents. Moreover, if there is a set of extracted structures, then clustering is a good approach for discovering domains because based on the clusters, we could create views to improve efficiency for queries over this set of structures. In contrast, the keyword search approach is simple and potentially effective if the DBAs have prior knowledge about the documents. This approach is appropriate for identifying a scope for applying domain-specific extractors. This example demonstrates the robustness of our relational workbench, as it supports a wide variety of approaches for doing the same task.

As yet another approach to find clusters, we note that for many online unstructured data sets, the contents have already been organized into subsets. For example, the contents in Wikipedia are organized into categories and presented as many lists (which can in turn contain more lists). We can certainly leverage this existing organization.

Identifying clusters of attributes can help increase the efficiency of queries over the data set. For example, after we cluster the documents based on these domains over the table that contains all Wikipedia documents, the scanning times for City, University, and TennisPlayer are 26.12 ms, 24.62 ms, and 25.69 ms, respectively. Without this clustering information, to find the documents from each cluster, we have to scan the entire table, which takes about 44 seconds.

In concluding this stage, we note that extracting only one kind of structure – SectionName(text) – already leads to many opportunities for evolving the structure. Therefore, it is very important to have a flexible infrastructure to handle this evolution.

Stage 3: Extracting info box as a blob

In the next stage of processing, we extracted info boxes, which are a general template that contains predefined attributes and vary depending on the domain. That is, the definition of a city info box is different from that of a university info box. Figure 11 shows a portion of the info box of the University of Wisconsin-Madison.

Although ideally we would want to extract each attribute-value pair, a much simpler alternative is to just store the entire info box as a blob. This blob would not support structured queries over the attributes; however, it allows focused keyword search over the info box. For instance, we can find out which universities have "cardinal" as their school color, by posing the keyword query "cardinal" over the university info boxes in our control set. The answer set for this query includes seven schools. Six of them have "cardinal" as one of their school colors. One of them has "red" as its school color, but has a mascot called "Cardinal Burghy." In contrast, running the keyword query "cardinal university" over the PageText column of the control data set returns 51 pages.

Wide Table

PageId	PageText	History	Economy	temperature
Madison, Wisconsin	“Madison is the captial of the ...”	“Madison was created...”	“Wisconsin state government..”	temperature_wiki

temperature_wiki

City	Month	Low_F	Low_C	High_F	...
Madison, Wisconsin	1	6	-14	23	...
Madison, Wisconsin	2	12	-11	29	...
Madison, Wisconsin	12	13	-11	29	...
Seattle, Washington	1	36	2	46	...

Figure 12. Implementation for storing an internal table in the wide table – a table identifier is assigned to the internal table (temperature_wiki), which is created as a separate physical table.

Most of them are either university pages that mention sports teams whose names contain “cardinal,” or city pages that contain the term “cardinal” in various contexts, such as sports teams, high schools, religion, and “cardinal-direction.” This example shows that sometimes even a small effort in processing data can greatly improve the quality of keyword search.

Stage 4: Extracting structured data from info boxes and wiki tables

In the last stage of our case study, we demonstrate the process of extracting and querying structured data from info boxes and wiki tables. For our first example, we go back to address the first query given at the beginning of the paper: compute the average of the average low temperatures of December, January, and February from the temperature wiki table of Madison (Figure 1). To do this query with our relational workbench, first we need to transform the contents of the temperature wiki table into a relation. There are many alternatives. We chose to use the following schema:

temperature_wiki(city, month, lowF, lowC, highF, ...)

Since PostgreSQL does not allow a column to store a relation, we simulated the effect by storing a table identifier in the column that originally stores temperature_wiki according to our model, and created temperature_wiki as a separate table. We used the name “temperature_wiki” as the table identifier. Figure 12 shows the wide table and a portion of temperature_wiki. The following query computes the average of the average low temperatures of January, February, and December:

Q1:
```
SELECT AVG(Low_F)
FROM     temperature_wiki as T
WHERE   T.city = ‘Madison, Wisconsin’ AND
Month = 1 OR Month = 2 OR Month = 12;
```

Since temperature_wiki associates each measurement with a city, we could reuse it for any city page that contains this wiki table. Once we have extracted many temperature wiki tables from pages of Wikipedia into temperature_wiki in this fashion, we could pose queries that do powerful comparisons, such as “find the coldest city,” or “rank the cities based on their precipitation.”

