Building Community Wikipedias: A Machine-Human Partnership Approach

Pedro DeRose [1], Xiaoyong Chai [1], Byron J. Gao [1], Warren Shen [1]
AnHai Doan [1], Philip Bohannon [2], Xiaojin Zhu [1]

[1]*University of Wisconsin-Madison* [2]*Yahoo Research*

Abstract— **The rapid growth of Web communities has motivated many solutions for building community data portals. These solutions follow roughly two approaches. The first approach (e.g., Libra, Citeseer, Cimple) employs semi-automatic methods to extract and integrate data from a multitude of data sources. The second approach (e.g., Wikipedia, Intellipedia) deploys an initial portal in wiki format, then invites community members to revise and add material. In this paper we consider combining the above two approaches to building community portals. The new hybrid machine-human approach brings significant benefits. It can achieve broader and deeper coverage, provide more incentives for users to contribute, and keep the portal more up-to-date with less user effort. In a sense, it enables building "community wikipedias", backed by an underlying structured database that is continuously updated using automatic techniques. We outline our ideas for the new approach, describe its challenges and opportunities, and provide initial solutions. Finally, we describe a real-world implementation and preliminary experiments that demonstrate the utility of the new approach.**

I. Introduction

The growing presence of Web communities has motivated many solutions to build community data portals. These solutions follow roughly two approaches. The first, *machine-based*, approach employs semi-automatic methods to extract and integrate data from a multitude of data sources, to create structured data portals. Examples include Cimple, Libra, Rexa, BlogScope, and Blogosphere [1], [2], [3], [4], [5], [6].

The above approach incurs relatively little human effort, often generates a reasonable initial portal, keeps portals fresh with automatic updates, and enables structured services (querying, browsing, etc.) over portals. However, it usually suffers from inaccuracies, caused by imperfect extraction and integration methods, and limited coverage, because it can only infer whichever information is available in the data sources.

The second, *human-based*, approach manually deploys an initial portal in wiki format, then invites community users to revise and add materials. Examples include Wikipedia, Intellipedia, umasswiki.com, ecolicommunity.org, and many wiki-based intranets. This approach avoids many problems of the machine-based approach, but suffers from its own limitations. In particular, it may be difficult to solicit sufficient user participation, can incur significant user effort to keep portals up to date, and cannot accommodate structured services, because users contribute mostly text and images.

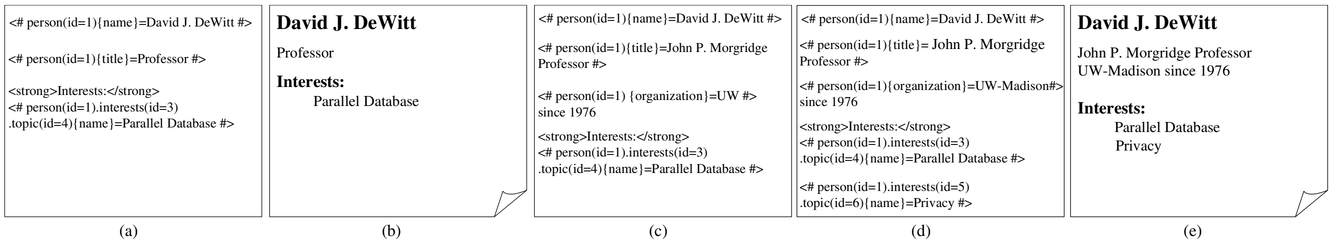
In this paper we consider combining the above two complementary approaches to build community portals. Specifically, we use "machines" to deploy an initial portal in wiki format, then allow *both* machines and human users to revise and add materials. Machines can add structured information to certain parts of wiki pages, while users can add both text and structured information. Machines and human can also correct and augment each other's contributions, in a synergistic fashion. We refer to this approach as Madwiki (shorthand for <u>M</u>achine <u>a</u>ssisted <u>d</u>evelopment of <u>wiki</u>pedias). The following example illustrates the approach.

Example 1.1: Suppose we apply Madwiki *to build a portal for the database community. We can start by applying a semi-automatic approach (i.e., "machines") to extract structured data from the Web, then use the data to create and deploy wiki pages, such as page W in Figure 1.a. Page W contains "structured data pieces" mixed with ordinary wiki text, and will display as the HTML page in Figure 1.b. In effect, W describes a person entity who has three attributes: id = 1, name = "David J. DeWitt", and title = "Professor". This person also participates in a relationship called "interests" with an entity of type "topic", whose name is "Parallel Database".*

Once W has been deployed, a user U may come in and edit page W, e.g., by correcting the value of attribute title from "Professor", which was generated by machines, to "John P. Morgridge Professor". U may also contribute a structured data piece "$<\#$ person(id=1)$\{$organization$\}$= UW #$>$", to state that this person is working for an organization called "UW". Finally, U adds free text "since 1976" after this data piece. The edited page W' is shown in Figure 1.c.

Later a machine M may discover from data sources that the above person also participates in "interests" relationship with topic "Privacy". M can then add this piece of information to the page, as "$<\#$ person(id=1).interests (id=5).topic(id=6)$\{$name$\}$=Privacy #$>$". With high confidence, M may also correct the value of attribute organization from "UW", which was contributed by U, to "UW-Madison". The resulting wiki page W'' is in Figure 1.d, and it will display as the HTML page in Figure 1.e. Thus, page W has evolved over time, with both machines and users contributing and correcting each other's contributions. □

As described, this new hybrid machine-human approach enables building "community wikipedias" that are backed by an underlying structured database that is continuously updated using automatic techniques. The approach can bring significant benefits. First, it can achieve broader and deeper coverage, because it exploits both machines and human users. Second, it can provide more incentives for users to contribute, because the initial portal built by machines can already be reasonably useful and comprehensive, thus motivating users to further improve it. Third, it can keep the portal more up-to-date, with less user effort, because machines can continuously monitor data sources and update certain parts of the portal. Finally, the structured data in the wiki pages of the portal is also stored



```
<# person(id=1){name}=David J. DeWitt #>

<# person(id=1){title}=Professor #>

<strong>Interests:</strong>
<# person(id=1).interests(id=3)
.topic(id=4){name}=Parallel Database #>
```

(a)

David J. DeWitt

Professor

Interests:
Parallel Database

(b)

```
<# person(id=1){name}=David J. DeWitt #>

<# person(id=1){title}=John P. Morgridge
Professor #>

<# person(id=1) {organization}=UW #>
since 1976

<strong>Interests:</strong>
<# person(id=1).interests(id=3)
.topic(id=4){name}=Parallel Database #>
```

(c)

```
<# person(id=1){name}=David J. DeWitt #>

<# person(id=1){title}= John P. Morgridge
Professor #>

<# person(id=1){organization}=UW-Madison#>
since 1976

<strong>Interests:</strong>
<# person(id=1).interests(id=3)
.topic(id=4){name}=Parallel Database #>

<# person(id=1).interests(id=5)
.topic(id=6){name}=Privacy #>
```

(d)

David J. DeWitt

John P. Morgridge Professor
UW-Madison since 1976

Interests:
Parallel Database
Privacy

(e)

Fig. 1. An example to illustrate the machine-human approach.

in an underlying structured database, thus enabling a variety of structured services over the portal.

