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], Jerry 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., Cimple, Libra, Citeseer) 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 efforts. In a sense, it enables building "community wikipedias", backed up 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.

1 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.info, BlogScope, and Blogosphere [16, 21, 6, 17, 11, 5].

The above approach incurs relatively little human efforts, often generates a reasonable initial portal, keeps portals fresh with automatic updates, and enables structured queries 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 efforts to keep portals up to date, and cannot accommodate structured queries, because users contribute mostly text and images.

In this paper we consider combining the above two complimentary 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 Cwiki. The following example illustrates the approach.

Example 1.1. *Suppose we apply* Cwiki *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 been evolved over time, with both machines and users' contributing

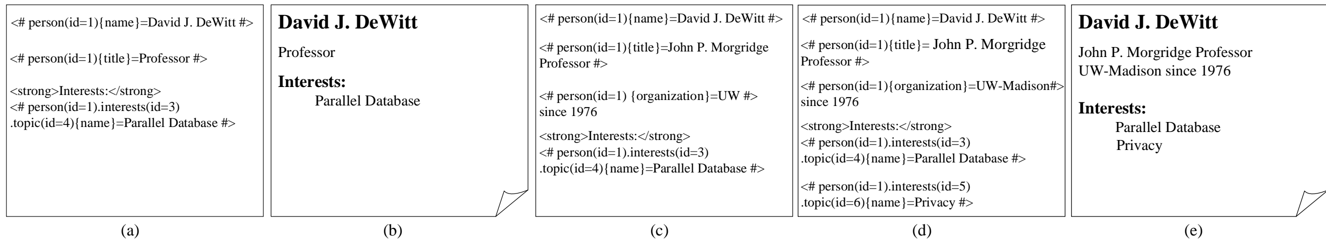


Figure 1. An example to illustrate the machine-human approach.

and correcting each other's contributions.

As described, this new hybrid machine-human approach enables building "community wikipedias" that are backed up 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 improving it. Third, it can keep the portal more up to date, with less user efforts, 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 in an underlying structured database, thus enabling a variety of structured queries 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 [2] 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 4.3).

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 5 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 6 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 research community (see the live system at [1]). In Section 7 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 up 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 structure 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.

2 Related Work

We are not aware of any published work that has studied combining "machines" and human 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 [24]. 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 4.3). But our constructs are far more powerful: we can embed arbitrary ER data graphs in a wiki page, whereas the constructs in [24] in a sense only allow embedding node and relation *attributes*. More importantly, Semantic Wikipedia and several similar efforts, including semantic wikis [3] and Metaweb [4], 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 [14] for an extensive discussion). Any of these can be employed as “machines” in our current work.

Processing structured user edits in our context is a variation on the classical view update problem [10, 12]. 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 5.2). Such UI translations have been studied, e.g., translating a natural-language user query into a structured one [9, 19]. Translating free natural-language queries is well known to be difficult [9, 19]. 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 [8, 20, 25, 22, 15, 7]. Our work here contributes to this emerging direction.

3 The Cwiki Approach

In the rest of the paper we describe the Cwiki approach. Figure 2 illustrates how Cwiki 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

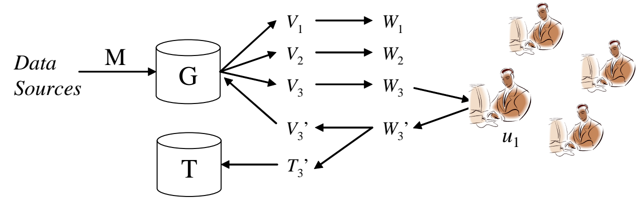


Figure 2. The Cwiki architecture

database G. Next, it initializes an empty text database T, which will be used in the future to store text generated by users. Then Cwiki 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 Cwiki extracts the structured data portion V_3' from W_3' and uses it to update the structured database G. Next, Cwiki extracts the text portion T_3' from W_3' and stores it in the text database T. Cwiki 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 machines can add new pages or delete existing ones.

The next two sections describe the key contributions of this paper: how to build the initial portal and to manage user contributions. Section 6 briefly touches upon the issue of managing multiple users and machine.

4 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.

4.1 Creating a Structured Database G

Here we describe in detail the language we use to model database G, how we extend a conventional RDBMS to capture temporal aspect of G, and how we initialize G using the Cimple solution [14].

4.1.1 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 4.3 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

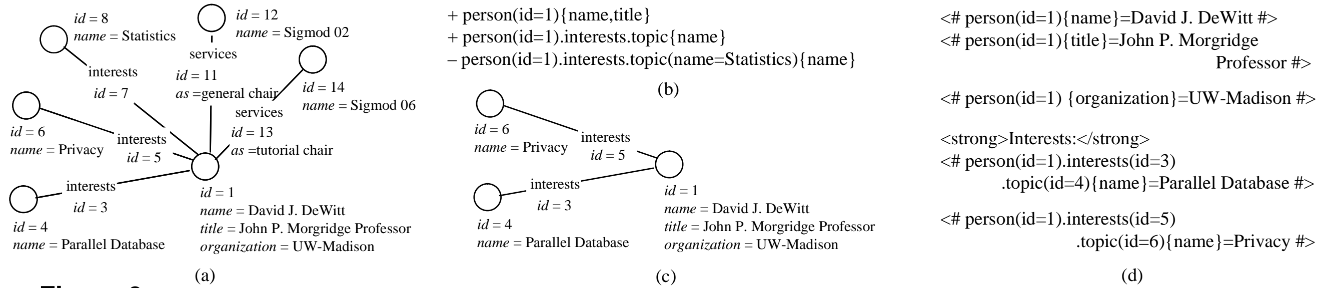


Figure 3. (a) A snapshot of the ER graph G, (b) a sample view schema, (c) a sample data of the above view, and (d) how the above sample data is exported into a wiki page in the s-slot wiki language.

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

4.1.2 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. The key questions are then: (1) How to convert G_s, essentially an ER graph, into a set of relational tables? (2) How to extend a conventional RDBMS to store temporal data? (3) How to manage data from multiple users and machine? In what follows, we first elaborate on and propose an initial solution to each question. Then we present a complete solution to storing G using an RDBMS.

