ERCIS

European

Research

Center for

Information

Systems

Working Paper No. 8

Hagemann, S.
Vossen, G.

Web-Wide Application Customization: The Case of Mashups

ERCIS – European Research Center for Information Systems

Editors J. Becker, K. Backhaus, H. L. Grob, B. Hellingrath, T. Hoeren, S. Klein, H. Kuchen, U. Müller-Funk, U. W. Thonemann, G. Vossen

Westfälische Wilhelms-Universität Münster

Working Papers

ERCIS — European Research Center for Information Systems
Editors: J. Becker, K. Backhaus, H. L. Grob, B. Hellingrath, T. Hoeren, S. Klein, H. Kuchen, U. Müller-Funk, U. W. Thonemann, G. Vossen

Working Paper No. 8

Web-Wide Application Customization: The Case of Mashups

Stephan Hagemann, Gottfried Vossen

ISSN 1614-7448

cite as: Stephan Hagemann, Gottfried Vossen: Web-Wide Application Customization: The Case of Mashups. In: Working Papers, European Research Center for Information Systems No. 8. Eds.: Becker, J. et al. Münster 2010.

Working Paper Sketch

Type

Research Report

Title

Web-Wide Application Customization:
The Case of Mashups.

Authors

Stephan Hagemann and Gottfried Vossen

{hagemann | vossen }@ercis.uni-muenster.de

Abstract

Application development of is commonly a balancing of interests, as the question of what should actually be implemented is answered differently by different stakeholders. This paper considers *mashups*, which are a way of allowing an application to grow beyond the capabilities of the original developers. First, it introduces several approaches to integrate mashups into the services, or Web pages, that they are based upon. These approaches commonly implement ways to determine which mashups are potentially relevant for display in a certain Web page context. One approach, *ActiveTags*, enables users to create reliable mashups based on tags, which effectively, leads to customized views of Web pages with tagged content. A scenario that demonstrates the potential benefits of this approach is presented. Second, a formalization of the approaches is presented which uses a relational analog to show their commonalities. The abstraction from implementation specifics opens the range of vision for fundamental capabilities and gives a clear picture of future work.

Keywords

Application customization; mashups; social tagging; meta-programming.

Contents

List of Figures

List of Tables

1 Introduction

Web applications ("Web apps") have been developed very rapidly over the past years. Their development is different from the development of "classic" applications in that Web apps are perceived and delivered as services, which "impacts the entire software development and delivery process" (Musser and O'Reilly, 2007). What is immanent to both kinds of applications, traditional and Web, is that there can never be a full accordance between providers, users, or other stakeholders as to which features a certain application should have. The utility of new features and functions differs between the stakeholders, and there will always be functions that will not be implemented by the provider (although they are requested by users). With their ever growing importance, the need for customizability henceforth also increases. Moreover, as Macías and Paternò (2008) note, "there is an ongoing shift to end-user centered technology, and even users with poor or no skill in Web-based languages may feel the need to customize Web applications according to their preferences." The present paper studies an approach to Web application customization that fits particularly well with frameworks whose purpose is to integrate mashups into Web apps.

The problem of application customization is particularly interesting for the context of Web applications. Karger et al. (2009); Lutteroth and Weber (2008); Macías and Paternò (2008) each deal with different aspects of this. A key observation of Karger et al. (2009) is that "instead of warping their data to fit rigid applications, users should warp applications to fit their data and tasks." The present paper takes a complimentary approach by suggesting practical Web app enhancement through mashups. In the first part of this paper, we present the ActiveTags prototype originally outlined in Hagemann and Vossen (2009), which enables this based on tag-based mashups that are integrated into Web pages on the fly via a browser extension. In the second part, the generality of the approach is documented by a transformation of the application into a relational equivalent.

Attempts to cope with the challenge of application customization from a technical point of view are Application Programming Interfaces (APIs), which allow skilled developers or users to create custom functionalities based on the data and functions provided by Web applications. The number of such APIs has been rising continuously over the last years. This allows for what Hippel (2005) has called "democratizing innovation." Well-designed APIs turn Web sites into platforms that make themselves indispensable by providing the grounds for "ecosystems" of applications (cf. Musser and O'Reilly, 2007).

