One Table Stores All: Enabling Painless Free-and-Easy Data Publishing and Sharing

[System Overview]

Bei Yu
School of Computing
National University of Singapore
yubei@comp.nus.edu.sg

Guoliang Li
Department of Computer Science and Technology
Tsinghua University
liguoliang@tsinghua.edu.cn

Beng Chin Ooi
School of Computing
National University of Singapore
ooibc@comp.nus.edu.sg

Li-Zhu Zhou
Department of Computer Science and Technology
Tsinghua University
dcszlz@tsinghua.edu.cn

ABSTRACT

In this paper, we present a free-and-easy data publishing and sharing system based on folksonomy. The system accepts data objects described with user-created metadata, called *data units*. The system supports flexible structure on the data units, and places no restrictions on the vocabulary used. We devise a generic table model for storing and representing the data units of various structures. We propose a framework for managing the data units and providing browsing, searching and querying services over them. We present our current approaches and discuss relevant research issues.

1. INTRODUCTION

Digital information publishing and searching becomes increasingly necessary in recent years, due to the popularity of the Internet services. We have witnessed the growth of a number of such web services including Google Base [3], Delicious [1], liveplasma [4], and flickr [2]. While these systems vary in their particular services provided, they share the same operation mode — users publish data items such as URLs, pictures, advertisements, etc. that are associated with simple descriptions such as tags and attributes created by them. The system organizes the published data items based on users' own descriptions and makes them accessible. For example, Delicious allows an user to bookmark URLs in its server, and requires each URL to have a set of user-provided tags (a.k.a. keywords). In some sense, the users collaboratively contribute tags to the system, which are then used to categorize the URLs to facilitate webpage searching (by browsing or querying the tags). Such collaborative but unsophisticated way of organizing information with user-created metadata is coined as *folksonomy* (combination of "folk" and "taxonomy"), and such systems are sometimes also called the collaborative tagging systems [13]. A distinguishing feature of these systems is that the user plays the dominating role — the collaborative behavior of the users decides the data semantics and organization of the systems. Although the lack of controlled vocabulary and systematic taxonomy of the concepts makes the classification of the data objects in these systems imprecise and imperfect, they are convenient to users and easy to deploy, and more importantly, they can adapt to the dynamic changes of the Web content.

Besides simple tags, richer and flexible data structure could be used to describe a published item to provide more powerful expressiveness to the user. For example, Google Base allows users to define their own attributes, and to describe their published objects with variable number of attributes. The freedom from a strict syntax of the published data items is very convenient to users. However, it is a challenging task to organize and classify the data items with variable "schemas" and topics, in order to make them searchable. In Google Base, a list of types such as *products*, *recipes*, are provided, and users are encouraged but not restricted to publish their data item to a specific type.

In this paper, we describe our attempt to build a general system framework for supporting such free-and-easy data publishing and searching services. The system accepts data objects described with user-created metadata, stores and classifies them, and provides various querying and searching interfaces such as browsing, keyword search, and structured query. The metadata created by users can be both for their own use (for labeling their published information to the system) and for the system to use to organize all the published information. The data unit, that users use to describe their published information of any kind, consists of

title, a number of *fields*, and a set of *tags*. Basically, a *field* is an attribute/value pair for describing certain property of an object, e.g., *color:yellow* for a puppy. A *tag* is a word or a phrase the user uses as "keyword" to characterize the published object. For example, the tags of a puppy could be *animal*, *dog*, etc.. [13] discusses 7 types of tags an user uses to label URL bookmarks on Delicious website. Figure 1 shows two example data units that describe different types of information. To illustrate, the left data unit describes the blog of uzzer. It uses four fields for showing the location, author, type, and language of the blog. In addition, it has 9 tags that are "keywords" of the blog.

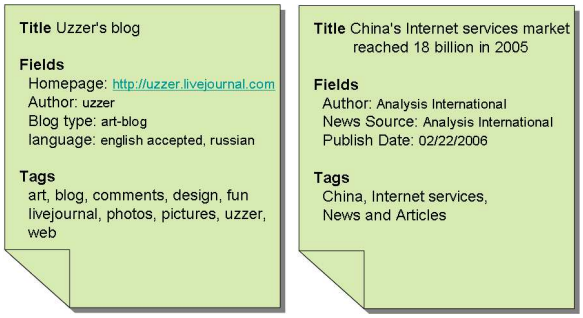


Figure 1: Examples of data units.

Within our system framework, we propose a data model for storing and representing the collection of data units accepted from users. It includes a single generic table for storing the data units, and a set of virtual relations as views of the generic table for representing different topics of the data stored in the generic table. The user browses and queries over the virtual views, and the system retrieves results from the generic table. Our generic table model differs from the universal relation model [17, 12] proposed and studied earlier in two main points. First, the universal relation is designed for logical representation of an application domain in order to free the user from dealing with specific access paths when issuing queries, while our generic table is the schema for physically representation and store of the tuples, which is not visible to the user. Second, compared with that the universal relation schema describes a specific application domain, our generic table model is for comprehensively storing data of all types of domains.

Our proposed system framework also includes a data units categorizer that dynamically clusters and assigns incoming data units into various virtual relations according to their different topics and structures, a multi-function query processor for dealing with various kinds of queries, a storage manager for storing and indexing the data units whose volume may grow quickly depending on the popularity of the system. Figure 2 illustrates the architecture of our system.

The rest of the paper is organized as follows. We first present the data model in our system in Section 2. Following it, we generally describe our system framework in Section 3. Next, we discuss our design and implementation status of the three main components: data units categorizer, storage manager, and query processor in Section 4, Section 5, and Section 6, respectively. Then, we present the related work to our approach in Section 7 and finally conclude the paper in Section 8.

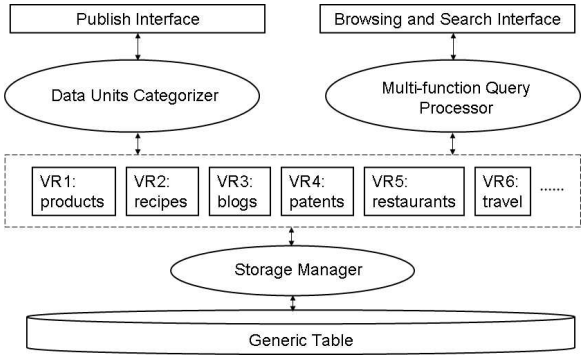


Figure 2: System architecture. (VR is the short for Virtual Relation.)

2. DATA MODEL

The data model in our system is a generic relational table and a set of virtual relations that are views over the generic table. Data units published to the system are all physically stored in the generic table in relational format. Different from traditional relational database design, the schema of the generic table in our system is designed in an ad-hoc and dynamic way. It contains a set of fixed attributes and a set of non-fixed attributes. Fixed attributes are defined by the system, while non-fixed attributes are dynamically inserted according to the data units published to the system. That is, its schema is collaboratively decided by users. The choice of such a single table model is suitable for storing the data units in our system because there is no predictable data dependencies among the attributes collaboratively defined by mass users.