In the rest of the paper we elaborate on the above approach. First, we consider how to build an initial wiki-based portal, using machines. We cast this as a *view creation* problem: store the data generated by machines in a structured database G, create structured views over G, then export the views in wiki pages. The key questions are then: How to model and implement the structured database G? What should be the view language? And how to export the structured data of the views into wiki pages? As parts of our solution, we represent the machine-generated data using an entity-relationship (ER) model, define a path-based view language over this model, extend the standard wiki language [7] with *s-slots* – constructs to embed structured data into the natural text of wiki pages, then show how to export the views in wiki pages, using s-slots (Section IV-C).

Next, we consider how to manage user contributions to the portal. If a user U has edited a wiki page W, then we want to extract the "structured" part of U's edits, and "push" it all the way into the underlying database G. The key questions here are: What is it that U is conceptually allowed to edit? And how to efficiently infer such edits based on what U has done to a wiki page W? To answer these questions, we cast the problem of processing user contributions as a problem of mapping U's edits over the wiki page into edits over the corresponding view, then from this view into edits over G. This is a *view update* problem. But it is complicated (compared to RDBMS view update) by the facts that here (a) U can also edit the *schema*, not just the data, of the view, and (b) U's edits, being limited to the wiki interface, are often ambiguous. Furthermore, after we have updated database G with edits from W, we must decide how to propagate this update to other views and corresponding wiki pages. In Section V we elaborate on these issues, then provide a solution.

Finally, for the sake of completeness (but not as a part of the contribution of this paper), in Section VI we briefly touch upon the problem of managing *multiple* users, where we extend current solutions employed in Wikipedia (namely, optimistic concurrency control and access rights based on a user hierarchy) to handle concurrent editing and malicious users. We also consider how to let machines join users in updating the portal. The key challenge is the following: once a user has entered an edit, can machines be allowed to overwrite the edit, and when?

We have been applying the above solution to build a community wikipedia for the database community (see the live system at [8]). In Section VII we report on our experience and preliminary experiments that demonstrate the potentials of this approach, and suggest opportunities for future research.

To summarize, we make the following contributions:

- Introduce a new hybrid approach that employs both machines and human users to build community portals, backed by an underlying structured database. As far as we know, ours is the first work that studies this direction in depth.
- Provide solutions to modeling the underlying structured database, representing views over this database with a path-based language, and exporting these views in wiki pages.
- Provide an efficient solution to process user edits in wiki pages and "push" these edits into the underlying database. The solution recasts this problem as translating edits across different user interfaces.
- Empirical results over a real-world implementation that demonstrates the promise of the approach and suggests opportunities for future research.

II. Related Work

We are not aware of any published work that has studied combining machine-based approaches and human-based approaches to building community portals. Many portals (e.g., Wikipedia) do employ automatic programs (called "bots") to generate new pages according to some template, and to detect problems (e.g., vandalism) with current pages. But these programs do not contribute structured data nor do they update existing data, as we do here.

Perhaps the work closest to ours is Semantic Wikipedia [9]. This work develops new wiki language constructs that allow users to add structured data to wiki pages. We also develop similar wiki language constructs (see Section IV-C). But our constructs are far more powerful: we can embed arbitrary ER data graphs in a wiki page, whereas the constructs in [9] in a sense only allow embedding node and relation *attributes*. More importantly, Semantic Wikipedia and several similar efforts, including semantic wikis [10], WikiLens [11], and Metaweb [12], have focused largely on *extending wiki languages* so that *users* can contribute structured data. They have not focused on allowing machines to contribute, nor do they study how to "push" structured contributions from users into an underlying database. Our work here is therefore complementary to these efforts.

Many semi-automatic approaches have been developed to build structured portals (see [13] for a discussion). Any of these can be employed as "machines" in our current work.

Madwiki's s-slots dynamically query the underlying database to insert data into wiki pages. Similar constructs have also been heavily used to query databases to insert data into HTML pages (e.g., PhP, ColdFusion, MS SharePoint). S-slots however differ in that they are used to *both* query and update the underlying database. Specifically, users can modify values in the database simply by editing the appropriate s-slots.

Processing user edits in our context is a variation on the classical view update problem [14], [15]. Unlike relational view update, however, in our context users can also edit the schemas of views as well as of the underlying database. Since users employ the wiki interface, which is rather limited for expressing structured edits, this poses problems in interpreting user intentions that do not arise in relational view updates.

We recast processing structured user edits in our context as a problem of translating these edits across different user interfaces (wiki, ER, and relational, see Section V-B). Such UI translations have been studied, e.g., in translating a natural-language user query into a structured one [16], [17]. Translating free natural-language queries is well known to be difficult [16], [17]. Our problem here is still difficult, but more manageable, as we only translate *structured* edits.

Finally, our work can be viewed as a mass collaboration, Web 2.0 effort to build, maintain, and expand a hybrid structured data-text community database. Mass collaboration approaches to data management have recently received increasing attention in the database community (e.g., mass collaboration panel at VLDB-07, Web 2.0 track at ICDE-08, see also [18], [19], [20], [21], [22], [23]. Our work here contributes to this emerging direction.

III. The Madwiki Approach

In the rest of the paper we describe the Madwiki approach. Figure 2 illustrates how Madwiki works. It starts by applying M, a machine-based solution, to extract and integrate data from a set of data sources, then loads this data into a structured database G. Next, it initializes an empty text database T, which will be used in the future to store text generated by users. Then Madwiki generates structured views over G (e.g., $V_1 - V_3$ in Figure 2), and exports them in wiki pages (e.g., $W_1 - W_3$). The initial portal $\mathcal{W}$ then consists of all such wiki pages.

Community users and machine M then revise and add materials to $\mathcal{W}$. Suppose a user u_1 has revised wiki page W_3 into page W_3' (Figure 2). Then Madwiki extracts the structured data portion V_3' from W_3' and uses it to update the structured database G. Next, Madwiki extracts the text portion T_3' from W_3' and stores it in the text database T. Madwiki also reruns machine M at regular intervals (to obtain the latest information from the data sources), updates G based on the output of M, then updates the views and wiki pages accordingly. Updating a wiki page W_i, for example, means creating a new version of W_i that combines the latest versions of its structured data portion from G and text portion from T. In addition to revising existing wiki pages, as described above, both users and machine M can add new pages or delete existing ones.

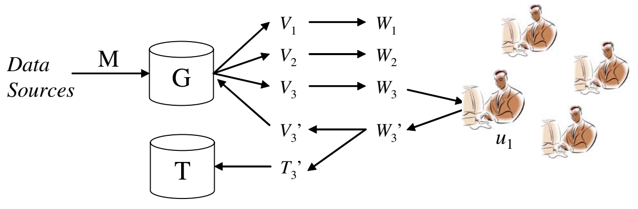


Fig. 2. The Madwiki architecture

The next two sections describe the key contributions of this paper: how to build the initial portal and manage user contributions. Section VI briefly touches upon the issue of managing multiple users and machines. For space reasons, we can only motivate and describe the basic ideas behind our proposed solutions. We refer the reader to the full paper [24], which is available online, for a detailed description of these solutions.