Converting G_s into Relational Tables: As described above, schema G_s consists of a set of entity types and relation types. A standard approach to translating G_s into a relational database schema is to convert each entity (or relation) type into a table. And each attribute of the entity (or relation) type becomes an attribute of the table. An example is shown in Figure 4.a. Table *person* store person entity instances. In the table, column *id* gives the ID of a person entity, and *name*, *title* and *organization* are the three attributes describing each person. Such a design, however, is not space-optimized in our scenario for the following reasons:

- A table may be sparse. Take a person table for example. Most title values may be missing. This happens when title values are obtained from data sources, but the extractors are not powerful enough to extract them, or many title values are simply not available in the sources.
- Users may create new attributes for an entity type (e.g., creating attributes *homepage* and *country* for *person*). In this case, we need to enlarge the schema of the entity table, and entries for these new attributes are empty.
- Last but not the least, as we will see later, space utilization gets worse when we extend an RDBMS to store temporal data. When an attribute is updated, instead of updating the value in place, we *logically* delete the record with the old value and insert a new record with the new value. Other attribute values in the old record are copied to the new record. Consequently, we waste space in duplicating other attributes. Waste is significant when the table is wide (i.e., contains many attributes) and updates are frequent.

To address these problems, we chose to vertically partition an entity or relation table along each attribute. Consider an entity type E. Let $A_1, \ldots, A_n$ be the set of attributes E has. We convert E into n attribute tables. Each attribute table T_i ($1 \leq i \leq n$) is defined as $T_i(\underline{id}, value)$, where id stores the ID of an entity e, and $value$ stores the A_i value of e. In table T_i, we only store those entities that have an A_i value. A partitioning of the person table in Figure 4.a is given in Figure 4.b-d. Note that table *person_title* has only one record since title values for the other two persons are missing. Similarly, we can convert a relation type into a set of attribute tables. For a relation type, in addition to the attribute tables, we need one more table to store the IDs of the entities that each relation relates, as we will see later.

person

id	name	title	organization
1	David J. DeWitt	Professor	UW-Madison
2	Mike Brown	NULL	NULL
3	Chris Clifton	NULL	Purdue

(a)

person_name

id	value
1	David J. DeWitt
2	Mike Brown
3	Chris Clifton

(b)

person_title

id	value
1	Professor

(c)

person_organization

id	value
1	UW-Madison
3	Purdue

(d)

Figure 4. Tables for *person* entities: (a) a single table for all attributes, and (b)-(d) vertical partitions of the single table.

Supporting Temporal Data: A user may enter an incorrect data value into the database, either unintentionally or intentionally. Once detected, we need to be able to rollback the data item to its previous correct value, which may be a long time ago. To provide such undo facilities, we require G to be a temporal database. Extending a conventional database to support temporal data has been well-studied [23, 18]. Currently we use the transaction-time table solution which is described in detail in [23].

Specifically, to convert a non-temporal attribute table $T(\underline{id}, value)$ into a temporal table T', we append T with two columns, denoted as *start* and *stop*. Thus we obtain $T'(\underline{id}, value, start, \underline{stop})$. Attributes *start* and *stop* are two timestamps: *start* indicates when a value was first inserted into the database, and *stop* indicates when the value was updated or deleted. Note that the primary key of T' consists of *id* and *stop*. This is because an entity (or relation) attribute may take different values at different times. In our design, all these values are stored in the same table with the same entity (or relation) ID but different *stop* values.

Figure 5.a gives an example of a temporal table for *person_organization* (Figure 4.d). In the example, at time 2007-04-01 08:01:20, a user entered organization "UW-Madison" for the person entity with $id = 1$ (person1 for short). Attribute *start* was set to "2007-04-01 08:01:20" to indicate that the value "UW-Madison" started to be current at the time of insertion. Attribute *stop* was set to "9999-12-31 23:59:59", which is the largest timestamp, to indicate that the value would be current forever.

Moreover, when an attribute value is updated or deleted, we first *logically* delete its record, then insert a new record with the new value (for update) or a NULL value (for deletion). Consider again the table in Figure 5.a. Suppose that at time 2007-05-27 09:50:10, another user modified the organization value of person1 from "UW-Madison" to "UW". To reflect the modification in the table, first we located the record with the value "UW-Madison", and changed the *stop* time of the record to the current time, denoting that the value of "UW-Madison" stopped to exist at 2007-05-27 09:50:10. Next, we inserted a new record for value "UW". We set *start* and *stop* of the new record to "2007-05-27 09:50:10" and "9999-12-31 23:59:59", respectively, denoting that "UW" would be the current value from 2007-05-27 09:50:10 on. The temporal table after the modification is shown in Figure 5.b.

id	value	start	stop
1	UW-Madison	2007-04-01 08:01:20	9999-12-31 23:59:59
3	Purdue	2007-05-02 11:40:35	9999-12-31 23:59:59

(a)

id	value	start	stop
1	UW-Madison	2007-04-01 08:01:20	2007-05-27 09:50:10
3	Purdue	2007-05-02 11:40:35	9999-12-31 23:59:59
1	UW	2007-05-27 09:50:10	9999-12-31 23:59:59

(b)

Figure 5. Examples of transaction-time tables for *person_organization*: (a) before entity with id=1 is updated, and (b) after entity with id=1 is updated.

By adding *start* and *stop* to an attribute table and by doing logical deletions and updates, we keep track of all values that an attribute has taken, and for each value, the time period during which it was current. This way, we are able to recover an attribute value of any time in the past. Besides tracking attribute values, we also need to maintain temporal information regarding entities and relations themselves, e.g., when an entity was created, and when a relation was deleted. To store such temporal information, for each entity and relation type, we first create a special attribute *exists*, then create a temporal table for *exists* the same as we do for other attributes. Attribute *exists* can take one of the two values, 1, denoting that the corresponding instance was created or reinstated, or 0, denoting that the instance was deleted. Creating an entity or relation instance can thus be implemented as inserting a record into an *exists* table with a value of 1, and deleting an instance can thus be implemented as logically updating the value to 0. This way, we are able to tell from an *exists* table whether an instance existed at a given time.

Managing Data from Multiple Users: Multiple users may contribute data into the database. For user management (Section 6), we need to know which user inserted, updated or deleted a data item. Moreover, two users may disagree on the value for one data item. And we need to decide whose value to use in generating V_d for a wiki page (Section 4.2).