Definition 1 *The generic relational table schema is an expression of the form $R(U)$, where R is the name of the table, and U is the set of attributes such that $U = U_F \cup U_N$ and $U_F \cap U_N = \phi$. $U_F = \{A_1, A_2, \cdots, A_m\}$ is the finite set of fixed attributes, $U_N = \{A_{m+1}, A_{m+2}, \cdots\}$ is the infinite set of non-fixed attributes.*

The domains of the attributes in U_N are initially undefined, and assumed to be infinite and stored as string format. When the generic table is populated with enough number of tuples, we can use machine learning approach to learn the patterns and statistics of the values in order to determine the domains of attributes.

In our system, the fixed attributes set U_F contains: *id*, *author*, *title*, and *tags*, where *id* is system created when a new data unit is inserted and it is the primary key of the table, *author* and *title* correspond to the person who publishes the data unit and the title of it, and *tags* is a textual attribute for storing all the tags listed in the data unit. When the system initializes, non-fixed attributes set U_N is empty. More and more attributes will be automatically added to it when data units having new attributes are published to the system continuously. On the other hand, when obsolete data units are deleted from the system, some attributes may also be removed from U_N.

In the generic table, *null* values are allowed. The semantics of the *null* is treated as *inapplicable*, and the operations on *null* values are the same as in traditional relational model.

When the system adds an attribute to U_N of the generic table, all tuples of the current instance I are assigned *null* values for the new attribute, and they maintain equivalence with their original forms.

The generic table is the schema for storing data units in our system, but it cannot be semantically meaningful to users, because the data units stored in it are very diverse in their topics. In our system, the user poses queries in terms of a set of virtual relations, which are defined as views of the generic table. A virtual relation schema is mapped to U_F and a subset of the attributes in U_N of the generic table. The set of virtual relations $(R_1, R_2, \cdots, R_n)$ defined in the system is called the *virtual schema* (or *semantic schema*) of the generic table.

Definition 2 *A virtual relation $R'(U)$ of the generic table $R(U_F, U_N)$ is defined as a view with a query q_I over instance I of $R(U_F, U_N)$, denoted as $R'(U) \mapsto R(U_F, U_N) = q_I$.*

We allow different virtual relations to have overlap in the data units they contain, since a data unit may be relevant to multiple topics. In this case, the same data unit may be presented as tuples with different schemas based on the virtual relations that contain it.

We can query the generic table or virtual relations with attribute-based queries in terms of selection (σ) and projection (π). In addition to ordinary comparison operators such as $=, <, >$, there are also textual-based comparisons in our data model, since most data units have lots of textual fields, e.g., *tags* attribute. Given a set of keywords $K = (k_1, k_2, \cdots, k_q)$, a relation $R(U)$ and its instance I, and a subset of textual attributes $V \subseteq U$, for each tuple $t \in I$, the textual comparison operator $match(t[V], K)$ returns a score $score(t[V], K) \in [0, 1]$ indicating its relevance to the keywords. The *match* operator relies on a fulltext index on the keywords of the values of textual attributes. Given a threshold τ, $\sigma_{match(V,K)>\tau}(I)$ will return all tuples in I whose scores are matched higher than τ.

Definition 3 *A virtual schema $\{R_1(U_1), R_2(U_2), \cdots, R_n(U_n)\}$ is complete in terms of an instance I of the generic table $R(U_F, U_N)$ if $I(R_1(U_1)) \bowtie_o I(R_2(U_2)) \bowtie_o \cdots \bowtie_o I(R_n(U_n)) = I$, where $\bowtie_o$ denotes outer join, and $I(R_i(U_i))(1 \leq i \leq n)$ is the instance of $R_i(U_i)$ generated by evaluating query q_i, which is the view defined for $R_i(U_i)$, over I.*

With complete virtual schema, the content stored in the generic table can be fully exposed to users.

3. SYSTEM FRAMEWORK

As a collaborative publishing and searching system, our system initializes with an empty generic table *AllUnits* with system created attributes, defined as

$$AllUnits(id, author, title, tags).$$

As users publish data units to the system, the generic table is populated with more and more tuples, and consequently it will have more and more attributes.

When storing a data unit into the generic table, we represent it as a tuple according to the schema of the generic table. If there are attributes defined in the data unit that are not in $U_N \in AllUnits$, the system will add those attributes to U_N during the insertion of the new tuple. For example, if upon the system initialization, the two data units shown in Figure 1 are published to the system one at a time, the resultant generic table is shown in Table 1. A data unit becomes a tuple in the generic table after it is stored in the system. In the rest of the paper, we will use data unit and tuple interchangeably for referring to the tuples stored in the tables.

The generic table is maintained by the storage manager component (Figure 2), and it is not visible to the user. Users access the stored data through the virtual relations over the generic table, which are built and updated incrementally as new data units are published to the system. Each virtual relation should represent a semantic category meaningful to the user. For example, a virtual relation $blog(blog\_name, blog\_type, homepage)$ represents a category of data units describing blogs. Since the domains of published data units are unrestricted, constructing the virtual relations is identified as the task of incrementally clustering the incoming data units into various virtual relations. This task is performed by the data unit categorizer depicted in Figure 2.

Therefore, our system need perform several actions when accepting a new data unit, which is submitted via the publish interface, a form indicating various fields users can provide. First, the data unit categorizer either assigns the new data unit to one or more existing virtual relations, or creates a new virtual relation with the new data unit as the only tuple. Then, the data unit is passed to the storage manager, which actually inserts the new data unit into $AllUnits$, and updates the mapping between the virtual relation accepting the new tuple and $AllUnits$ if the new tuple causes changes of the schema of $AllUnits$. It also updates the related indices built over the generic table.

We intend to provide in the multi-function query processor (Figure 2) a broad range of services over the published data to the user including:

- *Virtual schema browsing:* The user can browse the virtual relation schemas in a manner similar to browsing each of semantic categories of a large content classification system, in order to zoom the query to one or more categories he or she is interested in.
- *Keyword search:* Given a keyword query, we return a list of matching data units ranked according to their estimated relevance to the query.
- *Structured querying:* We will also provide structured query interface for user to issue structured query over one or more virtual relations. The structured query will be transformed to SQL-like query over the generic table, and be executed over it.