IV. Creating the Initial Community Portal

To create the initial portal, we proceed in three steps: employ a machine M to create a structured database G, create structured views V_i over G, then convert each view V_i into a wiki page W_i.

A. Creating a Structured Database G

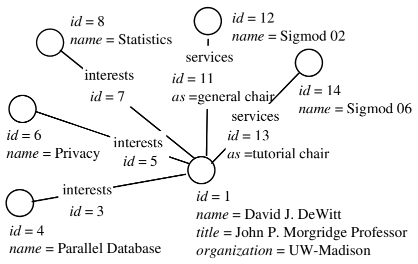
Modeling Database G: To model G, we can choose from a wide variety of data languages. Since the data from G will eventually appear in wiki pages as structured constructs (see Section IV-C for a motivation for this), we had to select a data language that *ordinary, database-illiterate* users are familiar with, and can quickly understand and edit. Since most users are already familiar with the concepts of entity and relationship, as commonly employed by current community portals, we choose an ER language to represent the data in G.

Specifically, we define the schema G_s of G to consist of a set of entity types $E_1, \ldots, E_n$ and a set of relation types $R_1, \ldots, R_m$. Each entity/relation type is specified using a set of attributes. Attributes are either atomic, taking string or numeric values, or set-valued.

Next, we define the data G_d of G to be a temporal ER data graph. This graph contains (a) a set of nodes that specify entity instances (or entities for short when there is no ambiguity), (b) a set of edges that specify relation instances (or relations for short when there is no ambiguity), (c) temporal information regarding attributes, entities, and relations, e.g., when an attribute/entity/relation was created, by which user, when it was deleted, by whom, when it was reinstated, etc. This information will be used in managing users (Section VI). We view machine M as a special user M.

We require G to be a temporal database that captures all changes so far, so that later we can develop undo facilities (not yet considered in this paper). Note also that even if G_s specifies that a person entity has an attribute email, this attribute can be missing from a particular person instance.

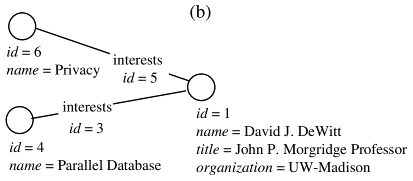
Figure 3.a shows for example the snapshot of a tiny G_d at time 1. On this snapshot the nodes are entities and the edges



(a)

```
+ person(id=1){name,title}
+ person(id=1).interests.topic{name}
– person(id=1).interests.topic(name=Statistics){name}
```

(b)



(c)

```
<# person(id=1){name}=David J. DeWitt #>
<# person(id=1){title}=John P. Morgridge
                                   Professor #>

<# person(id=1) {organization}=UW-Madison #>

<strong>Interests:</strong>
<# person(id=1).interests(id=3)
        .topic(id=4){name}=Parallel Database #>

<# person(id=1).interests(id=5)
               .topic(id=6){name}=Privacy #>
```

(d)

Fig. 3. (a) A snapshot of the ER graph G, (b) a sample view schema, (c) a sample data of the above view, and (d) how the above sample data is exported into a wiki page in the s-slot wiki language.

are relations (labeled with relation names). The attributes are described next to the nodes and edges.

Storing G using RDBMS: We want to query G efficiently and may want to implement a variety of concurrency control schemes later (to manage concurrent user edits), including lock-based schemes. Consequently, we decided to store G_s and G_d using an RDBMS. In particular, we extended the work [25] on building temporal RDBMSs, to support temporal data in a conventional RDBMS as follows.

First, we create several tables to store definitions of entity types, relationship types and their attributes. For instance, these tables specify that entity type person has attributes name, title, and organization, and that relationship type services relates person and conference, and so on.

After creating the above tables, we create a table Entity_ID($\underline{id}, etype$) to store all entity instances of all types: for each instance, we store only its ID and type. Similarly, we create a table Relationship_ID($\underline{id}, rtype, eid1, eid2$) to store all relation instances of all types: for each instance, we store its ID, its type, and the two entities with IDs $eid1$ and $eid2$ that it relates. Figure 4.a shows two such tables, which store the entity and relation instances of the ER graph G in Figure 3.a.

Next, for each attribute A of an entity or relationship type, we create two tables A_m and A_u to keep track of all values that have been entered for A by machine and human users, respectively.

Example 4.1: Figure 4.b-c shows two such tables for attribute organization*. The first tuple of table* Organization_m *for example specifies that attribute* organization *of entity (or relation) instance with ID 1 is assigned value "UW". This value is valid from time "2007-04-01 ..." to "9999-12-31 ...". Finally, the value was entered by machine with ID M. The first tuple of table* Organization_u *specifies that attribute* organization *of entity (or relation) instance with ID 2 is assigned value "Purdue". This value is valid from time "2007-05-02 ..." to time "9999-12-31 ...", and the value was entered by a human user U_1.* □

Thus, for these tables, field xid stores the ID of an entity or relation instance, value stores a value v entered for the attribute of this instance. Fields start and stop store two timestamps to indicate when v was first inserted, and when v was modified or deleted, respectively. When v is inserted, we set stop to '9999-12-31 23:59:59', the largest timestamp, to indicate that v is valid indefinitely. Then when v is modified or deleted at a later time t, we set stop to t, indicating that v is no longer valid (after that time). Finally, field who stores the ID of the user or machine who entered the value v.

Using the two tables A_m and A_u, we know all values entered for attribute A, by machine and human users, at all times. But at any particular time t, what should be the value of A? Intuitively, this value should be determined based on a *combination* of the values entered by machines and users up to time t.

Many possible combination policies can be specified. A simple one – which we call "human-first" – states that the value of A at any time t is the last value entered by a human user, up to time t, or the last value entered by machine, up to time t, provided that no human-entered value had existed so far. Section VI briefly describes the combination policy used in our current Madwiki implementation.

To obtain the combined values of an attribute A, we then create a table A_p, which is a view defined over tables A_m and A_u. The view definition encodes the combination policy in effect. The following example illustrates table A_p and the use of combination policies.

Example 4.2: Table Organization_p *in Figure 4.d shows the combined values for attribute* organization*, using the "human-first" combination policy described earlier, over tables* Organization_m *and* Organization_u *(Figure 4.b-c).*

Consider entity with ID 1, Table Organization_p *states that the organization of this entity is "UW" between 2007-04-01 and 2007-05-27, and is "UW-Madison" afterwards. Note that between 2007-04-01 and 2007-05-27, only machine M entered value "UW", so that value became the combined value. Then on 2007-05-27, a human user U_2 entered value "UW-Madison", and that value became the combined value, according to the "human-first" policy.*

Consider entity with ID 2. Note that a human user U_1 entered value "Purdue" for its organization on 2007-05-02, thus causing the combined value to be "Purdue". Afterward the machine entered value "MITRE" on 2007-05-20. However, this value is ignored, according to the "human-first" policy. Thus, the combined value remains "Purdue", as shown in Table Organization_p. □

Initializing G: To initialize G, we employ a machine-based solution M. Many such solutions exist [13]. Currently we use the Cimple solution which is described in detail in [13]. DBLife is an example portal built using the semi-automatic Cimple solution.