To track the source of each data item, we further extend a temporal table T' by appending a column *who*, which stores the ID of the user who entered that item. The resulting table T'' is defined as $T''(\underline{id}, value, start, \underline{stop}, who)$. Note that the primary key does not change, since we only allow one

value of each attribute to be current at any time, regardless of by whom.

Among all users, machine M is a special one. It automatically extracts and integrates data from a set of data sources. Thus it supplies data into the database much more frequently than any particular human user. On the other hand, M's data suffers from inaccuracies due to the capacity of the extraction and integration methods M uses. Consequently, M's data has lower credibility than other users' data. Therefore, we need to distinguish M from the rest of users. As a solution, for each attribute A, we create two temporal tables, $A_m(\underline{id}, value, start, \underline{stop}, who)$[1] and $A_u(\underline{id}, value, start, \underline{stop}, who)$. Table A_m stores attribute values entered by M, and table A_u stores values entered by human users.

An attribute may have different values in tables A_m and A_u. To decide which value to use in generating V_d, we need to resolve conflicts between the two tables. As a solution, we define a view table A_p over A_m and A_u. Table A_p has the same schema as A_m and A_u, and it stores the [[negotiated]] value of each attribute. Table A_p is updated when A_m or A_u is updated. Thus we can embed in its update procedure how we resolve conflicts in attribute values. Specifically, when an attribute a is updated in either of A_m and A_u, we first check whether a is already in A_p. If not, we simply insert a with its value into A_p. (Values for *start*, *stop* and *who* are assigned accordingly.) Otherwise, we need to decide whether we should overwrite a's value in A_p. A reasonable approach is to allow a user U to overwrite data entered by M or another user. We also allow M to overwrite its own data, but only allow it to overwrite U's data in certain situations, for example, when M is sufficiently confident in its data.

An example of A_m, A_u and A_p for table *person_organization* is shown in Figures 6. For simplicity of illustration, we assume that a user U can overwrite machine M's data but M cannot overwrite U's data. Based on this assumption, when user U_2 entered value "UW-Madison" for person1, we first inserted the value into table *person_organization_u*, then logically updated the existing value "UW" in table *person_organization_p*. Value "UW" was entered by M and thus we overwrote it with U_2's value. In contrast, when M entered "MITRE" for person3 into *person_organization_m*, we did not update value "Purdue" in *person_organization_p* since "Purdue" was entered by a human user.

Finally, table A_p can be explicitly stored in the database or computed as needed. In our design, we chose to materialize A_p for efficiency.

A Complete Solution: With all the problems addressed, we now present a complete solution to storing G in an RDBMS.

[1] In an A_m table, $who \equiv$"M" since M is the only machine involved. We keeps attribute who in the table so that our design is easily extensible to multiple machines.

Formally, let G_s and G_d be the schema and the data of G. Let $E_1, \ldots, E_n$ be the set of entity types in G_s, and $R_1, \ldots, R_m$ be the set of relation types in G_s. Suppose for simplicity that each relation type is binary. We create the following relational tables to store G:

- An entity ID table $Entity_ID(\underline{id}, ename)$, where id and $ename$ store the ID and the type of an entity.
- For each entity type E, we create a special attribute *exists*, whose value can be either 1 or 0. Denote *exists* as A_0. Let $A_1, \ldots, A_k$ be the attributes of E in G_s. For each attribute $A \in \{A_0, \ldots, A_k\}$, we create three temporal tables, A_m, A_u and A_p. Each table $T \in \{A_m, A_u, A_p\}$ is defined as follows:

  $$T(\underline{id}, value, start, \underline{stop}, who),$$

  where *id* is the ID of an entity, and *value* is the value of attribute A of that entity. Timestamps *start* and *stop* specifies a time period during which the value was current. And finally, *who* gives the ID of the user who entered that value.
- For each relation type R, we create a relation ID table R_ID. Let E_1 and E_2 be the two entity types that R relates in G_s, table R_ID is defined as follows:

  $$R_ID(\underline{id}, eid1, eid2),$$

  where id is the ID of a relation, and $eid1$ and $eid2$ are the IDs of the two related entities. Similar to converting an entity type, we first create attribute *exists* for R, then create tables A_m, A_u and A_p for attribute *exists* and each attribute of R in G_s.

A user may create an entity type (same for a relation type and an attribute), delete an entity type, or reinstate a deleted entity type. To enrich catalog data with temporal information, we also create three meta tables:

- Table $meta_entity(\underline{ename}, start, \underline{stop}, who)$, which stores the names of the entity types that have been created. Attributes *start*, *stop* and *who* (same for those attributes in tables $meta_relation$ and $meta_attribute$ below) have the same semantics as they do in an attribute table.
- Table $meta_relation(\underline{rname}, ename1, ename2, start, \underline{stop}, who)$, which stores the names of the relation types that have been created. For each relation type R, the table also store the names of the two entity types that R relates.
- Table $meta_attribute(\underline{tname}, \underline{aname}, category, type, start, \underline{stop}, who)$, which stores the name of each attribute (*aname*) that each entity or relation type (*tname*) has. For each attribute, the table also gives its category (atomic or set-valued) and data type (string or numeric) specifications in *category* and *type*, respectively.

Examples of the meta tables are shown in Figure 7.

person_organization_m

id	value	start	stop	who
1	UW-Madison	2007-04-01 08:01:20	9999-12-31 23:59:59	M
3	MITRE	2007-05-20 16:20:30	9999-12-31 23:59:59	M

(a)

person_organization_u

id	value	start	stop	who
3	Purdue	2007-05-02 11:40:35	9999-12-31 23:59:59	U_1
1	UW	2007-05-27 09:50:10	9999-12-31 23:59:59	U_2

(b)

⇨

person_organization_p

id	value	start	stop	who
1	UW-Madison	2007-04-01 08:01:20	2007-05-27 09:50:10	M
3	Purdue	2007-05-02 11:40:35	9999-12-31 23:59:59	U_1
1	UW	2007-05-27 09:50:10	9999-12-31 23:59:59	U_2

(c)

Figure 6. An example of attribute tables for *organization* of entity type *person*: (a) A_m, (b) A_u, and (c) A_p.

meta_entity

ename	start	stop	who
person	2007-03-12 05:10:00	9999-12-31 23:59:59	M
pub	2007-03-12 05:10:10	9999-12-31 23:59:59	M
topic	2007-03-12 05:10:20	9999-12-31 23:59:59	M
…	…	…	…