4. DATA UNITS CATEGORIZER

The data units categorizer is the most important component of our system. It constructs and maintains virtual relations for organizing and presenting data units published to the system of various types through clustering. Consequently, each virtual relation corresponds to a cluster of data units with similar topics and structures. In the next, we can use either cluster or virtual relation for referring to a group of data units resulted in the clustering process. For such a task of clustering data units with heterogeneous schemas, our hypothesis is that data units with similar attributes or

id	title	author	tags	homepage	blog type	language
0	Uzzer's blog	uzzer	Art, blog, comments, design, fun, livejournal, photos, pictures, uzzer, web	http://uzzer.livejournal.com	art-blog	english accepted, russian

(a) Generic table after the first data unit is inserted.

id	title	author	tags	homepage	blog type	language	news source	publish date
0	Uzzer's blog	uzzer	Art, blog, comments, design, fun, livejournal, photos, pictures, uzzer, web	http://uzzer.livejournal.com	art-blog	english accepted, russian	null	null
1	Chinas International services market reached 18 billion in 2005	Analysis International	China, internet Services, News and Articles	null	null	null	analysis international	02/22/2006

(b) Generic table after the second data unit is inserted.

Table 1: Generic table for storing data units.

tags are likely to have similar topics and can be queried together, and we should categorize them into the same virtual relation.

As mentioned, every time a new data unit is accepted, it is assigned to one of the virtual relations, and can immediately be searchable. This calls for an on-line clustering model, which is generalized as the following three steps:

1. Accepts a new data unit u and extracts its attributes and tags, which are the features of u used for clustering.

2. Compare features of u with each virtual relation from the existing set of virtual relations $\mathcal{R} = \{R_1, R_2, \cdots, R_n\}$, and assigns u to the virtual relations that result in a match.

3. If none of the existing virtual relation matches u, create a new virtual relation and assign u to it as the only tuple.

This model allows a tuple to be assigned to multiple virtual relations. This is necessary considering that the topics accumulated in the system over the time will be very diversified, and a data unit is possible to be related to several topics. For example, a blog about travel may be assigned to two clusters that represent *blog* and *travel*, respectively. In addition, there is no predefined parameter for limiting the total number of clusters. This also suits to the free-and-easy and open nature of the service we intend to provide, as the topics in the system are constantly changing and not predictable.

Based on this clustering model, the first important problem is the representation of features of a cluster, in order to effectively compare the matchness of a new data unit with a cluster. This is related to the problem of clustering criteria, and we discuss it in the next subsection. Following it, we consider how to utilize the attribute and tag features of a data unit to match with features of clusters.

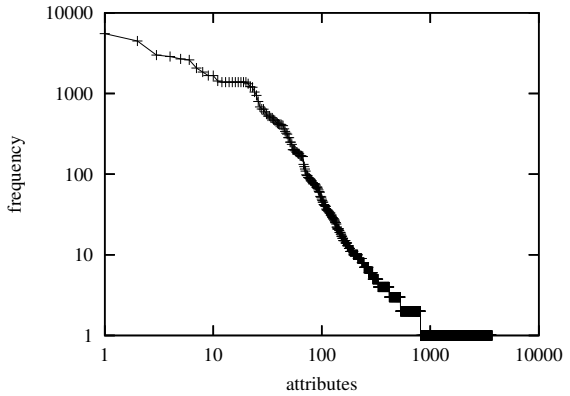
4.1 Clustering criteria

First of all, we need to answer a fundamental problem — what is a qualified virtual relation in our system? Although the ultimate answer is subjective, there is need of some objective criteria for the automatic clustering task.

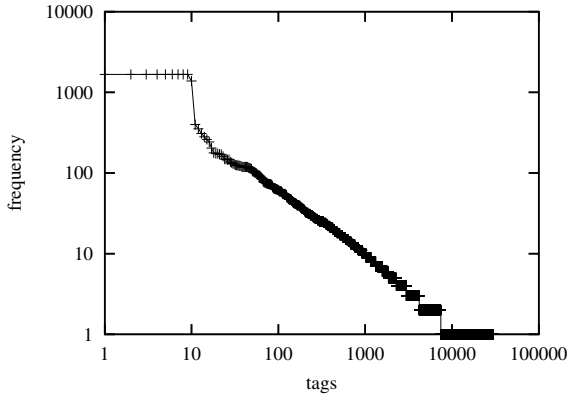
In most clustering problems such as document clustering, the quality of a cluster is measured based on the similarities between the features of the elements in it [15], and the feature of a cluster is calculated by taking median or average of the values of features of its elements. Based on this criteria, it is crucial to select the most important and discriminative features of the data units, and apply appropriate weightings for different features [10].

However, after investigation, we realize that this criteria is not applicable to our problem for two reasons. First, the attributes and tags associated with each data unit contain much "noise" because they are directly created by end users, and it is hard to separate "noise" from useful features given a new data unit due to the one pass nature of the clustering task. Figure 3 shows the frequencies of unique attributes and tags in our collected data units, exhibiting power-law like distribution. We can observe that both curves have very long tails, which correspond to vocabularies with very low frequencies. This tends to cause the measured inter-similarity between data elements inaccurate. Second, since the data units are added to the system continuously that contain uncontrolled vocabularies, the total numbers of attributes and tags keep growing as new data units are published to the system (as illustrated in Figure 4), which makes it difficult to collect corpus level statistics.

Therefore, considering the existence of large volume of

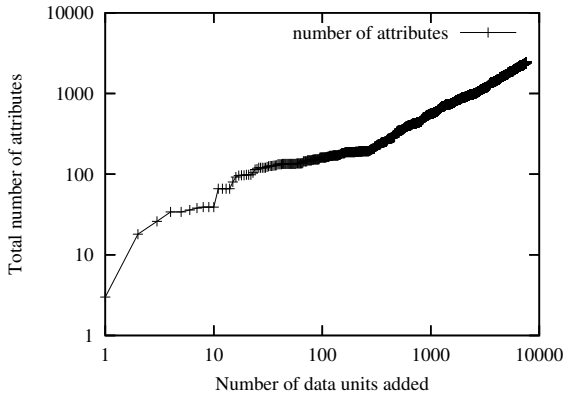


(a) attributes

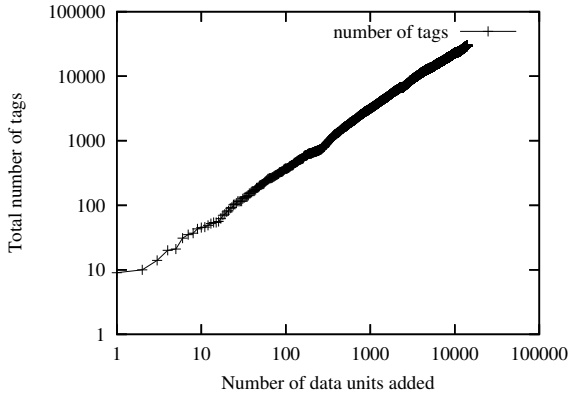


(b) tags

Figure 3: Distribution of attributes and tags.



(a) attributes



(b) tags

Figure 4: Number of unique attributes and tags with the increasing of data units.

infrequent attributes and tags, we choose to measure the goodness of a cluster based on the following two metrics.

1. We test if there are dominating attributes in the virtual relation, i.e., the attributes for which a dominating number of tuples in it have non-*null* values.