B. Creating Views over Database G

View Language Requirements: To create views over G, we must define a view language $\mathcal{L}$. We now discuss the requirements for $\mathcal{L}$. First, we note that a primary goal of community portals is to describe interesting entities and

Entity_ID

id	etype
1	person
4	topic
12	conf
···	···

Relationship_ID

id	rtype	eid1	eid2
3	interests	1	4
11	services	1	12
···	···	···	···

(a)

Organization_m

xid	value	start	stop	who
1	UW	2007-04-01 ···	9999-12-31 ···	M
2	MITRE	2007-05-20 ···	9999-12-31 ···	M

(b)

Organization_u

xid	value	start	stop	who
2	Purdue	2007-05-02 ···	9999-12-31 ···	U_1
1	UW-Madison	2007-05-27 ···	9999-12-31 ···	U_2

(c)

$\Rightarrow$

Organization_p

xid	value	start	stop	who
1	UW	2007-04-01 ···	2007-05-27 ···	M
2	Purdue	2007-05-02 ···	9999-12-31 ···	U_1
1	UW-Madison	2007-05-27 ···	9999-12-31 ···	U_2

(d)

Fig. 4. Example relational tables to store a Madwiki ER graph.

relations in the community. Toward this goal, we use each wiki page W to describe an entity e or a relation r. A popular way to describe an entity e, say, is to describe a "neighborhood" of e on the ER data graph G, e.g., all or most nodes within two hops from e. Consequently, language $\mathcal{L}$ must be such that we can easily write and modify views that describe such "neighborhoods".

Second, when a user requests a wiki page W, we materialize it on the fly, to ensure the page contain the latest updates. This in turn requires materializing the view V underlying W (see Section V-C). Consequently, $\mathcal{L}$ must be such that its views can be materialized quickly, to ensure real-time user interaction.

Finally, when a user U edits a wiki page W, we assume that U may also edit the schema of view V underlying W, e.g., by removing all papers from W, U may be modifying V's schema to exclude all papers (Section V-A discusses this assumption in depth). Hence, language $\mathcal{L}$ must be such that we can modify a view schema quickly, based on user edits, to ensure real-time user editing.

A Path-based View Language: The above requirements led us to design a path-based view language $\mathcal{L}_p$. To define $\mathcal{L}_p$, first we define *data* and *schema paths*. Intuitively, a *data path* is a path on the ER graph G that (a) starts with an entity node e_1 and ends at an entity node e_n, and (b) retains only certain attributes for each node/edge along the path.

A *schema path* $p = ep_1.rp_2.ep_3.\ldots.rp_{n-1}.ep_n$ then specifies a set of data paths, which start with node ep_1, follow edge rp_2, etc., then end with node ep_n. To further constrain these data paths, we express each ep_i as $T_i(C_i)\{A_i\}$, meaning that (a) ep_i must have type T_i and satisfy condition C_i (which is a conjunction of conditions over the attributes), and (b) we keep only those attributes of ep_i that appear in A_i (which is a set of attribute names). T_i is required, but (C_i) and $\{A_i\}$ are optional. A missing $\{A_i\}$ means that we retain all attributes. We express each rp_i in an analogous fashion.

Example 4.3: The schema path $person(id = 1)\{name, title\}$ specifies a single data path that corresponds to person entity with id=1 and that contains only attributes name and title of this entity. The schema path, person(id=1).give-tutorial.conf{name}, specifies a set of data paths, each of which starts with a person node whose id is 1, follows an edge give-tutorial, then ends with a conf node. For each path, we retain all attributes of person node and give-tutorial edge, but retain only the name attribute of conf node. □

We can now define ER views considered in this paper as follows:

Definition 1 (Path-based ER views): A path-based ER view (or view for short when there is no ambiguity) V has a schema $V_s = (In, Ex)$, where In and Ex are disjoint sets of schema paths over G. Evaluating V_s over G yields the view data V_d. V_d is a subgraph of G that contains only data paths that are (a) specified by any path schema in In and (b) not specified by some path schema in Ex. We refer to schema paths in In and Ex as inclusive and exclusive paths, respectively.

Example 4.4: Figure 3.b shows a sample V_s that has two inclusive paths and one exclusive path. This view schema selects a person e with id $= 1$, retains name and title of e, then selects all interests of e except those named "Statistics". Evaluating this view schema over the ER graph G of Figure 3.a produces the view data V_d in Figure 3.c. □

We now discuss how language $\mathcal{L}_p$ satisfies the requirements outlined earlier. First, most "neighborhoods" of an entity e (e.g., all nodes within two hops of e on ER graph G) can be expressed with a set of inclusive and exclusive data paths. Hence, $\mathcal{L}_p$ allows us to quickly write views that capture such neighborhoods, in an intuitive manner. Second, evaluating schema paths amounts to performing selection operations over the ER graph G. Hence, views in $\mathcal{L}_p$ can be materialized quickly. Finally, if a user edits a view schema (using a wiki page), then such edits can be quickly mapped into a set of inclusive and exclusive schema paths, allowing us to modify the view schema quickly and easily (see [24] for an in-depth discussion).

Creating Views over ER Graph G: Now that we have defined the view language $\mathcal{L}_p$, we can discuss how Madwiki uses $\mathcal{L}_p$ to create views over G. First, Madwiki decides on the set of entities and relations to be "wikified". Currently, for simplicity, we "wikify" all entities, but no relations. Next, for each entity type E to be "wikified" (e.g., person), Madwiki specifies a *default* view schema V_E that specifies a "neighborhood" of instances of E. These view schemas are application specific. Finally, for each instance e of type E, Madwiki creates a view schema V_s by initializing it with V_E. Thus, each entity instance has its own view schema. The data of the views is not stored, but will be materialized on the fly when creating and refreshing wiki pages, which we discuss next.

C. Converting Views to Wiki Pages

For each entity instance e, let V be its view – defined by a schema V_s – over the structured database G, as defined earlier. Let V_d be the data of view V, obtained by materializing V_s

over G. Data V_d is an ER graph (which is a subgraph of the larger ER graph G). We now discuss how to convert V_d into a wiki page W.

Since most current wiki data (e.g., Wikipedia) is natural text, the straightforward solution is to convert V_d into a set of natural-language sentences. For example, suppose V_d specifies that person X works for organization Y. Then we can convert this into sentence "X works for Y" in wiki page W. Knowing this template, if a user later modifies the sentence to be "X works for Y'", we can still parse it back, realize that Y has been modified to be Y', then update the underlying database G accordingly.

This was indeed the first solution we tried. It is very easy for users to edit natural-language wiki pages generated by this solution. But after extensive experiments, we found that it is difficult to extract and update structured data. The set of templates that we can use in natural language settings are somewhat limited; hence, they get reused in multiple contexts, causing many ambiguities for the extractor. Furthermore, suppose G has been updated (e.g., by machine) so that X is now working for Z. To update page W with this information, we must be able to pinpoint the location of Y. This is equivalent to being able to extract Y, a difficult task, as discussed earlier.