(a)

meta_relation

rname	ename1	ename2	start	stop	who
interests	person	topic	2007-03-12 05:10:30	9999-12-31 23:59:59	M
write-pub	person	pub	2007-03-12 05:10:40	9999-12-31 23:59:59	M
advise	person	person	2007-05-24 16:04:27	9999-12-31 23:59:59	U_1
…	…	…	…	…	…

(b)

meta_attribute

tname	aname	category	type	start	stop	who
person	title	atomic	CHAR(100)	2007-03-12 05:10:50	9999-12-31 23:59:59	M
person	age	atomic	INT	2007-04-18 12:40:19	2007-06-10 10:30:25	U_2
person	age	NULL	NULL	2007-06-10 10:30:25	9999-12-31 23:59:59	U_3
…	…	…	…	…	…	…

(c)

Figure 7. Examples of meta tables: (a) $meta_entity$, (b) $meta_relation$, and (c) $meta_attribute$.

4.1.3 Initializing G

To initialize G, we employ a machine-based solution M. Many such solutions exist [14]. Currently we use the Cimple solution which is described in detail in [14]. The solution works in two steps: (1) creating an entity-relationship (ER) graph, and (2) importing the ER graph into database G.

Creating an Entity-Relationship Graph: First, a community expert provides Cimple with a set of relevant data source. Use the community of database researchers as an example. Data sources can be home pages of database researchers, DBLP, conference pages, etc.. The expect also provides domain knowledge about entities and relations of interest. For example, *person* and *conference* are two entity types, and between them exists a relation type *give-talk*.

Then Cimple uses simple but focused automatic methods to create an ER graph of the community. Specifically, Cimple first crawls the sources at regular intervals to obtain data pages, then marks up mentions of relevant entities. Examples of mentions include people names (e.g., "D. DeWitt", "David J. DeWitt"), conference names, and paper titles. Next, Cimple matches mentions and groups them into entities (e.g., mentions "D. DeWitt" and "David J. DeWitt" refer to the same person entity). Cimple then discovers relations among the entities. As a result, Cimple creates an ER graph from the raw data sources.

DBLife is an example portal built using such a semi-automatic solution.

Importing the ER Graph into Database G: Cimple stores the ER graph in a set of XML files. To initialize database G, we first convert G_s into a set of relational tables, as described in Section 4.1.2. Then we use an import module to bulk load the XML data into G.

4.2 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 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 5.3). 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 5.1 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 4.1. *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 some 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 4.2. *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 neighborhood, in an intuitive manner. Second, evaluating schema paths amounts to performing selection operations over 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 [13] 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 Cwiki uses $\mathcal{L}_p$ to create views over G. First, Cwiki decides on the set of entities and relations to be "wikified". Currently, for simplicity we consider all entities, but no relations. Next, for each entity e of a particular type (e.g., person), Cwiki specifies a default view schema V_s that specifies a "neighborhood" of e. Cwiki thus specifies as many default view schemas as the number of entity types to be "wikified". These default view schemas are application specific. 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.

4.3 Converting Views to Wiki Pages

Given a view V with schema V_s and data V_d as defined above, we now discuss converting V_d into a wiki page W. In the following, we introduce our novel s-slot solution. We also discuss some other non-trivial design issues, such as the ordering of entities, the formation of URLs and the use of schema pages.