2. We test if there are dominating tags, i.e., the tags used by a dominating number of tuples.

We intend to use these two metrics to capture the collective characteristics of tuples in a cluster and avoid the affect of "noise", since we expect it is normal for a virtual relation to have a large number of infrequent attributes or tags. Therefore, we intend to represent the features of each cluster with the dominating attributes and tags, which are also used as the schema and description, respectively, of the virtual relation exposed to users.

4.2 Clustering method

Based on the clustering criteria described above, we propose the algorithm for incrementally clustering incoming tuples into various virtual relations. Since there are two kinds of features — attributes and tags, we propose to first develop algorithms for clustering data units based on attributes (Section 4.2.1) and tags (Section 4.2.2) independently, and later we study how to combine the two kinds of features to generate better clustering (Section 4.2.3).

4.2.1 Attributes-based clustering

We define the *popularity* of an attribute in a virtual relation $R(U)$ with instance $I(R)$ as the ratio between the number of tuples that have non-*null* values for it and the total number of tuples in it, i.e.,

$$popularity(a \in U) = \frac{|\{t[a] \neq null | t \in I\}|}{|\{t \in I\}|}.$$

The *popular attribute set (PAS)* of a cluster R is the set of all attributes whose popularity values are not less than a predefined threshold θ ($0 \leq \theta \leq 1$). Formally, it is represented as

$$\forall a \in PAS. |\{u | u \in I(R) \text{ and } a \in A(u)\}| \geq \theta \cdot |R|, \quad (1)$$

where $A(u)$ is the set of attributes of tuple u.

Example 1 *Suppose we have three data units u_1, u_2, and u_3, where u_1 has attributes $\{brand, condition, manufacturer\}$, u_2 has attributes $\{brand, condition, manufacturer, product\ type\}$, and u_3 has $\{brand, condition, manufacturer, product\ type, color\}$. If $\theta = 1$, the PAS of the group $\{u_1, u_2, u_3\}$ is $\{brand, condition, manufacturer\}$. If $\theta = 0.5$, the PAS will be $\{brand, condition, manufacturer, product\ type\}$.*

As illustrated by the example, we intend to use PAS of a cluster based on properly chosen θ for characterizing the dominating attribute features of a cluster, in order that we

can decide the relevance of a given new tuple to the cluster by only comparing its set of attributes with PAS of the cluster. The attributes of the virtual relation that are not in PAS are considered as "noise", since their popularity is low, hence only reflecting features of individual tuples in the cluster, instead of the collective features of the cluster (e.g., *color* attribute of u_3 in Example 1). Therefore, we require a "good" cluster to have a non-empty PAS with respect to a properly chosen θ, in order to make sure that the tuples in it have similar structures.

On the other hand, in a cluster of closely related data units, each data unit should have sufficient number of attributes from the PAS of the cluster. In Example 1, u_1 have 3 attributes from the PAS (when $\theta = 0.5$), while u_2 and u_3 have 4 of PAS. Consider that if a new unit u_4 has attributes {*condition*, *model*, *vehicle type*}, there is only 1 attribute of u_4 in PAS, which may indicate that u_1 cannot join into the cluster composed by u_1, u_2, and u_3. Given this observation, we realize that we need another threshold α ($0 \leq \alpha \leq 1$) to indicate whether a tuple is relevant to the PAS of a cluster. That is, we require all the tuples in a cluster have more than the portion of α of attributes from the PAS, which is formulated as

$$\forall u \in I(R).|A(u) \cap PAS| \geq \alpha \cdot |PAS|. \quad (2)$$

The parameter α also affects the formation of PAS in a cluster. Extremely, when $\alpha = 1$, PAS is actually the intersection of the attributes of all tuples in a cluster, since it requires every tuple have all the attributes in PAS. In this case, θ has no effect on PAS whatever value it is set to. It is actually a very strong condition, and tends to lead to large number of small clusters. On the other hand, if $\alpha = 0$, there is no requirement on the relevance of tuples' attributes to the PAS, and thus all the tuples could be put in one cluster. Usually, α should be chosen to be greater than 0.5, considering that when $\alpha < 0.5$, two tuples with disjoint attribute sets might be assigned to the same cluster and regarded as similar if they separately match with two disjoint subsets of PAS of the cluster.

Therefore, we use the parameters θ and α together to control the quality of a cluster, and assign incoming tuples to relevant virtual relations.

As we have mentioned a bit previously, θ and α are actually not independent. If we model the virtual relation as a matrix with one dimension representing attributes in PAS and the other representing tuples, where the entries are binary values with 1 indicating the association of the corresponding tuple and attribute and 0 otherwise, i.e., for a cluster R,

$$R = \begin{array}{c} \\ t_1 \\ t_2 \\ \vdots \\ t_{|R|} \end{array} \begin{array}{cccc} a_1 & a_2 & \cdots & a_{|PAS|} \\ 1 & 0 & \cdots & 1 \\ 0 & 1 & \cdots & 1 \\ \vdots & \vdots & & \\ 1 & 1 & \cdots & 0 \end{array}$$

the frequencies of each attribute $a_i \in PAS$ is the number of "1"s in its column, and the number of attributes in PAS associated with each tuple $t_i \in R$ is the number of "1"s in its row. Suppose in i-th column, the ratio of the number of "1"s to the length, $|R|$, is θ_i; and in j-th row, the ratio of the number of "1"s to the length, $|PAS|$, is α_i, then we have

$$\Sigma_i^{|PAS|}\theta_i \cdot |R| = \Sigma_j^{|R|}\alpha_j \cdot |PAS|,$$

where both sides count the number of "1"s in R.

Let $\bar{\theta} = \frac{\Sigma_i^{|PAS|}\theta_i}{|PAS|}$ and $\bar{\alpha} = \frac{\Sigma_j^{|R|}\alpha_j}{|R|}$, we have

$$\bar{\theta} \cdot |PAS| \cdot |R| = \bar{\alpha} \cdot |R| \cdot |PAS|,$$

and therefore we can conclude

$$\bar{\theta} = \bar{\alpha}.$$

We can see that although θ and α regulate the densities of "1"s in two dimensions, respectively, the means of the densities of "1"s along the two dimensions means are always equal.

Next, we examine how to match an incoming data unit with existing clusters in the system, and assign data unit to the clusters that result in a match, given predefined parameters θ and α. Based on our clustering criteria, the matching between a new data unit u and a cluster R is for checking that whether R will be in *good* state, i.e., having non-empty PAS according to Equation 1 and 2, if u were assigned to R. When a virtual relation receives a new tuple u, its PAS might also be changed, with the affect of attributes associated with u. In this way, the PAS of a cluster is dynamically maintained each time a new tuple is added to it.

In addition, we also consider that a new tuple u matching PAS of a cluster based on α and θ might incur too much "noise" if only a small portion from u's own attributes match PAS.