For these reasons, we wanted a solution where *it is trivial to pinpoint pieces of structured data* contributed by V_d. A wiki page then contains multiple "islands" of structured data from V_d, in a "sea" of natural text contributed by users. We refer to these "islands" as *s-slots* (shorthand for *structured slot*). Below we describe this *s-slot solution*. In Section VII we discuss how the natural-language and s-slot solutions lie at two ends of a spectrum of solutions that trade off (a) ease of user edit, (b) ease of extracting and updating structured data, and (c) ease of moving data around on wiki pages.

The S-Slot Solution: We first define the notion of attribute path. Recall that a schema path p has the form $T_1(C_1)\{A_1\}.\dots.T_n(C_n)\{A_n\}$. We say that p is an *attribute path* iff (1) A_1-A_{n-1} are empty sets and A_n identifies a single attribute a, and (2) p evaluates to a single path instance. Thus, p uniquely identifies attribute a. Examples of attribute paths are $person(id = 1)\{title\}$ and

$$person(id = 1).\text{write-pub}(id = 5).pub(id = 14)\{name\}.$$

An s-slot s then has the form $<\# \; p = v \; \#>$, which specifies that the attribute a uniquely identified by the attribute path p takes value v. An example of wiki text including an s-slot is

```
<# person(id=1){name}=David DeWitt #> works for
<# person(id=1).work-org.org(id=13){name}=UW #>
since 1976.
```

The HTML presentation of this wiki text will display "David DeWitt works for UW since 1976".

Now let V be a view with schema V_s that Madwiki has defined over database G (see Section IV-B). Then Madwiki generates the default wiki page W for V in two steps: (a) evaluating V_s over G to obtain the view data V_d, which is a subgraph of the ER graph G, and (b) converting V_d into a wiki page W using s-slots interleaved with English text.

Step (a) is relatively straightforward. Step (b) can be executed in many different ways. We currently adopt a default solution. Suppose we know that view V (and thus wiki page W) describes entity e, e.g., David DeWitt. Then our default solution first generates the line $< \#person(id = 1)\{name\} = David\ DeWitt\ \#>$ as the title of the wiki page. Next, it displays the attributes of e, then the relationships. Figure 3.d shows how the data graph V_d in Figure 3.c may have been displayed in a wiki page (see [24] for the algorithm description).

The set of all wiki pages generated as above constitutes the initial community portal $\mathcal{W}$. The next section discusses how users can contribute to this portal.

V. Managing User Contributions

In this section we discuss what users can edit and how to process those edits.

A. What Can Users Edit?

Consider a user U editing a wiki page W. We allow U to edit both text and structured data of W. Editing text is trivial. Editing structured data of W means U can modify or delete s-slots, or insert new ones. Currently, U modifies s-slots manually, though Madwiki will eventually provide helpful tools such as form-based GUI interfaces for editing.

In modifying an s-slot $s = <\#p = v\#>$, U can modify the attribute path p as well as value v, but is not allowed to modify the formatting characters (e.g., $<\#$, $=$, and $\#>$). If U were to do so, then the parser would fail to recognize the s-slot, and hence would interpret the modified s-slot as text, not structured data.

Let V be the underlying view of W. Conceptually, editing structured data of W means editing one or a combination of the following components: the data of V, the schema of V, the data of G, and the schema of G (denoted V_d, V_s, G_d, G_s, respectively).

In traditional settings such as RDBMS, ordinary users can only edit view data and thus also the underlying relational database data. This maps to editing V_d and G_d in our case. Should we also allow users to edit V_s and G_s? We decided to allow these actions, because there is often a natural need to do so. For example, a user U may naturally want to modify W so that it no longer *displays* emails. To do this, U must modify V_s. U cannot modify V_d because this would mean *removing* certain emails from G, not the desired effect. As another example, user U may naturally want to add to an entity e (described in W) a new attribute a that has not existed so far in the portal. To do this, U must modify both G_s and V_s.

The next question then is: what is the best way to allow users to modify V_s and G_s? A possible option is to expose these schemas in wiki pages, for users to edit. For example, we can expose V_s in a wiki page W_s. Then when U edits W, we interpret such edits as editing V_d, and when U edits W_s, we interpret such edits as editing V_s.

The above option would greatly reduce the ambiguity in interpreting user edits. However, we decided against it, because we found from experimentation that it is difficult for *ordinary,*

Basic ER Actions	V_d	V_s	G_d	G_s
a_1: Modify attribute value	✓		✓	
a_2: Insert an existing attribute	✓	✓	opt.	
a_3: Insert a new attribute	✓	✓	✓	✓
a_4: Insert an existing entity	✓	✓	opt.	
a_5: Insert a new entity	✓	✓	✓	✓
a_6: Insert an existing relationship	✓	✓	opt.	
a_7: Insert a new relationship	✓	✓	✓	✓
a_8: Delete an attribute	✓	✓	opt.	opt.
a_9: Delete an entity	✓	✓	opt.	opt.
a_{10}: Delete a relationship	✓	✓	opt.	opt.

Fig. 5. Basic ER actions that we have defined.

database-illiterate users to remember this option. In fact, users often are not even aware of the distinction between data and schema edits. Instead, they appear to prefer to edit only the wiki page W, then rely on Madwiki to assist them in executing the right kind of edit actions.

For these reasons, we allow U to edit only wiki page W, then ask U (in English) to clarify if he or she intends to edit the data or the schema. In what follows we discuss this process in detail.

B. Infer & Execute Structured Edits

Suppose user U has edited wiki page W into W'. Then we can parse W' to extract a text portion T' and a structured data portion D'. The text portion can immediately be stored in a text database T (see Figure 2). The structured data portion D' consists of all s-slots in W.

Next, we can merge all s-slots in D' together to obtain an ER graph that we will refer to as V'_d. Given that each s-slot maps uniquely into an attribute in the ER graph G, the merging process is relatively straightforward, and hence will not be discussed further, for lack of space. Our problem now is: given V'_d, infer what actions user U intends to execute on V_d, V_s, G_d, G_s, then execute those actions.

Basic Relational and ER Actions: To solve the above problem, we first define a set of basic actions that U can execute over V_d, V_s, G_d, G_s. For example, basic actions on V_d include modifying the value of an entity or relation attribute, and deleting an entity. Basic actions on V_s include inserting a new entity and deleting an attribute of a relationship. We have implemented each basic action as a program over the temporal relational database that stores G. The full paper [24] describes the complete sets of basic actions (there are 10, 8, 10, and 8 such actions for V_d, V_s, G_d, and G_s, respectively) as well as their implementations. Abusing notation, we will refer to these basic actions as *basic relational actions*, to distinguish them from the basic ER actions that we will introduce soon below.

Now given V'_d, we must infer the sequence of basic relational actions that we believe user U intends to execute. To do this in a manageable fashion, we introduce an intermediate user interface: the *ER interface*. This interface would display an ER data graph (e.g., V_d) in a graphical fashion, and allow users to execute a number of *basic ER actions*, such as modifying a node or an edge, deleting a node, etc.

