Information Extraction Challenges in Managing Unstructured Data

AnHai Doan, Jeffrey F. Naughton, Raghu Ramakrishnan, Akanksha Baid, Xiaoyong Chai, Fei Chen, Ting Chen, Eric Chu, Pedro DeRose, Byron Gao, Chaitanya Gokhale, Jiansheng Huang, Warren Shen, Ba-Quy Vuong

University of Wisconsin-Madison

ABSTRACT

Over the past few years, we have been trying to build an end-to-end system at Wisconsin to manage unstructured data, using extraction, integration, and user interaction. This paper describes the key information extraction (IE) challenges that we have run into, and sketches our solutions. We discuss in particular developing a declarative IE language, optimizing for this language, generating IE provenance, incorporating user feedback into the IE process, developing a novel wiki-based user interface for feedback, best-effort IE, pushing IE into RDBMSs, and more. Our work suggests that *IE in managing unstructured data* can open up many interesting research challenges, and that these challenges can greatly benefit from the wealth of work on *managing structured data* that has been carried out by the database community.

1. INTRODUCTION

Unstructured data, such as text, Web pages, emails, blogs, and memos, is becoming increasingly pervasive. Hence, it is important that we develop solutions to manage such data. In a recent CIDR-09 paper [12] we have outlined an approach to such a solution. Specifically, we propose building *unstructured data management systems (UDMSs)*. Such systems extract structures (e.g., person names, locations) from the raw text data, integrate the structures (e.g., matching "David Smith" with "D. Smith") to build a structured database, then leverage the database to provide a host of user services (e.g., keyword search and structured querying). Such systems can also solicit user interaction to improve the extraction and integration methods, the quality of the resulting database, and the user services.

Over the past few years at Wisconsin we have been attempting to build exactly such a UDMS. Building it has raised many difficult challenges in information extraction, information integration, and user interaction. In this paper we briefly describe the key challenges in information extraction (IE) that we have faced, sketch our solutions, and discuss future directions (see [11, 10] for a discussion of non-IE challenges). Our work suggests that managing unstructured data can open up many interesting IE directions for database researchers. It further suggests that these directions can greatly benefit from the vast body of work on managing structured data that has been carried out in our community, such as work on data storage, query optimization, and concurrency control.

The work described here has been carried out in the context of the Cimple project. Cimple started out trying to build community information management systems: those that manage data for online communities, using extraction, integration, and user interaction [13]. Over time, however, it became clear that such systems can be used to manage unstructured data in many contexts beyond just online communities. Hence, Cimple now seeks to build such a general-purpose unstructured data management system, then apply it to a broad variety of applications, including community information management [13], personal information management [3], best-effort/on-the-fly data integration [17], and dataspaces [14] (see `www.cs.wisc.edu/~anhai/projects/cimple` for more detail on the Cimple project).

The rest of this paper is organized as follows. In Sections 2-4 we describe key IE challenges in developing IE programs, interacting with users during the IE process, and leveraging RDBMS technology for IE. Then in Section 5 we discuss how the above individual IE technologies can be integrated and combined with non-IE technologies to build an end-to-end UDMS. We conclude in Section 6.

2. DEVELOPING IE PROGRAMS

To extract structures from the raw data, developers often must create and then execute one or more *IE programs*. Today, developers typically create such IE programs by "stitching together" smaller IE modules (obtained externally or written by the developers themselves), using, for example, C++, Perl, or Java. While powerful, this *procedural* approach generates large IE programs that are difficult to develop, understand, debug, modify, and optimize. To address this problem, we have developed xlog, a declarative language in which to write IE programs. We now briefly describe xlog and then techniques to optimize xlog programs for both static and dynamic data.

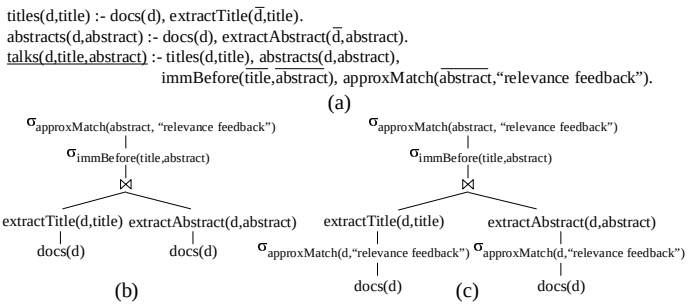

Figure 1: (a) An IE program in xlog, and (b)-(c) two possible execution plans for the program.