Example 2 *There is a cluster R, with $PAS = \{a, b\}$, and the rest attributes of R are $\{c, d\}$. When there is a new data unit u with attributes $A(u) = \{a, b, e, f, g, h\}$. Since u contains attributes $\{a, b\}$, which matches PAS of R. However, since $\{a, b\}$ is only a small portion (0.33) of u's all attributes, which may indicate that u should not be assigned to R.*

Therefore, we require that for each single tuple, most of its attributes should be in PAS of a cluster when it is assigned to it. However, note that we do not constrain the *ratio of PAS to noise* of a cluster, i.e., a virtual relation can have more "noise" attributes than those in PAS, because we regard the "noise" as individual properties of the tuples.

Figure 5 presents the procedure of matching the attributes of a new data unit u, $A(u)$, with an existing cluster R_i given parameters θ and α. Lines 1-2 are for comparing $A(u)$ with current PAS of R_i, in order to stipulate that u should be relevant to the tuples already in R_i. Lines 3-9 are for generating the new PAS of R_i, PAS_i', assuming u were assigned to R_i, $A(u)$. If PAS' becomes empty, u cannot be assigned to R_i (Lines 10-11). Then we check whether $A(u)$ have more "noise" compared with its intersection with PAS_i' (Lines 12-13). Finally, we check whether R_i will keep in good state by replacing PAS_i with PAS_i' — every tuple in R_i must satisfy PAS' in terms of α (Lines 14-20). Note that we do not need to retrieve all the tuples of R_i for this checking, which will be expensive. Instead, we keep a bit vector for each tuple in R_i representing the set of attributes it has. Therefore, the procedure can be performed very efficiently.

When the procedure $match(A(u), R_i, \theta, \alpha)$ in Figure 5 returns true, the new data unit u will be added into R_i. If a new data unit u cannot match to any of existing clusters, a

match($A(u)$, R_i, θ, α)
1. if $|A(u) \cap PAS_i| < \alpha \cdot |PAS_i|$
2. return false
3. $PAS'_i \leftarrow PAS_i$
4. for each attribute $a \in (PAS'_i - A(u))$
5. if $popularity'(a) < \theta$
6. remove a from PAS'_i
7. for each attribute $a \in (A(u) - PAS'_i)$
8. if $popularity'(a) \geq \theta$
9. add a into PAS'_i
10. if $|PAS'_i| = 0$
11. return false
12. if $|A(u) - PAS'_i| \geq |A(u) \cap PAS'_i|$
13. return false
14. if $PAS'_i = PAS_i$
15. return true
16. if $|A(u) \cap PAS'_i| < \alpha \cdot |PAS'_i|$
17. return false
18. for each tuple $u' \in R_i$
19. if $|A(u') \cap PAS'_i| < \alpha \cdot |PAS'_i|$
20. return false
21. return true

Figure 5: Match a new data unit to a virtual relation based on attributes.

new cluster is created with u as the seed. Thus, we have a simple incremental clustering algorithm based on matching of attribute features of the tuples, as shown in Figure 6. It matches u with each of existing clusters, and assigns u to the clusters that match u (Lines 2-5). If it turns out that u cannot match to any of existing clusters, a new cluster will be created with u as the only tuple, which is shown with lines 6-8.

attrClustering(u, $\mathcal{R} = \{R_1, R_2, \cdots, R_n\}$, θ, α)
1. boolean variable $assigned \leftarrow false$
2. for each existing cluster $R_i \in \{R_1, R_2, \cdots, R_n\}$
3. if $match(A(u), R_i, \theta, \alpha) = true$
4. add u into R_i
5. $assigned \leftarrow$ true
6. if $assigned = false$
7. create a new virtual relation R_{n+1} with u
8. add R_{n+1} to $\mathcal{R}$

Figure 6: One pass clustering algorithm based on attributes.

The one-pass nature of the clustering algorithm shown in Figure 6 causes the problem that the dissimilarity between different clusters is not guaranteed. There could result in multiple clusters with similar structures and heavily overlapping tuples, which may happen depending on the arriving sequence of the data units, since we only consider the local features of each cluster when assigning data units.

We have two solutions for this problem. The first is to constrain each tuple to be assigned to only one cluster. This solution is simple, and can effectively make sure the distinctness of each cluster. With our experiments, this approach leads to slightly more number of clusters than the algorithm in Figure 6. The second solution is using a *clustering monitor* for adjusting existing clusters periodically, in order to detect and combine similar clusters. We can try to combine clusters that have the same PAS or have large portion of overlapping tuples. The overhead of this approach is high, because when combining two clusters, say from A to B, we need to invoke the $match(A(u), R, \theta, \alpha)$ procedure in Figure 5 for every tuple in A to check whether it matches B, and we also need to relocate tuples in A that cannot be assigned to B. This solution is also more difficult in that it need some similarity measure for comparing different clusters. Therefore, we choose the first solution at this stage, and we are still exploring the second approach.

We conducted preliminary experiments for testing the proposed clustering method based on attributes. With an input of around 15K number of data units crawled from Google Base, our clustering method results in 756 number of virtual relations when allowing overlapping between clusters, and 842 when not allowing overlapping between clusters. Although the numbers of clusters are large, most tuples are concentrated on only a small portion of the clusters. By sorting the resulted virtual relations based on the number of tuples they contain in a decreasing order, we measure the *coverage* value of first k number of virtual relations, i.e., the ratio of the total number of distinct tuples covered by them to the total number of input data units. Figure 7 shows the results for both the methods with overlapping and without overlapping. We can see that over 90% data units are in the first 100 clusters, for both methods. When not allowing overlapping between clusters, the tuples are more condensed in fewer relations. This is because there indeed appear quite a number of similar clusters with large overlapping with the method that allows a tuple to go to multiple relations. Table 2 shows the PASs of the top 10 resulted virtual relations with most tuples.

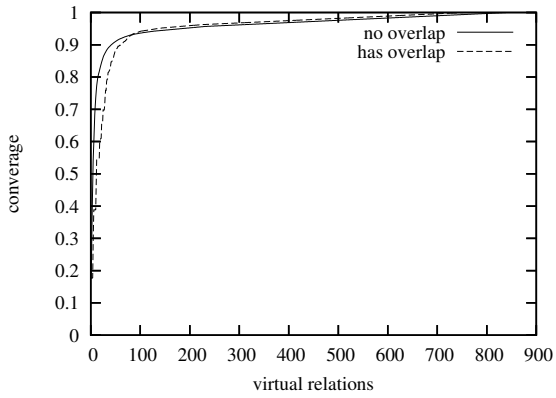


Figure 7: Coverage of virtual relations resulted from attribute based clustering ($\theta = 0.5$, $\alpha = 0.6$).

4.2.2 Tag-based clustering

We can also cluster the data units based on their tag features in a similar way as the method described above, by requiring each of the resulting virtual relations have a *popular tag set (PTS)*.