A Spectrum of Solutions: 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 is somewhat limited; hence, they get reused in multiple contexts, causing many ambiguities for the extractor. Furthermore, suppose G has been updated so that X is now working for Y'. To update 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 7 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 $T_1(C_1)\{A_1\}.\ldots.T_n(C_n)\{A_n\}$. We say that p is an *attribute path* iff $A_1 - A_{n-1}$ are empty sets and A_n identifies a single attribute a. 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.
```

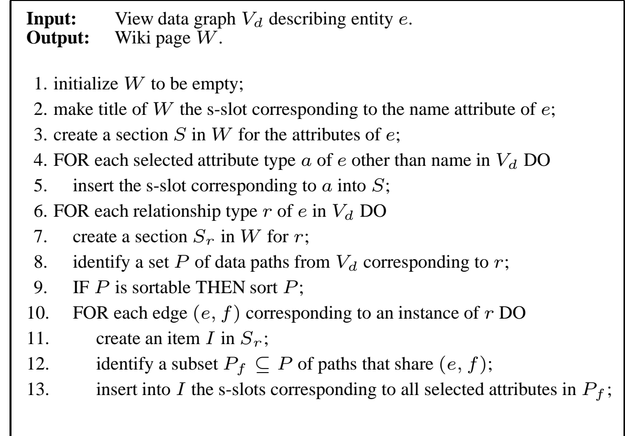
When a wiki page is rendered into an HTML page, only the value v of an s-slot $< \# \ p \ = \ v \ \# >$ is presented while other parts, as meta data, are suppressed. Thus the HTML presentation of the above example wiki text will display "David DeWitt works for UW since 1976".

An s-slot of $<\# \ p = v \ \#>$ can be marked with a "nodisplay" attribute as in $<\# \ p = v$ nodisplay$\#>$. In this case, the whole s-slot will be suppressed and even the value v will not be presented in the HTML page. Such s-slots are useful when the values are confidence scores used for entity ordering, as to be discussed shortly.

An s-slot of $< \# \ p \ = \ v \ \# >$ can also be marked with an "invalid" attribute as in $< \# \ p \ = \ v$ invalid$\# >$, indicating that the path p is broken and unsupported by the underlying database, and thus, the validity of the value v expired. This situation is generally caused by deletion of structured data from other related wiki pages. When a page containing invalid structured data is requested, the "invalid" attributes will be added by machine for the corresponding s-slots. $<\# \ p = v$ invalid$\#>$ will be presented in the HTML page as "v(invalid)", reminding the user of the fact and leaving him/her the right to delete the s-slot or fix the broken path.

Now let V be a view with schema V_s that Cwiki has defined over database G (see Section 4.2). Then Cwiki generates the default wiki page W for V in two steps: (a) evaluates V_s over G to obtain the view data V_d, which is a subgraph of the ER graph G, then (b) convert 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. In the following, we explain the algorithmic details about how to generate a wiki page W from a view data graph V_d.


Input: View data graph V_d describing entity e.
Output: Wiki page W.

1. initialize W to be empty;
2. make title of W the s-slot corresponding to the name attribute of e;
3. create a section S in W for the attributes of e;
4. FOR each selected attribute type a of e other than name in V_d DO
5. insert the s-slot corresponding to a into S;
6. FOR each relationship type r of e in V_d DO
7. create a section S_r in W for r;
8. identify a set P of data paths from V_d corresponding to r;
9. IF P is sortable THEN sort P;
10. FOR each edge (e, f) corresponding to an instance of r DO
11. create an item I in S_r;
12. identify a subset $P_f \subseteq P$ of paths that share (e, f);
13. insert into I the s-slots corresponding to all selected attributes in P_f;

Figure 8. Generating wiki page W from view data V_d

The Algorithm Generating W from V_d: The algorithm presented in Figure 8 generates a wiki page W from a given view data graph V_d for entity e. In line 1, W is initialized to be empty. In line 2, the s-slot corresponding to the name attribute of e is made title of W. In lines 3–4, a section is created in W for other selected attributes of e, that provides the basic attributional information describing e. In lines 5–13, a section is created in W for each relationship type.

Each section is labeled properly with a uniform default look. This can be done since in building an initial community portal $\mathcal{W}$, the only participating user is the portal builder, for whom the semantics of each attribute type, entity type and relationship type are transparent to him/her since he/she was the one who created the initial view schema V_s. The selection of view V delivers the builder's intention and the look of the wiki page represents his/her preferences. For the same reason, in line 7, the data paths in V_d for relationship type r can be extracted properly. Notice that V_d itself does not embed such information that how it should be decomposed and presented. Rather, the extraction mechanisms are hard-coded for each relationship section. For example, for the "writes" relationship, the paths of type *person.write-pub.pub.write-pub.person* starting at entity e are extracted from V_d.

In line 8, the extracted paths are possibly sorted if the ordering information is provided in the paths. In lines 9–13, the extracted paths are grouped such that each group corresponds to a unique instance of the relationship type r. Then, the s-slots for the selected attributes of each group form an item and the item is inserted into the section.

The portal builder has every reason to capture the preferences of the majority of users. The above hard-coded interpretation mechanism translates V_d into a default wiki page W, so that the initial community portal $mathw$ features

HTML pages with a uniform look that is easy to the eyes of the majority of users. Later, W would be edited by different individuals, and this default interpretation mechanism will not be used in updating W by machine, in order not to intervene users' intentions and interpretations. Instead, all the fresh structured contents will be inserted into a special section called "New", from which users can pick up items and move them around according to their own preferences.

Ordering of Entities: Handling the ordering of entities is a non-trivial design issue. In many cases, entities have a natural ordering depending on how much they relate to a common entity. For example, the related people of a person can be ordered by the closeness of their relationships to that person. The related topics of a person can be ordered by the degree of interest and involvement of that person in those topics. As another obvious example, all the authors of a publication must be ordered by how they appear in the publication. To capture the ordering information, we assign each involving relationship a confidence score as attribute.

In order for applicable entities to appear ordered in the HTML page, the confidence score attribute needs to be selected in V_s. Then, the corresponding data paths in V_d will present this ordering information and be ordered properly by the algorithm (line 9) converting V_d to W. The corresponding s-slots in W will be marked with "nodisplay" and thus those actual confidence score values will not be displayed in the HTML page. This handling of entity ordering is not meant to be systematic and sophisticated to cover arbitrary ordering needs; rather, it focuses on simplicity and adequacy in terms of fulfilling the basic ordering functionality.

Formation of URLs: The formation of URLs raises another non-trivial design issue. For the HTML page specified by a URL of http://dblife-labs.cs.wisc.edu/wiki-test/index.php/David_DeWitt, the corresponding wiki page will have a URL of http://dblife-labs.cs.wisc.edu/wiki-test/index.php?title=David_DeWitt&action=edit. "David DeWitt" is the *page title* for the HTML page as well as the wiki page. As page title is the only replaceable element in a URL, the formation of URLs comes down to the formation of page titles.

Within the same namespace, each entity e must have a unique page title. A natural solution to achieve this uniqueness is to use entity ID's as titles. However, such page titles are neither informative to users nor cooperative with search engines. Entity names seem to be the most informative titles; however, they cannot guarantee the uniqueness since multiple entities may share the same name. In our design, a mapping table is maintained to map each entity ID to a unique page title. In general cases, entity names are used as page titles. In cases a title is used by another entity, a concatenation of entity name and ID will be used.

In particular, we create a mapping table with three fields *eid*, *title*, and *type*, storing entity ID's, page titles and entity types respectively. Both *eid* and *title* are keys. When a new entity e is inserted into the database, a default wiki page will be created for e and the mapping table is used to generate the page title. First, the entity ID of e, say 15, is checked against existing ones in the mapping table. If no duplicates, the name of e, say "David DeWitt", is then checked against existing page titles in the table. If no duplicates, 15 and "David_DeWitt" will form a tuple and be inserted into the table. Otherwise, a concatenation of "David_DeWitt" and 15, i.e., "David_DeWitt15", will be used instead as the page title. Obviously, the page titles thus-generated are guaranteed to be unique.

Use of Schema Page: As to be discussed in §5.1, we expose view schemas in wiki pages to allow user editing. Thus, a default schema page W_s will be created for each default wiki page W. W_s will have the same page title as W but under the namespace of "Schema". For example, the URL for the schema page of the wiki page of David DeWitt will be http://dblife-labs.cs.wisc.edu/wiki-test/index.php/Schema:David_DeWitt.

Since all the default schema pages of the same type differ only in entity ID, they can be automatically generated and bulk-loaded into the system when building the initial community portal. In particular, all the entities are first registered in the mapping table. Then, a default schema page is generated for each tuple in the table according to the entity type stored in the table. Next, all these schema pages are written in a single file, which is then bulk-loaded into the database supporting the wiki system, without utilizing the interface of the system.

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.

5 Managing User Contributions

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

5.1 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.

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, a 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, 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 Cwiki 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.

5.2 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 complete sets of basic actions on V_d, V_s, G_d and G_s are given in Table 1.

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.

Table 2. Basic ER actions that we have defined.

Appendix A give 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 Table 2 lists the ten basic ER actions we have defined. We have implemented each ER action as a 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 Table 2) 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 Table 2 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

	Actions on V_d	**Actions on V_s**	**Actions on G_d**	**Actions on G_s**
a_1	Modify an entity attribute value	Insert an entity attribute	Modify an entity attribute value	Create an entity attribute
a_2	Modify a relation attribute value	Insert a relation attribute	Modify a relation attribute value	Create a relation attribute
a_3	Insert an entity attribute	Insert an entity	Insert an entity attribute	Create an entity type
a_4	Insert a relation attribute	Insert a relation	Insert a relation attribute	Create a relation type
a_5	Insert an entity	Delete an entity attribute	Insert an entity	Drop an entity attribute
a_6	Insert a relation	Delete a relation attribute	Insert a relation	Drop a relation attribute
a_7	Delete an entity attribute	Delete an entity	Delete an entity attribute	Delete an entity type
a_8	Delete a relation attribute	Delete a relation	Delete a relation attribute	Delete a relation type
a_9	Delete an entity		Delete an entity	
a_{10}	Delete a relation		Delete a relation	

Table 1. Basic relational actions on V_d, V_s, G_d and G_s.

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 ∈ E' − E DO
2.    IF entity type exists THEN append a_4 to S_ER;
3.    ELSE append a_5 to S_ER;
4. FOR each relationship instance r ∈ R' − R DO
5.    IF relationship type exists THEN append a_6 to S_ER;
6.    ELSE append a_7 to S_ER;
7. FOR each attribute a ∈ A' − A DO
8.    IF attribute type exists THEN append a_2 to S_ER;
9.    ELSE append a_3 to S_ER;
10. FOR each attribute a ∈ A − A' DO
11.    append a_8 to S_ER;
12. FOR each relationship instance r ∈ R − R' DO
13.    append a_10 to S_ER;
14. FOR each entity instance e ∈ E − E' DO
15.    append a_9 to S_ER;
16. FOR each attribute a ∈ A ∩ A' DO
17.    IF it has the same value in V_d and V_d' THEN append a_1 to S_ER;
18. Return S_ER;
```