The xlog Declarative Language: xlog is a Datalog extension. Each xlog program consists of multiple Datalog-like *rules*, except that these rules can also contain user-defined *procedural predicates* that are pieces of procedural code (e.g., in Perl, Java).

Figure 1.a shows a tiny such xlog program with three rules, which extracts titles and abstracts of those talks whose abstracts contain "relevance feedback." Consider the first rule. Here $docs(d)$ is an extensional predicate (in the usual Datalog sense) that represents a set of text documents, whereas the term $extractTitle(\overline{d}, title)$ is a procedural predicate, i.e., a piece of code that takes as input a document d, and produces as output a set of tuples $(d, title)$, where $title$ is a talk title in document d. The first rule thus extracts all talk titles from the documents in $docs(d)$. Similarly, the second rule extracts all talk abstracts from the same documents. Finally, the third rule pairs the titles and abstracts, then retains only those where the title is immediately before the abstract and the abstract contains "relevance feedback" (allowing for misspellings and synonym matching).

The language xlog therefore allows developers to write IE programs by stitching together multiple IE "black-boxes" (e.g., $extractTitle$, $extractAbstract$, etc.) using declarative rules instead of procedural code. Such an IE program can then be converted into an execution plan and evaluated by the UDMS. For example, Figure 1.b shows a straightforward execution plan for the IE program in Figure 1.a. This plan extracts titles and abstracts, selects only those (title,abstract) pairs where the title is immediately before the abstract, then selects further only those pairs where the abstract contains "relevance feedback." In general, such a plan can contain both relational operators (e.g., $\bowtie$) and user-defined operators (e.g., $extractTitle$).

Optimizing xlog Programs: A key advantage of IE programs in xlog, compared to those in procedural languages, is that they are highly amenable to query optimization techniques. For example, consider again the execution plan in Figure 1.b. Recall that this plan retains only those (title,abstract) pairs where the abstract contains "relevance feedback." Intuitively, an abstract in a document d cannot possibly contain "relevance feedback" unless d itself also contains "relevance feedback." This suggests that we can "optimize" the above plan by discarding a document d as soon as we find out that d does not contain "relevance feedback" (a technique reminiscent of pushing down selection in relational contexts). Figure 1.c shows the resulting plan.

Of course, whether this plan is more efficient than the first plan depends on the selectivity of the selection operator $\sigma_{approxMatch(d,''relevance\ feedback'')}$ and the runtime cost of $approxMatch$. If a data set mentions "relevance feedback" frequently (as would be the case, for example, in SIGIR proceedings), then the selection selectivity will be low. Since $approxMatch$ is expensive, the second plan can end up being significantly worse than the first one. On the other hand, if a data set rarely mentions "relevance feedback" (as would likely be the case, for example, in SIGMOD proceedings), then the second plan can significantly outperform the first one. One way to address this choice of plans is to perform cost-based optimization, like in relational query optimization.

In [18] we have developed such a cost-based optimizer. Given an xlog program P, the optimizer conceptually generates an execution plan for P, employs a set of rewriting rules (such as pushing down a selection, as described above) to generate promising plan candidates, then selects the candidate with the lowest estimated cost, where the costs are estimated using a cost model (in the same spirit as relational query optimization). The work [18] describes the optimizer in detail, including techniques to efficiently search for the best candidate in the often huge candidate space.

Optimizing for Evolving Data: So far we have considered only *static* text corpora, over which we typically have to apply an xlog program only once. In practice, however, text corpora often are *dynamic*, in that documents are added, deleted, and modified. They evolve over time, and to keep extracted information up to date, we often must apply an xlog program *repeatedly*, to consecutive corpus snapshots. Consider, for example, DBLife, a structured portal for the database community that we have been developing [8, 9]. DBLife operates over a text corpus of 10,000+ URLs. Each day it recrawls these URLs to generate a 120+ MB corpus snapshot, and then applies an IE program to this snapshot to find the latest community information.