One form of API usage is mashup creation, i.e., Web applications which combine data or functions from one or more sources into one interface. Because of the perceived tentative character, mashups were first considered to be merely toys or gimmicks, but gradually they are receiving more recognition from both research and industry. The utilization of mashups in enterprises has been hampered by some important barriers which prevent widespread adoption (cf. Hinchcliffe, 2007). However, many companies, large and small, have started to turn to mashups as a useful option to quickly attain some needed functionality. [1] Offering APIs enables third parties to develop functionality externally. However, the prerequisites are still high, as they require programming skills, knowledge of the particular API, and the appropriate infrastructure to run on.

[1] In fact, the financial crisis that took off in late 2008 has strengthened the role of mashups in companies seeking cost savings and improved operations (cf. Rodier, 2009).

Mashups typically do not, however, integrate with the Web page they are based upon. So, although they may be adding relevant functionality to an application, they are still perceived as separate from their subjacent services. While this may be beneficial in many use cases, it vastly prevents them from being used for application customization.

This paper therefore takes a look at mashup frameworks that can be used to integrate mashups into Web applications, thereby allowing these to be extended in any desired way. The running example will be ActiveTags, which is the prototype of a Firefox extension (Hagemann and Vossen, 2009) developed as part of Hagemann (2009). It has the additional benefit of allowing Web-wide customizations. In order to do so, *tags* are used as a layer separating Web applications and mashups. Its approach takes advantage of the fact that there are quite a number of sites that provide a platform for communication and interaction, but little or no original content. Such sites intrinsically function as platforms for User-Generated Content (UGC): users explicitly and implicitly create data through their usage of a site. One form of UGC that has found particular interest in the research community is *tagging* (cf. Voss, 2007), i.e., manual, free-for-all keyword annotation. Typically created in social tagging systems, tags come with a dual role as personal and social metadata (cf. Ames and Naaman, 2007). While tags do not constitute the most reliable form of metadata, their flexibility and ease of use has made them popular among sites employing UGC. Despite being uncontrolled, tags have evolved beyond their support of search and browsing.

We go one step further in this paper and not only look at the technical side of mashups and their capabilities w.r.t. the creation of novel types of Web apps; beyond this, we are also interested in a sound theoretical foundation. To this end, we have observed an analogy between Web page requests and method execution, particularly immanent in our ActiveTags concept. This analogy is amenable to meta-querying, which has previously been studied in the context of relational databases. We adopt meta-querying for describing client-side mashups in a concise manner, and are aeven able to demonstrate the universality of this approach, which also subsumes other approaches to the creation of mashups. As an aside, this provides further evidence of the descriptive power of the relational database model.

The remainder of this paper is organized as follows: Section 2 introduces the ActiveTags prototype and details how it supports Web-wide application customization. Applications of similar capabilities as ActiveTags are presented in Section 3. These two sections form the first and the practical part of this paper. The second part, Section 4, deals with the formalization of (a generalized version of) these approaches which is based on a relational representation of the entire setting. In this setting, Document and Anchor relations, which have previously been introduced in the context of WebSQL (Arocena et al. (1997)), are enhanced by Extractor and Mashup relations, and it is shown how to specify mashup creation via suitably designed functions as well as meta-queries. The paper is concluded in Section 5.

2 ActiveTags

In this section we describe the main features as well as the design of the ActiveTags system and illustrate its usage in the construction of a personal learning environment.

2.1 ActiveTags as a Sample of Web-Wide Application Customization

Tagging is the common term for personal and free labeling (the *tags*) of information objects (or *items*). The result of social tagging is often called a folksonomy. Many variants of social tagging have been implemented and analyzed. A major distinction is into bag vs. set-based tagging systems, which were first discussed under the names "broad" and "narrow" in Vander Wal (2005). Analyses since then have focused on the introduction and conceptualizations of variants (e.g. Hotho et al., 2006; Lambiotte and Ausloos, 2006; Lee et al., 2007; Marlow et al., 2006; Voss, 2007). What is common to all these works is that an agent (the tagger) performs the act of tagging by attaching a label (the tag) to a resource. Of these three, at least the latter two (tag and resource) are always presented together. To the users of a tagging system, tags are more general than applications: Indeed, Thom-Santelli et al. (2008) have found that taggers try to tag consistently across distinct tagging systems.

The characteristics of tagging are at the heart of the ActiveTags implementation, which we capture in the four notions of tagging's (1) free nature, (2) flexibility, (3) scope, and (4) user-centricity. ActiveTags has been designed to keep and support these notions:

- ActiveTags allows multiple tag interpretations to coexist, not enforcing one particular interpretation.
- Tag-based mashups can be added to the running system, which allows interpretations and new uses to arise and evolve over time.
- Tags can be treated uniformly across the Web. Tags from one system can be, but do not have to be treated differently.
- It is the users who control the interpretation and extend the system with new tag-based mashups.