The first column of Figure 5 lists the ten basic ER actions we have defined. We have implemented each ER action as a

```
Input:   Data graphs $V_d$ and $V'_d$. $V_d$=($E$, $R$, $A$), $V'_d$=($E'$, $R'$, $A'$),
         where $E$, $E'$ are sets of entity instances, $R$, $R'$ are sets of
         relationship instances, and $A$, $A'$ are sets of attributes.
Output:  Sequence of GUI actions $\mathcal{S}_{ER}$.
 1. FOR each entity instance $e \in E' - E$ DO
 2.     IF entity type exists THEN append $a_4$ to $\mathcal{S}_{ER}$;
 3.     ELSE append $a_5$ to $\mathcal{S}_{ER}$;
 4. FOR each relationship instance $r \in R' - R$ DO
 5.     IF relationship type exists THEN append $a_6$ to $\mathcal{S}_{ER}$;
 6.     ELSE append $a_7$ to $\mathcal{S}_{ER}$;
 7. FOR each attribute $a \in A' - A$ DO
 8.     IF attribute type exists THEN append $a_2$ to $\mathcal{S}_{ER}$;
 9.     ELSE append $a_3$ to $\mathcal{S}_{ER}$;
10. FOR each attribute $a \in A - A'$ DO
11.     append $a_8$ to $\mathcal{S}_{ER}$;
12. FOR each relationship instance $r \in R - R'$ DO
13.     append $a_{10}$ to $\mathcal{S}_{ER}$;
14. FOR each entity instance $e \in E - E'$ DO
15.     append $a_9$ to $\mathcal{S}_{ER}$;
16. FOR each attribute $a \in A \cap A'$ DO
17.     IF it has the same value in $V_d$ and $V'_d$ THEN append $a_1$ to $\mathcal{S}_{ER}$;
18. Return $\mathcal{S}_{ER}$;
```

Fig. 6. Generating $\mathcal{S}_{ER}$ from V_d and V'_d.

sequence of relational actions. For example, action a_1 (see the table) translates into the sole relational action that modifies the value of an entity attribute (in both V_d and G_d).

However, it turns out that an ER action can be *ambiguous*, in that it can map into different sequences of relational actions, depending on the user intention, as the following example illustrates:

Example 5.1: Suppose a user U applies action a_8 (see Figure 5) to delete an attribute x of, say, a person entity e in an ER graph, e.g., V_d. Then U may mean to delete x from (a) V_s, i.e., do not display x in view V, or (b) G_d, thus declaring that entity e does not have attribute x, or (c) G_s, thus declaring that attribute x does not exist for person (the entity type of e). □

Since we do not know U's intention, if U executes action a_8, then we first ask U (in an English phrase) to choose among options (a)-(c) in the above example. Next, we translate a_8 into the appropriate sequence of relational actions, depending on U's answer. For example, if U chooses option (c), then the sequence of relational actions is: delete x from V_s, delete x from G_d, delete x from G_s.

For each ER action, Columns 2-5 of Figure 5 shows which components (V_d, V_s, etc.) that the action may modify ("opt." means "optional", depending on external conditions such as user intentions).

Mapping User Edits into Sequence of Basic Actions: With the introduction of the ER interface, our problem can be recast as follows. When user U edits the structured data portion of wiki page W, we view it to be equivalent to U editing the ER graph V_d in the ER interface, using basic ER actions. We do not know what basic ER actions U executes. But we do know the end result, which is the ER graph V'_d, as described earlier.

Thus, in this perspective, U has executed a sequence $\mathcal{S}_{ER}$ of basic ER actions on the original ER graph V_d, transforming it into a new ER graph V'_d. Our task then is to "reverse engineer" $\mathcal{S}_{ER}$, by comparing V_d with V'_d, then execute the resulting $\mathcal{S}_{ER}$. Figure 6 shows the pseudo code of our current algorithm to reverse engineer $\mathcal{S}_{ER}$.

To "push" the structured edits of U into the database G, we then execute the actions of $\mathcal{S}_{ER}$ sequentially. Recall that each such action is a basic ER action (see Figure 5), which can be ambiguous. If this happens, recall also that we resolve the problem by asking user U a disambiguating question. We then execute each basic ER action by executing the sequence of relational actions that it maps to, as described earlier.

A minor problem is that $\mathcal{S}_{ER}$ is not unique. Given any two V_d and V'_d, multiple sequences of actions $\mathcal{S}_{ER}$ may exist that all transform V_d into V'_d. Fortunately they all have the same effect, as this theorem shows:

Theorem 1: Let $\mathcal{S}_1, \ldots, \mathcal{S}_k$ be all sequences of basic ER actions that transform a V_d into a V'_d. Then when executing any $\mathcal{S}_i$, the set of questions we pose to user U will be the same for all i. If U gives the same answers to these questions, then executing any $\mathcal{S}_i$, $i \in [1, k]$, results in the same V_d, V_s, G_d and G_s.

Intuitively, each $\mathcal{S}_i$ is a sequence of insert, delete and update actions that transform ER graph V_d into V'_d. Among these actions, only deletions are ambiguous since they can be interpreted in multiple ways (see Example 5.1). Thus as long as user U disambiguates these deletions in the same way, any two sequences $\mathcal{S}_i$ and $\mathcal{S}_j$ will result in the same V_d, V_s, G_d and G_s.

C. Propagate Structured Edits

Let W_1 and W_2 be two wiki pages that describe two researchers A and B, respectively. Suppose A and B share one publication p. So p appears in both W_1 and W_2. Now suppose that a user U has edited p in W_1. When should we update p in W_2? In general, once a user has edited the structured data portion of a wiki page W, how should we propagate this edit to other pages?

A solution is to immediately refresh other pages, e.g., page W_2 in the above example. We call this *eager propagation*. This solution ensures timely updates of pages, but can raise tricky concurrency control issues. Hence, we currently adopt a *lazy propagation* approach, where we refresh a page, say W_2, only when a user requests the page again. At that moment, we rematerialize the page from the structured database G and the text database T. Section VII empirically shows that we can refresh pages on the fly quickly, in a few seconds, thus making this lazy approach a practical solution.

VI. Managing Multiple Users and Machine

While not a contribution of this paper, for completeness we will briefly touch on the key problems of managing multiple users and machines as they contribute to the portal. The full paper [24] discusses these problems and our proposed solutions in detail.

First, we must manage concurrent editing of a wiki page by multiple users, or concurrent editing of some structured data pieces (e.g., a paper) that appear in multiple wiki pages. Currently we employ the optimistic concurrency control scheme of Wikipedia for this purpose.

Next, we must detect and remove malicious users. To do this, we currently employ a hierarchy of users, reminiscent to the Wikipedia solution for the same problem. Specifically, we require that users log in to edit, and employ a set of editors whose job is to monitor most active wiki pages.