In such contexts, applying IE to each corpus snapshot *in isolation, from the scratch*, as typically done today, is very time consuming. To address this problem, in [5] we have developed a set of techniques to efficiently execute an xlog program over an evolving text corpus. The key idea underlying our solution is to recycle previous IE results, given that consecutive snapshots of a text corpus often contain much overlapping content. For example, suppose that a corpus snapshot contains the text fragment "the Cimple project will meet in room CS 105 at 3pm", from which we have extracted "CS 105" as a room number. Then when we see the above text fragment again in a new snapshot, under certain conditions (see [5]) we can immediately conclude that "CS 105" is a room number, without re-applying the IE program to the text fragment.

Overall, our work has suggested that xlog is highly

promising as an IE language. It can seamlessly combine procedural IE code fragments with declarative ones. In contrast to some other recent efforts in declarative IE languages (e.g., UIMA at `research.ibm.com/UIMA`), xlog builds on the well-founded semantics of Datalog. As such, it can naturally and rigorously handle recursion (which occurs quite commonly in IE [1, 2]). Finally, it can also leverage the wealth of execution and optimization techniques already developed for Datalog. Much work remains, however, as our current xlog version is still rudimentary. We are currently examining how to extend it to handle negation and recursion, and to incorporate information integration procedures (see Section 5), among others.

3. INTERACTING WITH USERS

Given that IE is an inherently imprecise process, user interaction is important for improving the quality of IE applications. Such interaction often can be solicited. Many IE applications (e.g., DBLife) have a sizable development team (e.g., 5-10 persons at any time). Just this team of developers *alone* can already provide a considerable amount of feedback. Even more feedback can often be solicited from the multitude of application users, in a Web 2.0 style.

The goal then is to develop techniques to enable efficient user interaction (where by "user" we mean both developers and application users). Toward this goal, we have been pursuing four research directions: explain query result provenance, incorporating user feedback, developing novel user interfaces, and developing novel interaction modes. We now briefly explain these directions.

Generating the Provenance of Query Result: Much work has addressed the problem of generating the provenance of query results [20]. But this work has focused only on *positive provenance*: it seeks to explain why an answer is produced.

In many cases, however, a user may be interested in *negative provenance*, i.e., why a certain answer is *not* produced. For example, suppose we have extracted two tables TALKS(talk-title, talk-time, room) and LOCATIONS(room,building) from text documents. Suppose the user now asks for the titles of all talks that appear at 3pm in Dayton Hall. This requires joining the above two tables on "room", then selecting those where "talk-time" is 3pm and "building" is Dayton Hall. Suppose the user expects a particular talk with title "Declarative IE" to show up in the query result, and is surprised that it does not. Then the user may want to ask the system why this talk does not show up. We call such requests "asking for the provenance of a non-answer". Such non-answer provenance is important because it can provide more confidence in the answer for the user, and can help developers debug the system.

In [15] we have developed an initial approach to providing the provenance of non-answers. In the above example, for instance, our solution can explain that no tuple with talk-title = "Declarative IE" and talk-time = 3pm has been extracted into the table TALKS, and that if such a tuple were to be extracted, then the non-answer will become an answer. Alternatively, our approach can explain that such a tuple indeed has been extracted into table TALKS, but that the tuple does not join with any tuple in table LOCATIONS, and so forth.

Incorporate User Feedback: Consider again the IE program P in Figure 1.b, which extracts titles and abstracts, pairs them, then retains only those satisfying certain conditions. Conceptually, this program can be viewed as an execution tree (in the spirit of an RDBMS execution tree), where the leaves specify input data (the table $docs(d)$ of text documents in this case), the internal nodes specify relational operations (e.g., join, select), IE operations (e.g., $extractTitle$), or procedures (e.g., $immBefore$), and the root node specifies the output (which is the table $talks(d, title, abstract)$ in this case).

Executing the above program then amounts to a bottom-up execution of the above execution tree. After the execution, a user may inspect and correct mistakes in the output table $talks(d, title, abstract)$. For example, he or she can modify a title, remove a tuple that does not correspond to a correct pair of title and abstract, or add a tuple that the IE modules fail to extract.