However, the settings of parameters for tag-based clustering are very different from that of the attribute-based clustering, due to different characteristics of attributes and tags described as follows. First, comparing Figure 3(a) with 3(b), and Figure 4(a) with 4(b), we can see that tags are more di-

No	*PAS*
1	condition(0.98), product type(0.96), brand(0.75)
2	home page(0.99), author(0.94), blog type(0.59)
3	author(0.96), publish date(0.84), news source(0.75)
4	review type(1), name of item review(0.99), url of item review(0.98), rating(0.97)
5	service type(1)
6	job function(0.97), job industry(0.96), employer(0.86), job type(0.77)
7	main ingredient(0.98), cuisine(0.97), cooking time(0.91)
8	gender(0.98), education(0.97), marital status(0.95), age(0.94), sexual orientation(0.88), interest in(0.86)
9	author(1), product type(1), name of item review(1), publication name(1), news source(1), purpose(1), patent prior art report(1), link to patent full text(1), blog for your comment(0.64)
10	author(1)

Table 2: The *PAS* of top 10 virtual relations with most tuples. (The numbers in the brackets are the popularity values of the attributes.)

verse than attributes, which results in larger set of vocabularies. For example, a PC could be labeled with many different tags, including "computer", "desktop", "PC", "IBM", "HP", "Dell", "sale", "product", etc.., depending on user's habit or preference. However, the attributes about a PC are relatively more uniform, typically listed as "CPU", "RAM", "hard disk", "price", etc.. The second difference between attributes and tags is that the tags associated with a data unit tend to be more independent to each other than the attributes, in terms of being features for the clustering task. This is because usually the different tags of a data unit characterize it from different angles, i.e., each tag corresponds to a "concept category", while the attributes of a data unit collectively describe its various properties. We can see it from the PC example described above, as well as from the two example data units shown in Figure 1.

The diversity and independence characteristics of tags determine that each cluster can only have few tags in PTS, which should be very similar to each other, and a data unit matches to a cluster based on tags only if it has a tag in the PTS, without considering the affect of its rest set of tags. This suggests that θ should be set to 1, in order that the tuples in a cluster have the same set of dominating tags, which represent a particular subject area. In this case, α has no effect on the clustering whatever it is. Note that we allow overlapping of data units in the clusters generated based on tags, since data units are likely to be related to multiple topics.

With the same set of input data units, the tag-based clustering generates 3488 number of clusters. This is much more than that of attribute based clustering, since the tags are more diversified. Similarly, we measure the coverage of the virtual relations resulted from the tag-based clustering. The results are shown in Figure 8. Compared with attributed based clustering, we can see that the tuples are distributed more evenly in the clusters resulted from tag based clustering. Actually, the PTS of each of the resulted clusters has only a single tag, which is shared among all the tuples in it. Figure 9 shows such tags of the top 20 most popular clusters. We can see that due to the diversity of the usage of tags, there are similar clusters labeled with different tags, such as "hotel" and "accommodation", "sales job" and "job", etc..

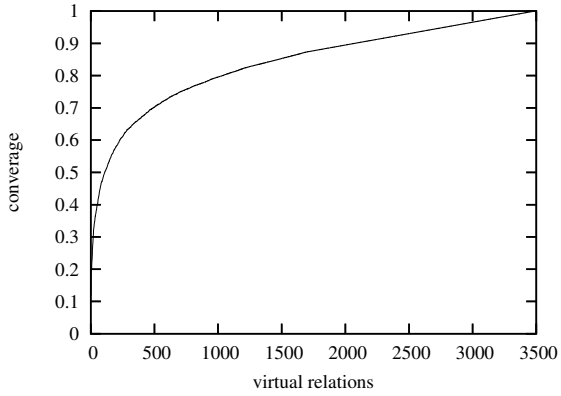


Figure 8: Coverage of virtual relations resulted from tag based clustering ($\theta = 1$).

4.2.3 *Combining attribute-based and tag-based clustering*

According to previous experiments, we realize the advantages and disadvantages of both attribute based clustering and tag based clustering. Specifically, attribute based clustering group data units with similar schemas, but the topics of the tuples in a cluster might not be closely related. For example, looking at the PAS of the No. 1 virtual relation resulted from attribute based clustering shown in Table 2, it represents a cluster of data units about products. However, the topics of the data units it contains are very diversified, including *motorcycle*, *jewelry*, *bag*, *home decor*, *art*, etc.. On the other hand, the clusters generated based on tags usually concentrate on very specific topics, but there often have different clusters with very similar topics, or with subordinate (superordinate) relationships, which causes the number of clusters very large.

We propose a two-phase clustering strategy to combine the two features, attributes and tags, in order to better organize the data units, which are described as follows.

1. Group tuples coarsely based on attributes at the first phase.
2. Divide tuples within each cluster resulted in phase 1

patent(1671), hotel(396), recipe(353), blog(309), travel(243), accommodation(202), seafood(174), all digital camera(174), music(173), google(169), computer(154), sales job(148), technology(146), dating(144), real estate(141), news(132), internet(131), business(125), podcast(125), job(124)

Figure 9: The top 20 most popular tags resulted from tag-based clustering. (The numbers in the brackets are the frequency of the tags.)

No	*PTS*	*PAS*
1.1	honda motorcycle	brand, condition, product type, size, speed rating
1.2	jewelry	product type, condition, brand, website, manufacturer
1.3	dico turf tamer at john deere gator	condition, product type, brand, size
1.4	bag	condition, product type, brand, size
1.5	rear radial shinko super sport tour	brand, condition, product type, size, speed rating
1.6	home decor	product type, condition, brand
1.7	art	product type, condition, brand, merchant
1.8	silver	condition, brand, product type
1.9	discount nascar jacket	apparel type, brand, condition
1.10	necklace	condition, product type, brand

Table 3: The *PTS* and *PAS* of top 10 virtual relations with most tuples divided from the No. 1 phase-1 cluster in Table 2.

into specific categories based on their tags at the second phase, in order that the resulted clusters can be represented with both a set of popular attributes and a set of popular tags.

When applying the above 2-phase clustering strategy for categorizing each incoming data units, a large number of very fine grained clusters are generated in total. With the same data set, we got more than 5000 such clusters. Considering that it is not convenient for users to browse the overwhelming number of clusters, we choose to first categorize arriving data units into coarse clusters in phase 1, which are represented as virtual relations to be browsed by users. Upon the request of users, we dynamically divide a virtual relation in phase 1 into multiple clusters with different specific topics based on tags, such that users can zoom in their queries to virtual relations with specific topics, and they can also query over more specific attributes of the phase-2 virtual relation. Table 3 shows the PTS and PAS of the top 10 most popular virtual relations resulted by further clustering the tuples in the No. 1 virtual relation in Table 2. We can see that the popular attributes of different phase-2 virtual relations are not uniform, since they describe more specific features.