The question that may still be unclear is how exactly mashups *can* be based on tags. In short, tags are used as the triggers and parameters of mashup execution. Mashups are defined in such a way that the occurrence of certain tags triggers the calling of the mashup, while the content of other tags is handed as parameters to the mashups.

One form of tags that is of particular interest for this is machine tags. Such tags use separators to distinguish between their different parts. So-called triple tags (Catt, 2006) are divided into three parts usually written in the form `namespace:property=value`. These tags are meant to enable users to store additional information in a structured way for which there are no predefined structured fields. Flickr, for example, supports triple tags by giving them special treatment in their Graphical User Interface (GUI) and API, calling them machine tags (cf. Straup Cope, 2007). Two examples of support for specific tags are present on Flickr, a photo sharing site which allows the owners of photos to tag pictures, and to share them with friends depending on settings. So-called *machine tags* were introduced in January 2007, see Straup Cope (2007), which marked the beginning of special treatment of tags of the form "`namespace:predicate=value`." More functions are supported for machine tags when the Flickr API is used (`flickr.com/services/api/flickr.photos.search.html`). Earlier, the site had begun to support two sets of specific tags: geotags and event tags: (1) Geotags are a combination of three tags that together indicate the presence of a geotag and encode a geographic location. Two of the tags are machine tags, which encode the longitude and

the latitude and can thus be used to position objects on a map. The site allows these two tags to be exported into structured data fields for geographic information, so that the pictures are automatically shown on maps as well. (2) Tags with the prefix "`upcoming:event=`" are interpreted as links to events on `Upcoming.org`, with the effect that a link to the corresponding event is included in the page.

As will be shown below, machine tags can be used to control the execution of mashups precisely. Note that non-machine tags may also be suitable for this purpose if properly disambiguated (cf. Hagemann, 2009).

A user survey has been conducted in Hagemann (2009), which has shown that there is a considerable interest in the possibilities of ActiveTags, at least among technically versed users. We believe this is due to the simplicity of the approach, which does not interfere with the original purpose of Web applications, but can nevertheless extend and customize them in meaningful ways.

2.2 Design

Figure 1 shows an overview of the main components of the ActiveTags system. Depicted at the top is the user-side component: a browser with the ActiveTags extension. The ActiveTags server (depicted on the left) stores global databases of definitions. The mashup or API providers are part the Web and are not considered a part of ActiveTags.

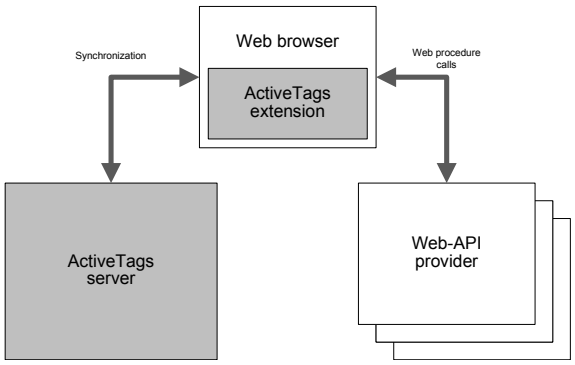

Figure 1: Overview of ActiveTags components.

In Figure 2 the high-level processes of ActiveTags are shown. The diagram abstracts from several aspects: There are detailed functions only when needed to complete the diagram. Data stores are only shown at an aggregated level. Transport and communication are not shown, neither are login and logout functions. Instead, processes are modeled to run indefinitely.

Two roles are shown in Figure 2, which form two separate processes: administrators and users. While the processes of users span all components of ActiveTags, administrator processes are limited to the ActiveTags server component. The administrator's role is the management of public definitions. To this end, the administrator performs his management function directly on the ActiveTags server, depicted in the top right of Figure 2.

The user process can be subdivided into two subtasks. On the one hand, there is the management of local definitions and feedback, which can also be subsumed as the user's administrative functions. On the other hand, it lays the execution of mashups. The latter function is controlled indirectly by the user, as it operates whenever Web pages are browsed and loaded. It is this function that initiates the communication with various providers to augment Web pages with a mashup when appropriate. Both functions are performed through the *ActiveTags extension*, use all the local definitions, and are depicted in the center of Figure 2.