But the user may go even further. If during the execution we have materialized the intermediate tables (that are produced at internal nodes of the above execution tree), then the user can also correct those. For example, the user may try to correct the intermediate table $titles(d, title)$ (the output of the node associated with the IE module $extractTitle$), then propagate these corrections "up the tree". Clearly, correcting a mistake "early" can be highly beneficial as it can drastically reduce the number of incorrect tuples "further up the execution tree".

Consequently, in recent work [4] we have developed an initial solution that allows users to correct mistakes *anywhere* during the IE execution, and then propagate such corrections up the execution tree. This raises many interesting and difficult challenges, including (a) developing a way to quickly specify which parts of the data are to be corrected and in what manner, (b) redefining the semantics of the declarative program, in the presence of user corrections, (c) propagating corrections up the tree, but figuring out how to reconcile them with prior corrections, and (d) developing an efficient concurrency control solution for the common case where multiple users concurrently correct the data.

[4] addresses the above challenges in detail. Here, we briefly focus on just the first challenge: how to quickly specify which parts of the data are to be corrected and in what manner. To address this challenge, our solution allows developers to write declarative "human interaction" (HI) rules. For example, after writing the IE program in Figure 1.a, a developer may write the following HI rule:

```
extracted-titles(d,title)#spreadsheet
                :- titles(d,title), d > 200.
```

This rule states that during the program execution, a

view *extracted-titles*$(d, title)$ over table *titles*$(d, title)$ (defined by the above rule to be those tuples in the *titles*$(d, title)$ table with the doc id d exceeding 200) should be materialized, then exposed to users to edit via a spreadsheet user interface (UI). Note that the system comes pre-equipped already with a set of UIs. The developer merely needs to specify in the HI rule that which UI is to be used. The system will take care of the rest: materialize the target data part, expose it in the specified UI, incorporate user corrections, and propagate such corrections "up the execution tree."

Develop Novel User Interfaces: To correct the extracted data, today users can only use a rather limited set of UIs, such as spreadsheet interface, form interface, and GUI. To maximize user interaction with the UDMS, we believe it is important to develop a richer set of UIs, because then a user is more likely to find an UI that he or she is comfortable with, and thus is more likely to participate in the interaction.

Toward this goal, we have recently developed a wiki-based UI [7] (based on the observation that many users increasingly use wikis to collect and correct data). This UI exposes the data to be corrected in a set of wiki pages. Users examine and correct these pages, then propagate the correction to the underlying data. For example, suppose the data to be corrected is the table *extracted-titles*$(d, title)$ mentioned earlier (which is a view over table *titles*$(d, title)$). Then we can display the tuples of this table in a wiki page. Once a user has corrected, say, the first tuple of the table, we can propagate the correction to the underlying table *titles*$(d, title)$.

A distinguishing aspect of the wiki UI is that in addition to correcting *structured data* (e.g., relational tuples), users can also easily add comments, questions, explanations, etc. in *text* format. For example, after correcting the first tuple of table *extracted-titles*$(d, title)$, a user can leave a comment (right next to this tuple in the wiki page) stating why. Or another user may leave a comment questioning the correctness of the second and third tuples, such as "these two tuples seem contradictory, so at least one of them is likely to be wrong". Such text comments are then stored in the system together with the relational tables. The comments clearly can also be accommodated in traditional UIs, but not as easily or naturally as in a wiki-based UI.

Developing such a wiki-based UI turned out to raise many interesting challenges. A major challenge is how to display the structured data (e.g., relational tuples) in a wiki page. The popular current solution of using a natural-text or wiki-table format makes it easy for users to edit the data, but very hard for the system to figure out afterwards which pieces of structured data have been edited. Another major challenge is that after a user U has revised a wiki page P into a page P' and has submitted P' to the system, how does the system know which sequence of edit actions U actually intended (as it is often the case that many different edit sequences can transform P into P')?. Yet another challenge is that once the system has found the intended edit sequence, how can it efficiently propagate this sequence to the underlying data? Our recent work [7] discusses these challenges in detail and proposes initial solutions.