4.3 Discussion

Generally, it is difficult to the measure the quality of clustering in terms of precision and recall as the traditional way in such a free-and-easy publishing system, because it is hard to get the "correct" clustering for the large amount of very irregular data units. We have tried to manually categorize a few of the data units, and we found it is often a very subjective choice for selecting the right cluster to assign a data unit, and actually, the clustering result do not show much advantages over that that of automatically clustering based on attributes and tags.

In addition, in our current implementation, we treat different attributes and tags as independent features. In fact, there are often correlations between them, such as synonym, subordinate, superordinate, etc.. It is a hard problem by itself to discover such correlations, as discussed in [13].

5. STORAGE MANAGER

The generic table for storing all the tuples can contain several to tens of thousands of attributes, and the tuples in it are very sparse — they have *null* values in most of the attributes. This poses challenge to the storage scheme for the generic table. To handle dynamism of data composites and huge amount of data, we need a method that is scalable over the number of attributes and data volume, in terms of both storage space cost and querying and updating efficiency.

At this stage, we propose to store the tuples in compact form with pairs of attributes and non-*null* values, and build an inverted index over the attributes of the generic table.The index includes a list of all the attributes in the generic table, where each attribute points to an inverted list containing the identifications of tuples that have non-*null* values for the attribute. Figure 10 illustrates the structure of this storage and index scheme for the generic table.

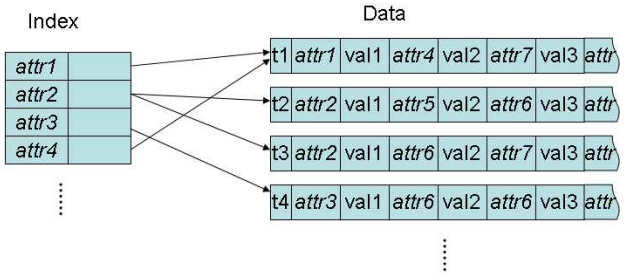


Figure 10: Storage and Index scheme for the generic table.

The inverted style index is well studied in IR literature for indexing keywords in the text documents [24]. We utilize it for efficiently selecting tuples given attribute-based queries. It is also convenient for inserting new attributes to the table when new data units are added.

We conducted preliminary experiments with more than 230K number of data units crawled from Google Base. The codes were implemented in Java, and executed on an AMD Athlon XP 2600+ computer with 1GB of RAM running Windows 2003 Server. We recorded the elapsed time for inserting each 10000 attributes. Figure 11 shows the result. We can observe that with the increase of attributes, the elapsed time also increases, since more number of index pages need to be retrieved for insertion of attribute entry. However, the elapsed time is proportional to the number of attributes. Since every tuple typically contains several to dozens of attributes, it is efficient for inserting new tuples to the table with the storage scheme.

Next, we evaluated the efficiency of processing attribute-based queries with the index structure. We varied the numbers of predicates (NOP), i.e., the numbers of distinct attributes referenced in the queries, from 1 to 4, and recorded the corresponding elapsed query processing time. The results are shown in Figure 12. We can see that the query

processing time grows linear with the increase of the number of attributes. On the other hand, when the number of predicates increases, the query processing time also increases, since we need to access the index for more number of times. We can see that generally, the processing time is proportional to the number of of predicates.

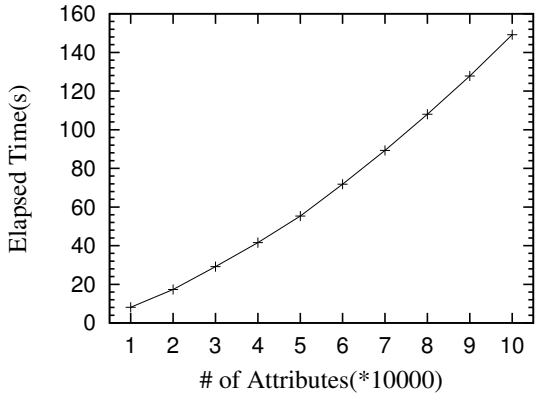


Figure 11: Accumulated insertion time for every 10000 attributes.

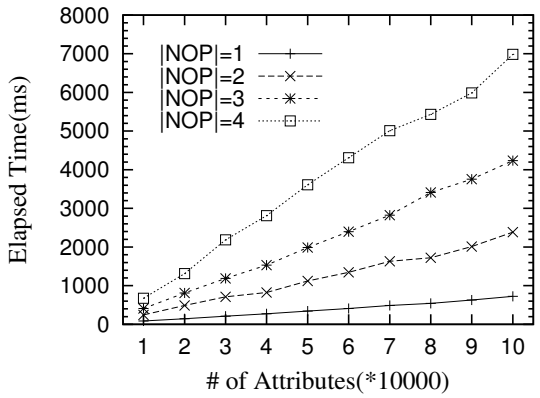


Figure 12: Query processing time for attribute-based queries.

The storage manager is also in charge of maintaining the mapping between each virtual relation R_i and $AllUnits$, which is a query over the instance I of $AllUnits$, i.e., $R_i \mapsto AllUnits = q_I^i$. As data items are assigned to R_i, q_I^i need be modified as well.

When the mapping between the attributes of R_i to the attributes of $AllUnits$ is one-to-one, q_I^i is a simple projection over a subset of attributes of $AllUnits$ that matched with the attributes of R_i, respectively. There could also be the case that an attribute of R_i is mapped to multiple attributes of $AllUnits$ when the multiple attributes are detected as synonyms. In this case, q_I^i is a union over multiple projections over different combinations of the attributes of $AllUnits$ according to the attribute mappings. At current stage, we only consider simple one-to-one attribute mappings. We will investigate the issues with one-to-many mappings at a later time.

6. BROWSING AND QUERY PROCESSING

Users look for interesting information by browsing the virtual relations, which are presented in a decreasing order according to the number of tuples they contain, in order that users can first see the most popular information.

Each virtual relation is represented with its schema — the set of attributes, and description — the set of tags, which are both ordered decreasingly according to their popularity. Some folksonomy-based web services visualize tags with different size reflecting their popularity, e.g., Delicious [1] and Flickr [2], called *tag cloud*. This can also be applied to our system.

The query processor supports both simple keyword queries and structured queries posed over the virtual relations. A keyword query $Q = (k_1, k_2, \cdots, k_q)$ can be posed to a particular virtual relation, or to the whole system. A fulltext inverted index is maintained for associating each of the keywords with the name of the attribute, the id of the tuple, and the virtual relation it appears in. A keyword query is evaluated by looking up the index and locating the positions of the relevant tuples in the generic table. An inverted index over the tags is also built in order to support queries for tuples with particular tags. Structured queries are processed by reformulating the queries posed over the virtual relations to that of the generic table according to the mappings between them, and executed over it.

7. RELATED WORK

We present the related work to our proposed system in this section.