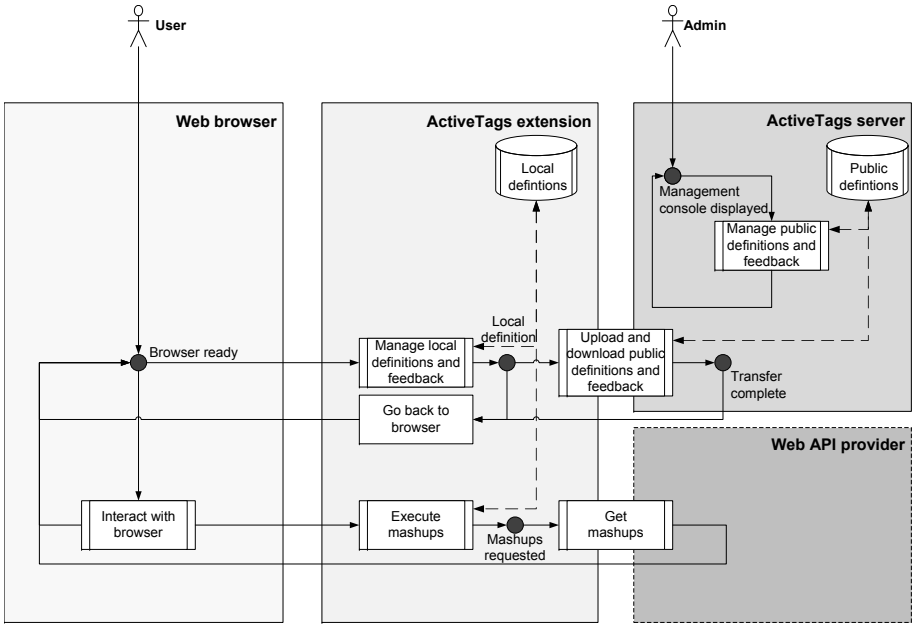

Figure 2: High-level processes of ActiveTags.

The central user process is the one depicted at the bottom of Figure 2 and is comprised of the functions "execute mashups" and "get mashups." It stands out from the other processes as it is the only one in which mashup providers are included in the communication and as it is not initiated through the interface of the ActiveTags extension, but through the browsing of the user in an ordinary session with the Web browser.

Next, the process implementing the execution of mashups is depicted in Figure 3. Two paths are shown in this diagram, one of which connects the ActiveTags extension to the mashup provider, the other is one that produces a MergeSpace.

The first path starts with the Web page being analyzed by the extension in the function "extract tags." This function uses TagExtractor definitions to determine which tags are present on the current Web page. TagExtractor definitions do not necessarily have to be applicable on a Web page, and those that are may return empty results if no tags have been entered for the object represented on that page. If no tags are found, the entire process is aborted (which is not shown). If tags are found, the next function in this path can determine relevant mashups by using the Uniform Resource Locator (URL) of the current page and the extracted tags to determine whether a mashup definition is active that demands execution under the given circumstances. Again, if none are found the entire process can be aborted. If mashups are found, their call parameters are determined on the basis of the tags provided. The tags are