Develop Novel Modes of User Interaction: So far we have discussed the following mode of user interaction for UDMSs: a developer U writes an IE program P, the UDMS executes P, then U (and possibly other users) interacts with the system to improve P's execution. We believe that this mode of user interaction is not always appropriate, and hence we have been interested in exploring novel modes of user interaction.

In particular, we observe that in the above traditional mode, developer U must produce a *precise* IE program P (one that is fully "fleshed out"), before P can be executed and then exposed for user interaction. As such this mode suffers from three limitations. First, it is often difficult to execute partially specified IE programs and obtain meaningful results, thereby producing a long "debug loop". Second, it often takes a long time before we can obtain the first meaningful result (by finishing and running a precise IE program), thereby rendering this mode impractical for time-sensitive IE applications. Finally, by writing precise IE programs U may also waste a significant amount of effort, because an approximate result – one that can be produced quickly – may already be satisfactory.

To address these limitations, in [17] we have developed a novel IE mode called *best-effort IE* that interleaves IE execution with user interaction from the start. In this mode, U uses an xlog extension called alog to quickly write an initial approximate IE program P (with a possible-worlds semantics). Then U evaluates P using an approximate query processor to quickly extract an approximate result. Next, U examines the result, and further refines P if necessary, to obtain increasingly more precise results. To refine P, U can enlist a *next-effort assistant*, which suggests refinements based on the data and the current version of P.

To illustrate, suppose that given 500 Web pages, each listing a house for sale, developer U wants to find all houses whose price exceeds \$500000. Then to start, U can quickly write an initial approximate IE program P, by specifying what he or she knows about the target attributes (i.e., price in this case). Suppose U specifies only that price is numeric, and suppose further that there are only nine house pages where each page contains at least one number exceeding 500000. Then the UDMS can immediately execute P to return these nine pages as an "approximate superset" result for the initial extraction program. Since this result set is small, U may already be able to sift through and find the desired houses. Hence, U can already stop with the IE program.

Now suppose that instead of nine, there are actually 120 house pages that contain at least one number exceeding 500000. Then the system will return these 120 pages. U realizes that the IE program P is "underspecified", and hence will try to refine it further (to "narrow" the result set). To do so, U can ask the next-effort assistant to suggest what to focus on next. Suppose that this module suggests to check if price is in bold font,

and that after checking, U adds to the IE program that price is in bold font. Then the system can leverage this "refinement" to reduce the result set to only 35 houses. U now can stop, and sift through the 35 houses to find the desired ones. Alternatively, U can try to refine the IE program further, enlisting the next-effort assistant whenever appropriate.

In [17] we describe in detail the challenges of best-effort IE and proposes a set of possible solutions.

4. LEVERAGING RDBMS TECHNOLOGIES

So far we have discussed how to develop declarative IE programs and effective user interaction tools. We now turn our attention to efficiently implementing such programs.

We begin by observing that most of today's implementations perform their IE without the use of an RDBMS. A very common method, for example, is to store text data in files, write the IE program as a script, or in a recently developed declarative language (e.g., xlog [18], AQL of System-T [16], UIMA at `research.ibm.com/UIMA`), then execute this program over these text files, using the file system for all storage.

This method indeed offers a good start. But given that IE programs fundamentally extract and manipulate *structured data*, and that RDBMSs have had a 30-year history of managing structured data, a natural question arises: *Do RDBMSs offer any advantage over file systems for IE applications?* In recent work [6, 19], we have explored this question, provided an affirmative answer, and further explored the natural follow-on questions of *How can we best exploit current RDBMS technology to support IE?* and *How can current RDBMS technology be improved to better support IE?*. For space reasons, in what follows we will briefly describe only the work in [19], our latest work on the topic.

We begin in [19] by showing that executing and managing IE programs (such as those discussed so far in this paper) indeed require many capabilities offered by current RDBMSs. First, such programs often execute many relational operations (e.g., joining two large tables of extracted tuples). Second, the programs are often so complex or run over so much data that they can significantly benefit from indexing and optimization. Third, many such programs are long running, and hence crash recovery can significantly assist in making program execution more robust. Finally, many such programs and their data (i.e., input, output, intermediate results) are often edited concurrently by multiple users (as discussed earlier), raising difficult concurrency control issues.