Figure 9. Generating $\mathcal{S}_{ER}$ from V_d and V_d'

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 $\mathcal{S}_{ER}$. Figure 9 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 Table 2), 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.*

5.3 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 7 empirically shows that we can refresh pages on the fly quickly, in a few seconds, thus making this lazy approach a practical solution.

6 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 [13] 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 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)".

7 Empirical Evaluation

To evaluate Cwiki, we have been applying it to build a community wikipedia for the database community (see [1] for the current portal, still under continuous development). We now report on preliminary experiments with this portal, which demonstrate the potentials of Cwiki and suggest research opportunities.

Building an Initial Community Portal: We began by employing DBLife as machine M (see Section 4). 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 [14]). 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 entities and nine relationships, 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 efforts.

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 G to efficiently accommodate frequent changes in uncertainty scores.

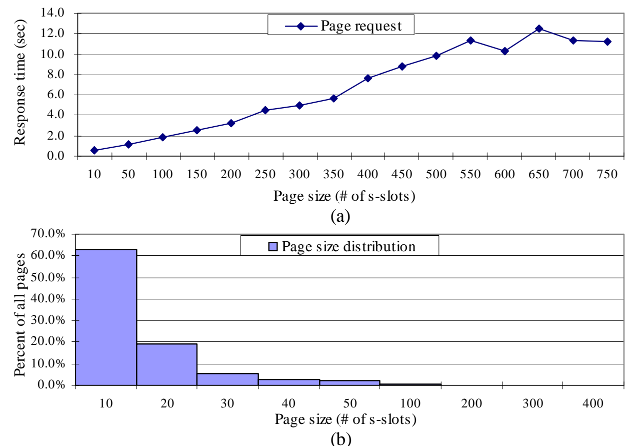


Figure 10. Time to request a wiki page and distribution of page size.

# 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

Figure 11. Time to process user edits on a wiki page.

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 4.3) 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 10.a shows the time it takes from when a user requests a page W until when W is served. Note that to ensure freshness,

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%

Figure 12. User performance on several editing tasks.

we materialize W on the fly, from the underlying structured database G and text database T (Section 5.3). 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. Figure 10.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 5.3) 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 11 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 5.2). 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 Cwiki 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 6 users, 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 them to finish the given editing tasks and the correctness of the results. For comparison purposes, we also asked users to edit some free text.

Figure 12 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 12 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 macro 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 4.3, a lesson we learned from our current Cwiki 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 spectrum end and the natural-language solution (see Section 4.3) 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 so-so for (a). The natural-language solution is best for (a), so-so for (c), and difficult for (b). An XML-like solution appears best for (a) and (b), but so-so for (c). Developing more solutions, evaluating them, and selecting a good one is an interesting future research direction.

8 Conclusions & Future Work

We have described Cwiki, 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 efforts. We have applied Cwiki to build a "wikipedia" portal for the database community [1]. We reported on our experience with this portal that demonstrates the potentials of Cwiki 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). Virtually any problem that we have discussed can be "drilled down" deeper. 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.

In addition, we will continue to develop the structured wikipedia for the database community [1], as a real-world application that we can use to evaluate Cwiki. Finally, we plan to release the Cwiki source code to encourage further development and evaluation of community wikipedias in additional domains.

References

[1] http://dblife-labs.cs.wisc.edu/wiki-test/index.php/main_page.

[2] http://en.wikipedia.org/.

[3] http://en.wikipedia.org/wiki/semantic_wiki.

[4] http://metaweb.com/.

[5] http://oak.cs.ucla.edu/blogocenter.

[6] http://rexa.info/.

[7] Sixth international workshop on information integration on the web. 2007.

[8] S. Amer-Yahia. A database solution to search 2.0. *WebDB-07*.

[9] I. Androutsopoulos, G. D. Ritchie, and P. Thanisch. Natural language interfaces to databases–an introduction. *Journal of Language Engineering*, 1(1):29–81, 1995.

[10] F. Bancilhon and N. Spyratos. Update semantics of relational views. *ACM Transactions on Database Systems*, 6(4):557–575, 1981.

[11] N. Bansal and N. Koudas. Blogscope: Spatio-temporal analysis of the blogosphere. In *WWW-07*.

[12] U. Dayal and P. A. Bernstein. On the correct translation of update operations on relational views. *ACM Transactions on Database Systems*, 7(3):381–416, 1982.

[13] 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/cwiki.pdf*, 2007.