7.1 Data clustering

Data clustering is useful for analysis and classification in various applications, and there is a large research literature for addressing various clustering problems [14, 15]. The most related to our problem is the incremental clustering model presented in [11], in which a set of k clusters is dynamically maintained as new data points are added in sequence. Another category related to ours is the on-line event detection problem for identifying stories in one or more news streams [7, 23]. When a new story arrives, it is either recognized as belong to an existing event, or identified as a new event, where an event is basically a group of stories of related topics. This problem also requires one-pass clustering of the stories. Compared to them, the difficulty of our problem lies in the unpredictability of both the number of clusters and the important features need to be extracted from each of the data points, due to the open nature of our system.

[9] presents a technique for clustering tags to improve user experience. It considers co-occurrences among tags for discovering semantic relationships between tags, in order that users can follow semantic related tags during browsing.

7.2 Topic taxonomy

Many web contents are organized with a taxonomy of topics, e.g., Yahoo![1] and Open Directory[2], for facilitating information access and searching. Such taxonomy need to be predefined by experts, and the documents can be classified into the taxonomy either manually, or with supervised learning techniques with the help of certain classifiers, such as C4.5

[1]http://www.yahoo.com
[2]http://dmoz.org/

[20], and neural networks [16]. Although this approach can render better organization of the documents with systematic topic hierarchies, the main concern is that it is harder to maintain and hence not scalable to the exponentially increasing of digital information on the Web.

7.3 Folksonomy

In contrast to organization of topics with the systematic taxonomy, folksonomy is an emerging on-line data sharing paradigm in a free-and-easy style, with the cooperation of all the end users [5, 13, 18]. The web services based on it are getting increasingly popularity [1, 2, 3, 4]. A distinguishing feature of folksonomy is that the categories are self-organizing, with the metadata provided by end users. Different from traditional taxonomy that is a hierarchy of concepts, folksonomy results in a flat organization of categories of various information units with fuzzy boundaries. In another word, the folksonomy approach trades precision a bit with increased efficiency and scalability for adapt to increasing and dynamic data over the Web.

Currently, there are emerging works for mining the tagging information in folksonomy-based web services for various purposes, such as automatically suggesting tags to users [22], and inducing ontology from tags [21].

7.4 Storage and querying of sparse data

There have been research works for supporting the storage of sparse data, which are often used in e-commerce applications and medical information systems, in RDBMS [6, 8]. [6] uses 3-ary vertical scheme for storing the original sparse tuple. The pairs of attributes and non-*null* values of the sparse tuple are stored in the vertical table associated with the id of the sparse tuple. With this approach, H2V (horizontal to virtual) and V2H (virtual to horizontal) need to be performed when processing queries and returning results. [8] introduces an interpreted storage scheme, in which tuples are stored as a list of non-*null* values associated with their attribute identifiers. The system catalog records all the attributes and their associated ids. Our storage of the generic tuples is similar this interpreted scheme.

8. CONCLUSION AND FUTURE WORK

We have presented our design and implementation for a folksonomy based free-and-easy data publishing and sharing system. We devise a data model for representing and storing the data units. We also present our approach for each system component and discuss the research challenges.

Currently, our focus is on meaningfully clustering and effective query processing on the data units. Our ongoing work includes designing of efficient indexes on the virtual relations and data units to facilitate efficient retrieval, and efficient and adaptive query processing strategies since the less constraints imposed on the storage increased the complexity of query processing.

One of the opportunities we see in such generic storage structure is data sharing over P2P system. We intend to adopt it onto our BestPeer [19] P2P platform after we have resolved various technical issues on a centralized server.

9. REFERENCES

[1] Delicious website. http:http://del.icio.us/.

[2] Flickr website. http://www.flickr.com/.

[3] Google base website. http://base.google.com/.

[4] Liveplasma website. http://www.liveplasma.com/.

[5] M. Adam. Folksonomies - cooperative classification and communication through shared metadata, December 2004. http://www.adammathes.com/academic/computer-mediated-communication/folksonomies.html.

[6] R. Agrawal, A. Somani, and Y. Xu. Storage and querying of e-commerce data. In *VLDB '01: Proceedings of the 27th International Conference on Very Large Data Bases*, 2001.

[7] J. Allan, J. Carbonell, G. Doddington, J. Yamron, and Y. Yang. Topic detection and tracking pilot study: Final report. In *DARPA Broadcast News Transcription and Understanding Workshop*, 1998.

[8] J. L. Beckmann, A. Halverson, R. Krishnamurthy, and J. F. Naughton. Extending RDBMSs to support sparse datasets using an interpreted attribute storage format. In *ICDE*, 2006.

[9] G. Begelman, P. Keller, and F. Smadja. Automated tag clustering: Improving search and exploration in the tag space. In *WWW 2006 Collaborative Web Tagging Workshop*, 2006.

[10] J. Callan. Document filtering with inference networks. In *SIGIR*, 1996.

[11] M. Charikar, C. Chekuri, T. Feder, and R. Motwani. Incremental clustering and dynamic information retrieval. In *Symposium on Theory of Computing*, 1997.

[12] R. Fagin, A. O. Mendelzon, and J. D. Ullman. A simplied universal relation assumption and its properties. *ACM Trans. Database Syst.*, 7(3), 1982.

[13] S. Golder and B. A. Huberman. The structure of collaborative tagging systems. Technical report, Information Dynamics Lab, HP Labs, 2005.

[14] A. K. Jain and R. C. Dubes. *Algorithms for clustering data*. Prentice-Hall, Inc., 1988.

[15] A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: a review. *ACM Computing Surveys*, 31(3):264–323, 1999.

[16] C. G. Looney. *Pattern recognition using neural networks: theory and algorithms for engineers and scientists*. Oxford University Press, Inc., 1997.

[17] D. Maier, J. D. Ullman, and M. Y. Vardi. On the foundations of the universal relation model. *ACM Trans. Database Syst.*, 9(2), 1984.

[18] D. Millen, J. Feinberg, and B. Kerr. Social bookmarking in the enterprise. *ACM Queue*, 3(9):28, 2005.

[19] W. S. Ng, B. C. Ooi, and K. L. Tan. BestPeer: A self-configurable peer-to-peer system. In *ICDE*, 2002. Poster Paper.

[20] J. R. Quinlan. *C4.5: programs for machine learning*. Morgan Kaufmann Publishers Inc., 1993.

[21] P. Schmitz. Inducing ontology from flickr tags. In *WWW 2006 Collaborative Web Tagging Workshop*, 2006.

[22] Z. Xu, Y. Fu, J. Mao, and D. Su. Towards the semantic web: Collaborative tag suggestions. In *WWW 2006 Collaborative Web Tagging Workshop*, 2006.

[23] Y. Yang, T. Pierce, and J. Carbonell. A study of retrospective and on-line event detection. In *SIGIR '98: Proceedings of the 21st annual international ACM SIGIR conference on Research and development in information retrieval*, 1998.

[24] J. Zobel, A. Moffat, and K. Ramamohanarao. Inverted files versus signature files for text indexing. *ACM Trans. Database Syst.*, 23(4), 1998.