Given the above observations, in the file-based approach the developers of IE programs can certainly develop all of the above capabilities. But such development would be highly non-trivial, and could duplicate substantial portions of the 30-year effort the DBMS community has spent developing RDBMS capabilities.

Consequently, leveraging RDBMS for IE seems like an idea that is worth exploring, and in [19] we outline a way to do so. First, we identify a set of core operations on text data that IE programs often perform. Examples of core operations include retrieving the content of a text span given its start and end positions in a document, verifying a certain property of a text span (e.g., whether it is in bold font, to support for instance best-effort IE as discussed in Section 3), and locating all substrings (of a given text span) that satisfy certain properties.

We then explore the issue of how to store text data in an RDBMS in a way that is suitable for IE, and how to build indexes over such data to speed up the core IE operations. We show that if we divide text documents into "chunks", and making this "chunking" visible to the IE operation implementations, we can exploit certain properties of these core operations to optimize data access. Furthermore, if we have sufficiently general indexing facilities, we can use indexes both to speed the retrieval of relevant text and to cache the results of function invocations, thereby avoiding repeatedly inferring useful properties of that text.

We then turn our attention to the issue of executing and optimizing IE programs within RDBMS. We show that IE programs can significantly benefit from traditional relational query optimization and show how to leverage the RDBMS query optimizer to help optimize IE programs. Finally, we show how to apply text-centric optimization (as discussed in Section 2) in conjunction with leveraging the RDBMS query optimizer. Overall, our work suggests that exploiting RDBMSs for IE is a highly promising direction in terms of possible practical impacts as well as interesting research challenges for the database community.

5. BUILDING AN END-TO-END UDMS

So far we have discussed the technologies to solve individual IE challenges. We now discuss how these technologies are being integrated to build an end-to-end prototype UDMS, an ongoing effort at Wisconsin. In what follows, our discussion will also involve information integration (II), as the UDMS often must perform both extraction and integration over the raw text data.

Figure 2 shows the architecture of our planned UDMS prototype. This architecture consists of four layers: the physical layer, the data storage layer, the processing layer, and the user layer. We now briefly discuss each layer, highlighting in particular our ongoing IE efforts and opportunities for further IE research.

The Physical Layer: This layer contains hardware that runs all the steps of the system. Given that IE and II are often very computation intensive and that many applications involve a large amount of data, the ultimate system will probably need parallel processing in the physical layer. A popular way to achieve this is to use a computer cluster (as shown in the figure) running Map-Reduce-like processes.

For now, for simplicity we plan to build the UDMS to run on a single machine. In the long run, however, it would be an important and interesting research direction to study how to run all steps of the system on a cluster of machines, perhaps using a Map-Reduce-like framework. This will require, among other tasks, de-

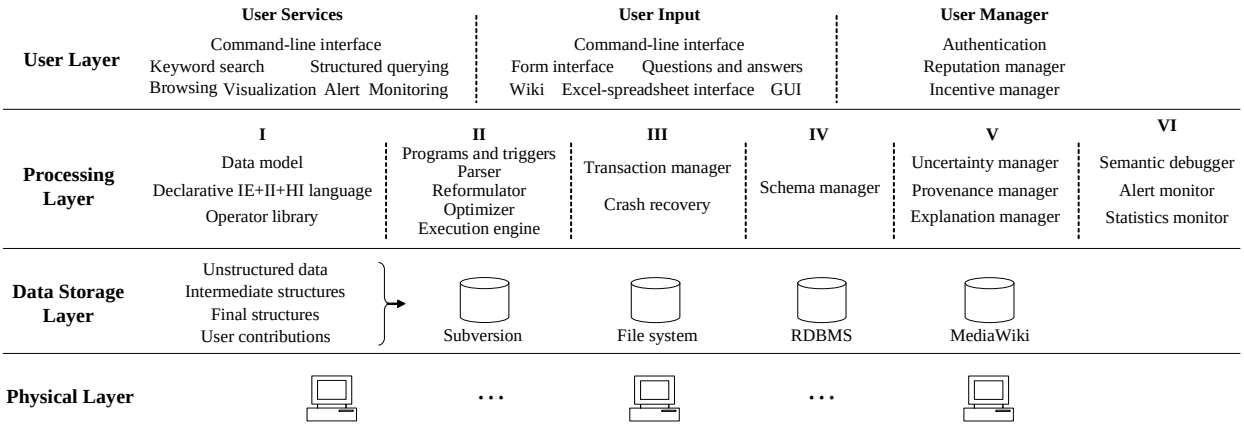

Figure 2: The architecture of our planned UDMS prototype.

composing a declarative IE/II program so that it can run efficiently and correctly over a machine cluster.