Finally, if a user U has modified a data item X, can machine M overwrite U's modification, and if so, then when? Our current solution allows M to overwrite U's data only for certain pre-specified data types (e.g., certain attributes of person), if M is sufficiently confident in its data. For all other data types, we do not allow M to overwrite U's modification, but allow it to add a suggestion next to U's modifications, in parentheses, e.g., "age is 45 (according to M, age is 47)".

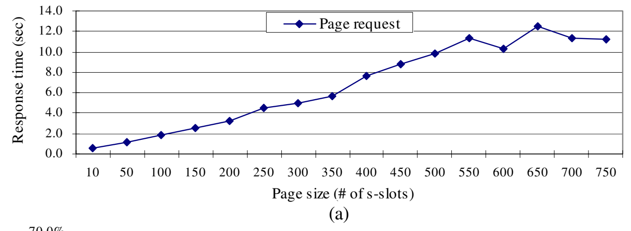
VII. Empirical Evaluation

To evaluate Madwiki, we have been applying it to build a community wikipedia for the database community (see [8] for the current portal, still under continuous development). We now report on preliminary experiments with this portal, which demonstrate the potentials of Madwiki and suggest research opportunities.

Building an Initial Community Portal: We began by employing DBLife as machine M (see Section IV). It took a two-person team four weeks to develop DBLife from scratch. DBLife was first deployed on May of 2005, and has been on "auto pilot" since, requiring only about one hour of maintenance per month (for more details, see [13]). Each day DBLife crawls 10,000+ database research related data sources, extracts and integrates the data, to generate a daily ER data graph.

We used one such daily ER data graph A (98M of XML data) to initialize the structured database G. G's schema has five entity types and nine relation types, and G's data contains 164,043 entity instances and 558,260 relation instances, for a total size of 413M. This size is greater than the ER data graph size of 98M due to the extra space needed to store temporal information. It took 216 seconds to load A into G, and 183 minutes to generate and store all wiki pages (164,043 pages for entities). These results suggest that we can create moderate-size initial portals (a one-time task) with relatively little effort.

Next, we wanted to know if the initial portal can be maintained efficiently, assuming no user contributions yet. We found that over 10 days, as DBLife contributed data to the structured database G, G's size increased from 413M to 600M. This was somewhat surprising, because DBLife data should not have changed so much over 10 days. Upon a closer inspection, we found that the confidence scores of most relation instances in G (e.g., person X is related to person Y with score .8) were changed by DBLife everyday, due to the changing raw data (retrieved by DBLife). Hence, the confidence scores of most relation instances in G were updated everyday, leading to a rapid growth in G's size (recall that G is a temporal database that does not allow update in place, hence changes are added to G). Once we disallowed updating confidence scores, then G grew very slowly (by less than 5M). Thus, this experiment suggests that the current design of G is efficient for maintaining all aspects of the initial portal over time, except for confidence/uncertainty scores. We are currently examining how to modify the temporal design of



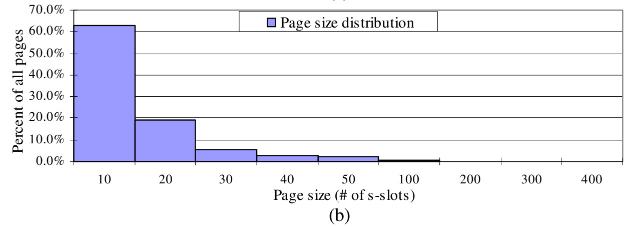


Fig. 7. Time to request a wiki page and distribution of page size.

# of s-slots = 52 time in sec					
Type of edits	**5 edits**	**10 edits**	**15 edits**	**20 edits**	**25 edits**
modification	0.258	0.266	0.275	0.283	0.291
insertion	1.041	1.314	1.583	1.826	2.115
deletion	1.012	1.122	1.253	1.363	1.483

# of s-slots = 196 time in sec					
Type of edits	**5 edits**	**10 edits**	**15 edits**	**20 edits**	**25 edits**
modification	1.183	1.209	1.231	1.247	1.266
insertion	1.971	2.214	2.436	2.662	2.855
deletion	1.615	1.633	1.649	1.665	1.681

Fig. 8. Time to process user edits on a wiki page.

G to efficiently accommodate frequent changes in uncertainty scores.

Expressive Power of the S-Slot Wiki Language: In the current DBLife system (see $dblife.cs.wisc.edu$) each user superhomepage is a structured view V over the underlying structured database. We found that the s-slot wiki language (Section IV-C) was sufficiently powerful to enable us to express all structured data pieces in such views in wiki pages, except two types of data pieces: top-k and aggregate. A top-k data piece is technically a view that lists the top k items of a ranked list, e.g., the top three authors, cited papers, etc. An aggregate data piece is an aggregate view such as the total number of papers per author, or the total number of citations.

We found that top-k and aggregate views also appear in many other community portals. Thus, any future attempt to extend wiki languages with structured constructs must address the problem of expressing such views. The challenge then is how to efficiently update such views.

Efficiencies of User Interaction: In the next step, we examined how fast users can interact with the portal. Figure 7.a shows the time it takes from when a user requests a page W until when W is served. Note that to ensure freshness, we materialize W on the fly, from the underlying structured database G and text database T (Section V-C). Hence, it is critical that such materialization can be done quickly, to ensure real-time user interaction.

The results show that request time increases linearly w.r.t. page size, measured in the number of s-slots in the page, and stays small, e.g., under 2 seconds for page sizes up to 150.

Editing tasks	**Time (sec)**	**Accuracy**
editing a sentence of free text	13.2 (10~21)	100%
modifying a data path	16.4 (10~30)	100%
inserting a data path	52 (30~60)	100%
inserting two bonded data paths	55 (30~85)	100%
inserting a paragraph of data paths	152 (60~240)	100%
deleting data paths	36.6 (15~60)	100%

Fig. 9. User performance on several editing tasks.

Figure 7.b shows that the vast majority of current pages have a size under 50 (the first five bars of the figure), and thus incur under 1 second request time. This result suggests that we can materialize wiki pages quickly, and that the lazy update approach (Section V-C) can work well in practice.

Since processing user edits requires us to translate these edits across different user interfaces and then to invoke the underlying relational database, we wanted to know if it can be done efficiently. Figure 8 shows the time it takes from when a user submits his/her edits until when the edits have been processed, i.e., updates on V_s, V_d, G_s, G_d, if any, have been carried out. This time does not include the time users spent answering disambiguating questions (Section V-B). The top table of the figure shows edit times over a wiki page with 52 s-slots (each time is averaged over 10 runs). Here each edit is a user action that affects a single s-slot.

The bottom table of the figure shows similar edit times, but over a wiki page with 196 s-slots. In both cases, the results show that the edit times remain small, under 2.2 seconds for the small wiki page and 2.9 seconds for the large wiki page. This suggests that Madwiki can process user edits efficiently.

Ease of User Interaction: Next, we evaluated how easy it is for users to edit structured data in a wiki page W. We conducted a preliminary experiment with six users (graduate students in this case), where each user was asked to edit a certain item on the HTML representation of W. To do so, they had to go to W, locate and then edit the appropriate piece of structured data. We measured how long it took to finish the given tasks and the correctness of the results. For comparison purposes, we also asked users to edit some free text.