[14] 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*.

[15] 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.

[16] 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*, 29(1), 2006.

[17] C. Giles, K. Bollacker, and S. Lawrence. Citeseer: An automatic citation indexing system. In *DL-98*.

[18] H. Gregersen and C. S. Jensen. Temporal entity-relationship models - a survey. *Knowledge and Data Engineering*, 11(3):464–497, 1999.

[19] Y. Li, H. Yang, and H. V. Jagadish. Constructing a generic natural language interface for an xml database. In *EDBT-06*.

[20] R. McCann, A. Kramnik, W. Shen, V. Varadarajan, O. Sobulo, and A. Doan. Integrating data from disparate sources: A mass collaboration approach. In *ICDE-05*.

[21] Z. Nie, J. Wen, and W. Ma. Object-level vertical search. In *CIDR-07*.

[22] R. Ramakrishnan. Community systems: The world online. In *CIDR-07*.

[23] R. T. Snodgrass. *Developing Time-Oriented Database Applications in SQL*. Morgan Kaufmann Publishers, Inc., 1999.

[24] M. Volkel, M. Krotzsch, D. Vrandecic, H. Haller, and R. Studer. Semantic wikipedia. In *WWW-06*.

[25] F. Wang, C. Rabsch, P. Kling, P. Liu, and P. John. Web-based collaborative information integration for scientific research. In *ICDE-07*.

Appendix

A Implementing Basic Relational Actions

We define a set of basic relational actions that a user can execute over V_d, V_s, G_d and G_s. There are 10 actions for V_d, 8 for V_s, 10 for G_d, and 8 for G_s. In the following, we give our implementation of each action. To distinguish actions in different categories, we prefix each action by its category name. For example, we denote action a_i for V_d as V_d::a_i.

A.1 Actions for V_d

Action a_1: Modify an Entity Attribute Value
Steps:

1. Execute G_d::a_1.

Action a_2: Modify a Relation Attribute Value
Steps:

1. Execute G_d::a_2.

Action a_3: Insert an Entity Attribute

Let eid be the entity ID and E be the entity type. Let A be the attribute to insert.

Steps:

1. Execute G_d::a_3;
2. Add an inclusive path $E(id = eid)\{A\}$ to V_s.

Action a_4: Insert a Relation Attribute

Let rid be the relation ID and R be the relation type. Let A be the attribute to insert.

Steps:

1. Execute G_d::a_4;
2. Add an inclusive path $R(id = rid)\{A\}$ to V_s.

Action a_5: Insert a New Entity

Let e be the entity to insert and E be its type.

Steps:

1. Execute G_d::a_5, let eid be the ID of e;
2. Add an inclusive path $E(id = eid)$ to V_s.

Action a_6: Insert a New Relation

Let r be the relation to insert and R be its type. Let eid_1 and eid_2 be the IDs of the two entities that r relates, and E_1 and E_2 be their types.

Steps:

1. Execute G_d::a_6, let rid be the ID of r;
2. Add an inclusive path $E_1(id = eid_1).R(id = rid).E_2(id = eid_2)$ to V_s.

Action a_7: Delete an Entity Attribute

Let eid be the ID of the entity and E be its type. Let A be the attribute to delete.

Steps:

1. Execute G_d::a_7;
2. Add an exclusive path $E(id = eid)\{A\}$ to V_s.

Action a_8: Delete a Relation Attribute

Let rid be the ID of the relation and R be its type. Let A be the attribute to delete.

Steps:

1. Execute G_d::a_8;
2. Add an exclusive path $R(id = rid)\{A\}$ to V_s.

Action a_9: Delete an Entity

Let e be the entity to delete. Let eid be e's ID and E be e's type.

Steps:

1. FOR each relation r that relates e DO
   Execute V_d::a_{10};
2. FOR each attribute A (including *exists*) of e DO
   Execute V_d::a_7;
3. Add an exclusive path $E(id = eid)$ to V_s.

Action a_{10}: Delete a Relation

Let r be the relation to delete. Let rid be r's ID and R be r's type. Let eid_1 and eid_2 be the IDs of the entities that r relates, and E_1 and E_2 be their types.

Steps:

1. FOR each attribute A (including *exists*) of r DO
   Execute V_d::a_8;
2. Add an exclusive path $E_1(id = eid_1).R(id = rid).E_2(id = eid_2)$ to V_s.

A.2 Actions for V_s

Action a_1: Insert an Entity Attribute

Let eid be the ID of the entity and E be its type. Let A be the attribute to insert.

Steps:

1. Add an inclusive path $E(id = eid)\{A\}$ to V_s.

Action a_2: Insert a Relation Attribute

Let rid be the ID of the relation and R be its type. Let A be the attribute to insert.

Steps:

1. Add an inclusive path $R(id = rid)\{A\}$ to V_s.

Action a_3: Insert an Entity

Let eid be the ID of the entity and E be its type.

Steps:

1. Add an inclusive path $E(id = eid)$ to V_s.

Action a_4: Insert a Relation

Let r be the relation to insert. Let rid be r's ID and R be r's type. Let eid_1 and eid_2 be the IDs of the entities that r relates, and E_1 and E_2 be their types.

Steps:

1. Add an inclusive path $E_1(id = eid_1).R(id = rid).E_2(id = eid_2)$ to V_s.

Action a_5: Delete an Entity Attribute

Let eid be the ID of the entity and E be its type. Let A be the attribute to delete.

Steps:

1. Add an exclusive path $E(id = eid)\{A\}$ to V_s.

Action a_6: Delete a Relation Attribute

Let rid be the ID of the relation and R be its type. Let A be the attribute to delete.

Steps:

1. Add an exclusive path $R(id = rid)\{A\}$ to V_s.

Action a_7: Delete an Entity

Let e be the entity to delete. Let eid be e's ID and E be e's type.

Steps:

1. FOR each relation r that relates e DO
   Execute V_s::a_8;
2. FOR each attribute A (including *exists*) of e DO
   Execute V_s::a_5;
3. Add an exclusive path $E(id = eid)$ to V_s.

Action a_8: Delete a Relation

Let r be the relation to insert. Let rid be r's ID and R be r's type. Let eid_1 and eid_2 be the IDs of the entities that r relates, and E_1 and E_2 be their types.
Steps:

1. FOR each attribute A (including *exists*) of r DO
   Execute V_s::a_6;
2. Add an exclusive path $E_1(id = eid_1).R(id = rid).E_2(id = eid_2)$ to V_s.