As a comparison to the earlier keyword queries that retrieve top ranked male tennis players, we extracted the tennis info box structure, which has the attribute highest_singles_rankings, and posed the equivalent structured query:

Q2:
```
SELECT   id
FROM     info_box_tennis_player
WHERE   highest_singles_rankings = 1;
```

Q2 returns 23 players, 22 of which have been ranked number one. This result has a much better precision than the keyword queries. One player that has been ranked number one is not included in the answer because that player’s page does not have an info box. This example demonstrates the improvement we can potentially get by exploiting structured data embedded in text; however, the correctness of the query is limited to the availability of the structure and the quality of our processing.

In addition to arithmetic comparison and aggregation, a powerful advantage that structured queries have over keyword search is their ability to join documents. For example, suppose a student wants to find out which university is located in a place that can get very cold in January. The student can pose a query that joins the university pages in the wide table with the temperature table of the cities where the universities are located, based on the “location” field extracted from the university pages:

Q3:
```
SELECT T1.ID
FROM  WideTable T1, temperature_wiki T2
WHERE     T1.location = T2.city AND T2.month =
1 AND Low_F < 20;
```

Not surprisingly, Q3 returns the University of Wisconsin-Madison. In our data set, this university is the only one returned because only a small number of cities have temperature tables, and many of them are in sunny California.

In this case study, we present the first few stages of incrementally evolving structure from a small set of Wikipedia pages. Although we have only scratched the surface of processing these pages, we have already seen a significant improvement in the type of queries that users can use: from keyword search over unstructured data, to focused keyword search over sections of text and blobs of info boxes and wiki tables, to doing arithmetic comparison, aggregation, and even joins. Note that during this process, we have only focused on extracting structure based on the syntax of the Wiki markup text; we have not even used extractors based on natural language processing and statistical learning. There is still so much structure yet to be discovered and exploited, that we need a flexible infrastructure that gives us many options in how to manage these evolving structures.

5 Related Work

There is a large body of literature relevant to various aspects of our workbench model. In this section, we try to cover a representative sample of this related work, but it is by no means exhaustive.

Our wide table resembles Google's Bigtable [12], a distributed storage system for managing structured data that is designed to scale to a very large size. The difference is that Bigtable focuses on storing document metadata, whereas our wide table needs to store different forms of extracted structures.

Our Extract, Integrate, and Cluster operators overlap somewhat in their functions with Data Cleaning tools [34], which deal with detecting and removing errors and inconsistencies from data in order to improve the quality of data. Also, supervised learning algorithms can be defined as operators that DBAs can apply to the data set. Exploring how to adapt existing data cleaning and learning tools in the context of our workbench is a very interesting and promising area for future work.

There is a large body of literature in information extraction [e.g., 3, 15, 17], data integration [e.g, 25], and data clustering [e.g., 5], which we will not describe here. In the following, we review some recent or ongoing projects that address problems similar to those addressed by the workbench.

AVATAR [28], a prototype by IBM, aims to provide seamless support for queries over unstructured and structured data, mainly from the business domain. It relies on hand-written annotators to emit the structures. Although a relational database (DB2) is used to store these structures, AVATAR devises an object model as an abstraction layer to hide the details of the underlying storage. It also features a statistical model to handle uncertainty about the extracted structures. One of its focuses is to support online analytic processing (OLAP) over uncertain and imprecise data [9].

ExDB [10] is an "extraction database" that extracts structures from web text and supports structured queries over them. Using IE systems that are domain-independent and unsupervised, such as KnowItAll [21], it extracts data values (e.g., "Einstein," "Switzerland"), binary relationships (e.g., "Einstein" was born in "Switzerland"), and semantic types (e.g., "Switzerland" is a "country"). The tuples are loaded into a probabilistic database, which records for each tuple the probability of that tuple being true. ExDB supports structured probabilistic queries that use a Datalog-like notation.

SEMEX [18] is a platform for personal information management and integration. It supports desktop search via semantically meaningful associations, which may be extracted by analyzing specific file formats (e.g., Latex and Bibtex), derived from external sources, or defined by users. Therefore, a major challenge is to identify different references that correspond to the same real-world concept [20]. Another line of work is the proposal of "malleable" object-oriented schemas to model uncertainty that arises in diverse and evolving structures [19].