Figure 9 shows that 100% correctness was achieved for all the editing tasks. Editing time is measured from when the edit button is clicked until when the new HTML page is rendered. Figure 9 shows the average and range of recorded editing time over all the users. The results show that the simplest structured data editing task, modifying a data path (modifying an attribute), took comparable time to editing a sentence of free text.

Inserting a data path generally involves adding several entities and relationships to the database. Users need to type a complete legal path. Inserting two bonded data paths is a bit more complex since users need to make sure that several entities (or relationships) are assigned the same id. Inserting a paragraph of data paths is probably the most complex task that generally involves multiple bonded data paths. Specifically, the users were asked to add a publication with a title, an ordered author list, and the conference, year, page, and citation information. The results show that the editing time is very reasonable considering the high complexity of the task.

Deletion of data paths would generate some ambiguities since the user may mean to delete the structured data from the underlying database or just from the wiki page. Thus after the user clicks the submit button, several questions may be presented as radio buttons to clear the possible ambiguities. Deleting a single data path or many data paths would take similar amount of time from the editing point of view. The only difference is the number of questions to ask.

This experiment is strictly preliminary. But it does suggest that the current solutions may already be adequate in the sense that users are able to correctly execute the various editing tasks within a reasonable amount of time.

The experiment also suggests that it may be even easier for users to edit if we introduce some macros that hide the details of the structured data and make the structured data looks very clean. This point was confirmed by the users' qualitative feedback on how convenient it is to use the system. On a scale of 1 (least convenient) to 5 (most convenient), the current system scored an average of 2.5. A typical comment is that while the system is easy to learn and functioning well, it is verbose. These comments meet our expectations since our goal for the current version focuses almost exclusively on the adequacy instead of convenience.

In general, as commented in Section IV-C, a lesson we learned from our current Madwiki experience is that there is a spectrum of solutions on how structured data can be represented in wiki pages. Our s-slot solution represents one end of the spectrum and the natural-language solution (see Section IV-C) the other. In between we can have solutions that present structured data using, e.g., XML formats (in wiki pages).

The key tradeoff factors for these solutions include (a) how easy it is for users to edit, (b) how easy it is for machines to re-extract structured data, and (c) how easy it is for users to move various pieces of structured data around, i.e., rearrange them in the wiki page.

The s-slot solution appears best for (b) and (c), and mediocre for (a). The natural-language solution is best for (a), mediocre for (c), and difficult for (b). An XML-like solution appears best for (a) and (b), but mediocre for (c). Developing more solutions, evaluating them, and selecting a good one is an interesting future research direction.

VIII. Conclusions & Future Work

We have described Madwiki, an approach that employs both "machines" and human users to build structured community portals. This new hybrid machine-human approach can bring significant benefits. It can achieve broader and deeper coverage, provide more incentives for users to contribute, and keep the portal more up-to-date with less user effort. We have applied Madwiki to build a "wikipedia" portal for the database community [8]. We reported on our experience with this portal that demonstrates the potentials of Madwiki and suggests many research opportunities.

Indeed, it is clear that our work here has only scratched the surface of this direction (of combining "machines" and human to build structured wikipedias). Many interesting research problems arise from the points where we used simple initial solutions in our prototype. Example problems include: (a) extending the s-slot wiki language to handle top-k and aggregate views and studying updating for such views, (b) developing "macros" that hide the low-level structured constructs to allow users to edit certain structured data pieces more efficiently, (c) developing efficient eager-update-propagation schemes, (d) developing better solutions to handle machine updates to data already modified by users, and (e) learning how to leverage user edits to improve the extraction and integration accuracy of machines.

Acknowledgment

This work is partially supported by NSF CAREER grant IIS-0347903, an Alfred Sloan fellowship, and a gift grant from Yahoo! Inc.

References

[1] A. Doan, R. Ramakrishnan, F. Chen, P. DeRose, Y. Lee, R. McCann, M. Sayyadian, and W. Shen, "Community information management," *IEEE Data Engineering Bulletin, Special Issue on Probabilistic Databases*, vol. 29, no. 1, 2006.
[2] Z. Nie, J. Wen, and W. Ma, "Object-level vertical search," in *CIDR-07*.
[3] "http://rexa.info/."
[4] C. Giles, K. Bollacker, and S. Lawrence, "Citeseer: An automatic citation indexing system," in *DL-98*.
[5] N. Bansal and N. Koudas, "Blogscope: Spatio-temporal analysis of the blogosphere," in *WWW-07*.
[6] "http://oak.cs.ucla.edu/blogocenter."
[7] "http://en.wikipedia.org/."
[8] "http://dblife-labs.cs.wisc.edu/wiki-test/index.php/main_page."
[9] M. Volkel, M. Krotzsch, D. Vrandecic, H. Haller, and R. Studer, "Semantic wikipedia," in *WWW-06*.
[10] "http://en.wikipedia.org/wiki/semantic_wiki."
[11] D. Frankowski, S. K. Lam, S. Sen, F. M. Harper, S. Yilek, M. Cassano, and J. Riedl, "Recommenders everywhere: The wikilens community-maintained recommender system."
[12] "http://metaweb.com/."
[13] P. DeRose, W. Shen, F. Chen, A. Doan, and R. Ramakrishnan, "Building structured web community portals: The case for an incremental and compositional approach," in *VLDB-07*.
[14] F. Bancilhon and N. Spyratos, "Update semantics of relational views," *ACM Transactions on Database Systems*, vol. 6, pp. 557–575, 1981.
[15] U. Dayal and P. A. Bernstein, "On the correct translation of update operations on relational views," *ACM Transactions on Database Systems*, vol. 7, no. 3, pp. 381–416, 1982.
[16] I. Androutsopoulos, G. D. Ritchie, and P. Thanisch, "Natural language interfaces to databases–an introduction," *Journal of Language Engineering*, vol. 1, no. 1, pp. 29–81, 1995.
[17] Y. Li, H. Yang, and H. V. Jagadish, "Constructing a generic natural language interface for an xml database," in *EDBT-06*.
[18] S. Amer-Yahia, "A database solution to search 2.0," *WebDB-07*.
[19] R. McCann, W. Shen, and A. Doan, "Web 2.0 style schema matching," in *ICDE-08*.
[20] F. Wang, C. Rabsch, P. Kling, P. Liu, and P. John, "Web-based collaborative information integration for scientific research," in *ICDE-07*.
[21] R. Ramakrishnan, "Community systems: The world online," in *CIDR-07*.
[22] 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, Special Issue on Data Management in Social Networks*, 2007.
[23] "Sixth international workshop on information integration on the web," 2007.
[24] P. DeRose, X. Chai, B. J. Gao, W. Shen, A. Doan, P. Bohannon, and J. Zhu, "Building community wikipedias: A machine-human partnership approach," *http://pages.cs.wisc.edu/∼xchai/madwiki.pdf*, 2007.
[25] R. T. Snodgrass, *Developing Time-Oriented Database Applications in SQL.* Morgan Kaufmann Publishers, Inc., 1999.