A.3 Actions for G_d

We use the following notations to represent tables in G:
E_A_m – table for attribute A of entity type E that stores attribute values entered by machine M;
E_A_u – table for attribute A of entity type E that stores attribute values entered by human users;
E_A_p – table for attribute A of entity type E that stores attribute values used in generating V_d;
R_A_m, R_A_u, R_A_p – similar to those above but for relation type R instead;
R_ID – relation ID table for relation type R.

Action a_1: Modify an Entity Attribute Value
Let E be the type of the entity and A be the attribute. Let w be the ID of the user who modifies A.
Steps:

1. IF $w = M$ THEN
   Logically delete the current value of A in E_A_m;
   Insert the new value of A into E_A_m;
   ELSE
   Logically delete the current value of A in E_A_u;
   Insert the new value of A into E_A_u;
2. IF the current value in E_A_p was entered by M
   OR w != M THEN
   Logically delete the current value of A in E_A_p;
   Insert the new value of A into E_A_p;

Action a_2: Modify a Relation Attribute Value
Let R be the type of the relation and A be the attribute. Let w be the ID of the user who modifies A.
Steps:

1. IF $w = M$ THEN
   Logically delete the current value of A in R_A_m;
   Insert the new value of A into R_A_m;
   ELSE
   Logically delete the current value of A in R_A_u;
   Insert the new value of A into R_A_u;
2. IF the current value in R_A_p was entered by M
   OR w != M THEN
   Logically delete the current value of A in R_A_p;
   Insert the new value of A into R_A_p;

Action a_3: Insert an Entity Attribute
Let eid be the ID of the entity and E be its type. Let A be the attribute and w be the ID of the user who inserts A.
Steps:

1. IF $w = M$ THEN
   IF exists a record with $id = eid$
   AND *stop*="9999-12-31 23:59:59" in E_A_m THEN
   Logically delete the record;
   Insert the new value of A into E_A_m;
   ELSE
   IF exists a record with $id = eid$
   AND *stop*="9999-12-31 23:59:59" in E_A_m THEN
   Logically delete the record;
   Insert the new value of A into E_A_u;
2. IF exists a record with $id = eid$
   AND *stop*="9999-12-31 23:59:59" in E_A_p THEN
   IF the record was entered by M OR $w! = M$ THEN
   Logically delete the record;
   Insert the new value of A into E_A_p;

Action a_4: Insert a Relation Attribute
Let rid be the ID of the relation and E be its type. Let A be the attribute and w be the ID of the user who inserts A.
Steps:

1. IF $w = M$ THEN
   IF exists a record with $id = rid$
   AND *stop*="9999-12-31 23:59:59" in R_A_m THEN
   Logically delete the record;
   Insert the new value of A into R_A_m;
   ELSE
   IF exists a record with $id = rid$
   AND *stop*="9999-12-31 23:59:59" in R_A_m THEN
   Logically delete the record;
   Insert the new value of A into R_A_u;
2. IF exists a record with $id = rid$
   AND *stop*="9999-12-31 23:59:59" in R_A_p THEN
   IF the record was entered by M OR $w! = M$ THEN
   Logically delete the record;
   Insert the new value of A into R_A_p;

Action a_5: Insert a New Entity
Let E be the entity type and max_eid be the largest ID in table *entity_ID*.
Steps:

1. Insert record $(max_eid + 1, E)$ into *entity_ID*.

Action a_6: Insert a New Relation
Let r be the relation to insert and R be its type. Let $eid1$ and $eid2$ be the IDs of the entities that r relates. Let max_rid be the largest ID in table *R_ID*.
Steps:

1. Insert record $(max_rid + 1, eid1, eid2)$ into *R_ID*.

Action a_7: Delete an Entity Attribute
Steps:

1. Execute G_d::a_1 with NULL as the attribute value.

Action a_8: Delete a Relation Attribute
Steps:

1. Execute G_d::a_2 with NULL as the attribute value.

Action a_9: Delete an Entity
Let e be the entity to delete.
Steps:

1. FOR each relation r that relates e DO
   Execute G_d::a_{10};
2. FOR each attribute A (including *exists*) of e DO
   Execute G_d::a_7;

Action a_{10}: Delete a Relation

Let r be the relation to delete.

Steps:

1. FOR each attribute A (including *exists*) of r DO
   Execute G_d::a_8;

A.4 Actions for G_s

Action a_1: Create an Entity Attribute

Let E be the type of the entity and A be the attribute to create.

Steps:

1. Insert attribute A into table *meta_attribute*;
2. Create tables E_A_m, E_A_u and E_A_p.

Action a_2: Create a Relation Attribute

Let R be the type of the relation and A be the attribute to create.

Steps:

1. Insert attribute A into table *meta_attribute*;
2. Create tables R_A_m, R_A_u and R_A_p.

Action a_3: Create an Entity Type

Let E be the entity type to create.

Steps:

1. Insert entity type E into table *meta_entity*;
2. Execute G_s::a_1 for attribute *exists*.

Action a_4: Create a Relation Type

Let R be the relation type to create.

Steps:

1. Insert relation type R into table *meta_relation*;
2. Create table R_ID;
3. Execute G_s::a_2 for attribute *exists*.

Action a_5: Drop an Entity Attribute

Let E be the entity type and A be the attribute to drop.

Steps:

1. Logically delete attribute A in table *meta_attribute*;
2. Logically delete all records in E_A_m, E_A_u and E_A_p.

Action a_6: Drop a Relation Attribute

Let R be the relation type and A be the attribute to drop.

Steps:

1. Logically delete attribute A in table *meta_attribute*;
2. Logically delete all records in R_A_m, R_A_u and R_A_p.

Action a_7: Drop an Entity Type

Let E be the entity type to drop.

Steps:

1. FOR each relation type R that relates E DO
   Execute G_s::a_8;
2. FOR each attribute A (including *exists*) of E DO
   Execute G_s::a_5;
3. Logically delete entity type E in *meta_entity*;

Action a_8: Drop a Relation Type

Let R be the relation type to drop.

Steps:

1. FOR each attribute A (including *exists*) of R DO
   Execute G_s::a_6;
2. Logically delete entity type R in *meta_relation*;