Cimple [16] is a platform for community information management. For instance, its prototype system, DBLife, manages information for the database research community. Although it is domain (or rather, community) specific, the data may come from multiple sources. With a larger group of users, Cimple explores techniques that leverage mass collaboration, for tasks such as improving the accuracy of data integration tools [33].

Google's PAYGO [30] is a data integration architecture intended to manage structured data on the Web scale. Therefore, it has to model any kind of structures, which can come from any domain (e.g., school, government, sports, etc.) and from any source (e.g., queryable HTML forms in the Deep Web [11], Flickr [22], Google Base [23], etc.). PAYGO and our workbench model share the same philosophy that a system should be able to incrementally evolve its understanding of the data.

For storing sparse data sets, Yu et al. [37] and we [14] separately advocated the use of a wide table – that is, forego complicated schema design and store all attributes in a single physical table. As explained in Section 2, our workbench uses a modified wide table to store the evolving set of structures.

Mansuri et al. [31] presented a system for automatically integrating unstructured text into a multi-relational database. By using statistical models for structure extraction and matching, the system loads unstructured records into columns that spread across multiple tables in the database, and resolves the relationship of the extracted text with existing column values.

Liu et al. [29] described an alternative approach to answer structured queries over unstructured data. Instead of extracting structures from unstructured data, it transforms a given structured query to a keyword query and poses this keyword query over the unstructured data directly. Although this approach does not require extraction, it somewhat defeats the purpose of posing structured queries, as it is inapplicable to express queries that involve disjunction, inequality predicates (e.g., <, >), or aggregation. However, we noted that when a data set is sparse, it is often possible to use keyword search as an optimization technique for many structured queries that contain only equality predicates in conjunction [14]. Jain et al. [26] describes another approach to answer structured queries over text data, by executing multiple extractors and combining their results on the fly.

Although many of these systems explore similar problems and propose similar techniques in comparison to our work, none of them employs their approach inside a relational database in an end-to-end fashion. For most of them, the use of a relational database is limited to storing a set of structures, which usually have a well-defined schema when they are loaded into the database. Some systems, such as SEMEX, do not even use a relational database (although it would be possible to use one). In contrast, we focus on the seemingly unpromising idea of incrementally processing data in a relational database.

6 Conclusion

In this paper, we propose the use of a relational database as a workbench not only for storing and querying structured data, but also for incrementally evolving structure from unstructured data. As a proof of concept, we conducted a case study of applying our approach to evolve and query the structure in the contents of Wikipedia. Our experience in this study demonstrated that our approach exploits existing technology effectively and allows one to quickly and incrementally discover and query the structure lurking in unstructured documents.

Much scope for future work remains – virtually every aspect of our system can be "drilled down" upon to discover and evaluate alternative approaches. Some interesting outstanding problems include:

- How to handle updates to the unstructured data. That is, how should these updates be propagated to the wide table, the mapping table, etc.? This topic is relevant especially for Wikipedia, in which users update the content all the time.
- How to record the evolution of data, so when a new document arrives and we find that it is similar to existing documents in the workbench, we know how we should process the new document.
- How to help users write queries that exploit the structure discovered in this workbench.
- How to optimize queries in this context, such as when they involve attributes that have many mappings.

We intend to address these and other questions in the future, and it is our hope that our initial work in this area will inspire other researchers to also address these questions.

References:

[1] E. Agichtein, L. Gravano. Querying Text Databases for Efficient Information Extraction. ICDE 2003: 113-124.

[2] E. Agichtein, L. Gravano. Snowball: Extracting relations from large plain-text collections. ACM DL, 2000.

[3] E. Agichtein and S. Sarawagi. Scalable Information Extraction and Integration. KDD 2006 Tutorial.

[4] R. Agrawal, A. Somani, Y. Xu. Storage and querying of e-commerce data. VLDB, 2001.

[5] P. Andritsos, P. Tsaparas, R. Miller, K. Sevcik. LIMBO: Scalable Clustering of Categorical Data. EDBT 2004.

[6] S. Auer, J. Lehmann: What have Innsbruck and Leipzig in common? Extracting Semantics from Wiki Content. ESWC 2007.