The Data Storage Layer: This layer stores all forms of data: the original data, intermediate structured data (kept around, for example, for debugging, user feedback, or optimization purposes), the final structured data, and user feedback. These different forms of data have very different characteristics, and may best be kept in different storage systems, as depicted in the figure (of course, other choices are possible, such as developing a single unifying storage system).

For example, if the original data is retrieved daily from a collection of Web sites, then the daily snapshots will overlap a lot, and hence may be best stored in a system such as Subversion, which only stores the "diff" across the snapshots, to save space. As another example, the system often executes only sequential reads and writes over intermediate structured data, in which case such data can best be kept in a file system.

For the prototype system, we will utilize a variety of storage systems, taking into account our work on storing certain parts of the IE process in RDBMSs (Section 4). Future research can then study what should be the best storage solution under which condition.

The Processing Layer: This layer is responsible for specifying and executing IE/II processes. At the heart of this layer is a data model (which is the relational data model in our current work), a declarative IE+II+HI language (over this data model), and a library of basic IE/II operators (see Part I of this layer in the figure). We envision that the above IE+II+HI declarative language will be a variant of xlog, extended with certain II features, then with HI (i.e., human interaction) rules such as those discussed in Section 3.

Developers can then use the language and operators to write declarative IE/II programs that specify how to extract and integrate the data and how users should interact with the extraction/integration process. These programs can be parsed, reformulated (to subprograms that are executable over the storage systems in the data storage layer), optimized, then executed (see Part II in the figure). Note that developers may have to write domain-specific operators, but the framework makes it easy to use such operators in the programs.

The remaining four parts, Parts III-VI in the figure, contain modules that provide support for the IE/II process. Part III handles transaction management and crash recovery. Part IV manages the schema of the derived structure. Part V handles the uncertainty that arise during the IE/II processes. It also provides the provenance for the derived structured data.

Part VI contains an interesting module called the "semantic debugger." This module learns as much as possible about the application semantics. It then monitors the data generation process, and alerts the developer if the semantics of the resulting structure are not "in sync" with the application semantics. For example, if this module has learned that the monthly temperature of a city cannot exceed 130 degrees, then it can flag an extracted temperature of 135 as suspicious. This part also contains modules to monitor the status of the entire system and alert the system manager if something appears to be wrong.

We are currently developing technical innovations for Parts I-II of the processing layer, as discussed throughout the paper. We are not working on the remaining parts of this layer, opting instead to adapt current state-of-the-art solutions.

The User Layer: This layer allows users (i.e., both lay users and developers) to exploit the data as well as provide feedback to the system. The part "User Services" contains all common data exploitation modes, such as command-line interface (for sophisticated users), keyword search, structured querying, etc. The part "User Input" contains a variety of UIs that can be used to solicit user feedback, such as command-line interface, form interface, question/answering, and wiki-based UI, as discussed in Section 3 (see the figure).

We note that modules from both parts will often be combined, so that the user can also conveniently provide feedback while querying the data, and vice versa. Finally, this layer also contains modules that authenti-

cate users, manage incentive schemes for soliciting user feedback, and manage user reputation data (e.g., for mass collaboration).

For this part, we are developing several user services based on keyword search and structured querying, as well as several UIs, as discussed in Section 3. When building the prototype system, we plan to develop other modules for this layer only on an as-needed basis.

6. CONCLUDING REMARKS

Unstructured data has now permeated numerous real-world applications, in all domains. Consequently, managing such data is now an increasingly critical task, not just to our community, but also to many others, such as the Web, AI, KDD, and SIGIR communities.