[7] J. Beckmann, A. Halverson, R. Krishnamurthy, J. Naughton. Extending RDBMSs to support sparse datasets using an inerpreted attributes torage format. ICDE, 2006.

[8] S. Brin. Extracting patterns and relations from the World-Wide Web. WebDB, 1998.

[9] D. Burdick, P. Deshpande, T. S. Jayram, R. Ramakrishnan, S. Vaithyanathan: OLAP Over Uncertain and Imprecise Data. VLDB 2005.

[10] M. Cafarella, C. Re, D. Suciu, O. Etzioni, M. Banko. Structured Querying of Web Text. CIDR 2007.

[11] K. Chang, J. Cho. Accessing the Web: From Search to Integration. SIGMOD 2006 Tutorial.

[12] Chang et al. Bigtable: A Distributed Storage System for Structured Data. OSDI 2006.

[13] N. Chapin. A Comparison of File Organization Techniques. In Proc. of 24th national conference, pg. 273-283, USA, 1969. ACM Press.

[14] E. Chu, J. Beckmann, J. Naughton. The Case for a Wide-Table Approach to Manage Sparse Relational Data Sets. SIGMOD, 2007.

[15] W. Cohen, A. McCallum. Information Extraction from the World Wide Web. KDD 2003 Tutorial.

[16] P. DeRose, W. Shen, F. Chen, A. Doan, R. Ramakrishnan. Building Structured Web Data Portals: A top-down, compositional, and incremental approach. VLDB 2007.

[17] A. Doan, R. Ramakrishnan, S. Vaithyanatha. Managing Information Extraction. SIGMOD 2006 Tutorial.

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

[19] X. Dong, A. Halevy. Malleable Schemas: A Preliminary Report. WebDB 2005.

[20] X. Dong, A. Halevy, J. Madhavan. Reference Reconciliation in Complex Information Spaces. SIGMOD 2005.

[21] O. Etzioni, M. Cafarella, D. Downey, S. Kok, A. Popescu, T. Shaked, S. Soderland, D. Weld, A. Yates. Unsupervised named-entity extraction from the web: An experimental study. Artificial Intelligence, 165(1):91-134, 2005.

[22] Flickr. http://www.flickr.com.

[23] Google Base. http://base.google.com.

[24] Google Advanced Search. http://www.google.com/advanced_search?hl=en

[25] A. Halevy, A. Rajaraman, J. Ordille. Data Integration: The Teenage Years. VLDB 2006.

[26] A. Jain, A. Doan, L. Gravano. SQL Queries over Unstructured Text Databases. ICDE 2007.

[27] A. Kementsietsidis, M. Arenas, R. Miller. Mapping Data in Peer-to-Peer Systems: Semantics and Algorithmic Issues. SIGMOD 2003.

[28] R. Krishnamurthy, S. Raghavan, J. Thathachar, S. Vaithyanathan, and H. Zhu. AVATAR information extraction system. IEEE Data Engineering Bulletin, Special Issue on Probabilistic Databases, 29(1), 2006.

[29] J. Liu, X. Dong, A. Halevy. Answering Sturctured Queries on Unstructured Data. WebDB 2006.

[30] J. Madhavan, S. Jeffery, S. Cohen, L. Dong, D. Ko, C. Yu, A. Halevy. Web-scale Data Integration: You can only afford to Pay As You Go. CIDR 2007.

[31] I. Mansuri, S. Sarawagi. A system for integrating unstructured data into relational databases. ICDE 2006.

[32] Many Eyes. http://services.alphaworks.ibm.com/manyeyes/home

[33] R. McCann, A. Kramnik, W. Shen, V. Varadarajan, O. Sobulo, A. Doan. Integrating Data form Disparate Sources: A Mass Collaboration Approach. ICDE 2005.

[34] E. Rahm, H. Do. Data Cleaning: Problems and Current Approaches. IEEE Bulletin of the Technical Committee on Data Engineering, Vol 23 No. 4, December 2000.

[35] G. Weikum. DB&IR: Both Sides Now. SIGMOD Keynote Talk, 2007.

[36] Wikipedia. http://en.wikipedia.org/wiki/Main_Page

[37] B. Yu, G. Li, B. Ooi, L. Zhou. One Table Stores All: Enabling Painless Free-and-Easy Data Publishing and Sharing. CIDR 2007