Toward solving this task, in this paper we have briefly discussed our ongoing effort at Wisconsin to develop an end-to-end solution that manages unstructured data. The discussion demonstrates that handling such data can raise many information extraction challenges, and that addressing these challenges requires building on the wealth of data management principles and solutions that have been developed in the database community. Consequently, we believe that our community is well positioned to play a major role in developing IE technologies in particular, and in managing unstructured data in general.

Acknowledgment: This work is supported by NSF grants SCI-0515491, Career IIS-0347943, an Alfred Sloan fellowship, an IBM Faculty Award, a DARPA seedling grant, and grants from Yahoo, Microsoft, and Google.

7. REFERENCES

[1] E. Agichtein, L. Gravano, J. Pavel, V. Sokolova, and A. Voskoboynik. Snowball: A prototype system for extracting relations from large text collections. In *SIGMOD*, 2001.

[2] S. Brin. Extracting patterns and relations from the world wide web. In *WebDB*, 1998.

[3] Y. Cai, X. Dong, A. Y. Halevy, J. Liu, and J. Madhavan. Personal information management with semex. In *SIGMOD*, 2005.

[4] X. Chai, B. Vuong, A. Doan, and J. F. Naughton. Efficiently incorporating user interaction into extraction and integration programs. Technical Report UW-CSE-2008, University of Wisconsin-Madison, 2008.

[5] F. Chen, A. Doan, J. Yang, and R. Ramakrishnan. Efficient information extraction over evolving text data. In *ICDE*, 2008.

[6] E. Chu, A. Baid, T. Chen, A. Doan, and J. F. Naughton. A relational approach to incrementally extracting and querying structure in unstructured data. In *VLDB*, 2007.

[7] P. DeRose, X. Chai, B. Gao, W. Shen, A. Doan, P. Bohannon, and X. Zhu. Building community wikipedias: A machine-human partnership approach. In *ICDE*, 2008.

[8] P. DeRose, W. Shen, F. Chen, A. Doan, and R. Ramakrishnan. Building structured web community portals: A top-down, compositional, and incremental approach. In *VLDB*, 2007.

[9] 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 (demo). In *CIDR*, 2007.

[10] A. Doan. Data integration research challenges in community information management systems, 2008. Keynote talk, Workshop on Information Integration Methods, Architectures, and Systems (IIMAS) at ICDE-08.

[11] A. Doan, P. Bohannon, R. Ramakrishnan, X. Chai, P. DeRose, B. Gao, and W. Shen. User-centric research challenges in community information management systems. *IEEE Data Engineering Bulletin*, 30(2):32–40, 2007.

[12] A. Doan, J. F. Naughton, A. Baid, X. Chai, F. Chen, T. Chen, E. Chu, P. DeRose, B. Gao, C. Gokhale, J. Huang, W. Shen, and B. Vuong. The case for a structured approach to managing unstructured data. In *CIDR*, 2009.

[13] A. Doan, R. Ramakrishnan, F. Chen, P. DeRose, Y. Lee, R. McCann, M. Sayyadian, and W. Shen. Community information management. *IEEE Data Engineering Bulletin*, 29(1):64–72, 2006.

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

[15] J. Huang, T. Chen, A. Doan, and J. F. Naughton. On the provenance of non-answers to queries over extracted data. *PVLDB*, 1(1):736–747, 2008.

[16] R. Krishnamurthy, Y. Li, S. Raghavan, F. Reiss, S. Vaithyanathan, and H. Zhu. Systemt: A system for declarative information extraction, 2008. SIGMOD Record, Special Issue on Managing Information Extraction.

[17] W. Shen, P. DeRose, R. McCann, A. Doan, and R. Ramakrishnan. Toward best-effort information extraction. In *SIGMOD*, 2008.

[18] W. Shen, A. Doan, J. F. Naughton, and R. Ramakrishnan. Declarative information extraction using datalog with embedded extraction predicates. In *VLDB*, 2007.

[19] W. Shen, C. Gokhale, J. Patel, A. Doan, and J. F. Naughton. Relational databases for information extraction: Limitations and opportunities. Technical Report UW-CSE-2008, University of Wisconsin-Madison, 2008.

[20] W. C. Tan. Provenance in databases: Past, current, and future. *IEEE Data Eng. Bull.*, 30(4):3–12, 2007.