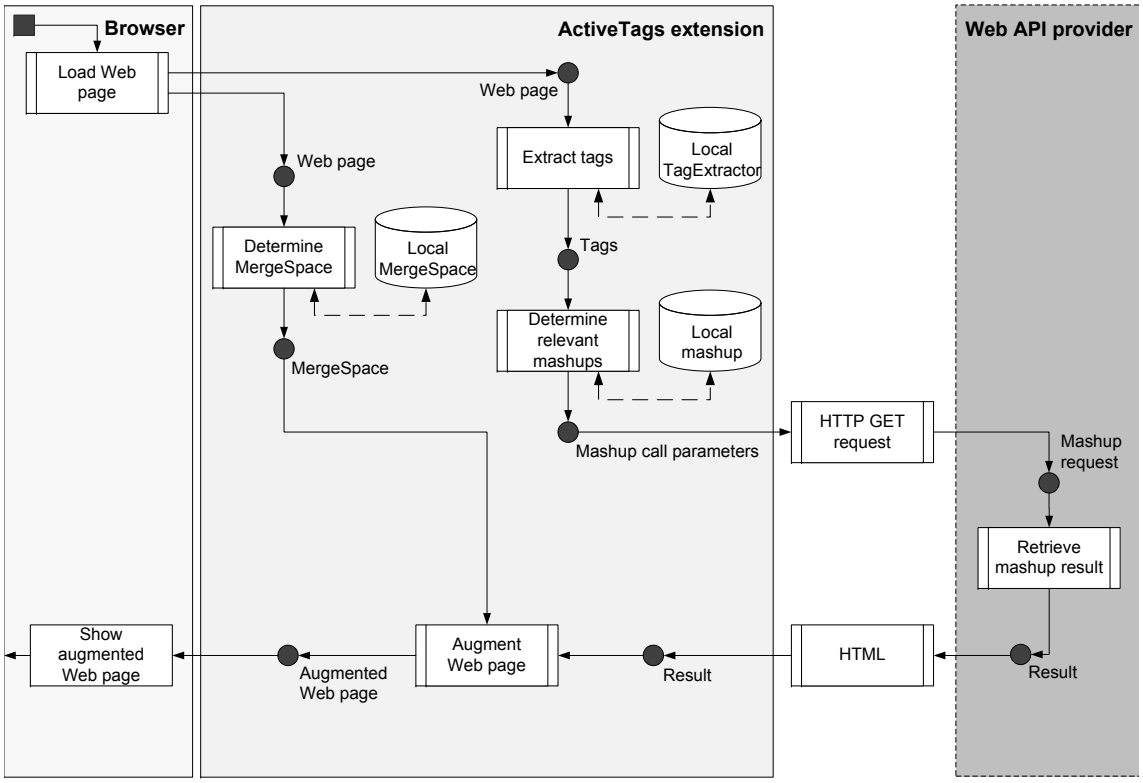

Figure 3: Execution of mashups in ActiveTags.

essentially used to create the URL used in the calling of the Web Procedure Call (WPC) server which provides the mashup. The following steps are performed for every mashup that fulfills the conditions: A Hypertext Transfer Protocol (HTTP) GET request is sent to the named URL, which triggers the execution of the mashup at the mashup provider. The result is sent back in Hypertext Markup Language (HTML) over HTTP and can now be used to augment the original Web page. There is one result for every mashup executed, and all of these are added to the Web page.

Before the Web page can be augmented, another mandatory function has to be performed: the determination of the MergeSpace. This space, which is either typically at the end of a Web page or somewhere among the content of the page as defined by a MergeSpace definition, determines where mashups are to be added into the page. If, for some reason, no MergeSpace can be determined, the entire process is aborted since there is no place to show the results of mashups.

With these two prerequisites, mashup results and MergeSpace, ready, the Web page can finally be augmented. The mashups are shown in their defined place and the loading of the augmented Web page is complete. Note that since this process is part of the Web page loading procedure performed by the browser, it can be aborted at any time during its execution, which happens, for example, when the user clicks on a link to load a new page.

The *ActiveTags server* stores the global database of definitions. Through this database the extension instances exchange their definitions. The extension regularly synchronizes its database

of definitions with the server to support sharing among users. This encompasses TagExtractors, MergeSpaces, and mashups.

2.3 A Use Case: Personal Learning Environments

The idea of employing service-orientation is still at the core of current developments in eLearning. One proposed concept are so-called PLEs: sets of tools which better support an entire learning process and tend closer to a learner's needs. According to Chatti et al. (2007), PLEs introduce two new ideas to eLearning, namely a decentralization of learning systems through the decoupling of learning from institutions, and a redirection of the focus from the content towards the learner. The rationale for PLEs is that while people move through different institutions over time (e.g., college, university, and corporations), their need for continuous learning remains. The challenge is that a PLE is inherently "personal" and either needs to be built individually or customized. As Severance et al. (2008) note, "monolithic VLEs are too hard to customize at the individual user level, and evolve far too slowly to meet teaching and learning of users who want their teaching and learning environments to be under their personal control." This problem with the adaptation of Virtual Learning Environments (VLEs) has led to proposals for studying mashups as an alternative implementation option for PLEs (cf. Johnson et al., 2006). The rationale for looking at mashups for the implementation of PLEs is that they are "distributed web-applications and services that support system-spanning collaborative and individual learning activities in formal as well as informal settings."[2] While this last note hints at the potential role of ActiveTags in realizing PLEs, it also leads to the question of whether its reliance on tags can make it powerful enough.

The first relevant observation is that many Web applications are available through APIs that can fit into a learning context. Figure 4 lists some of them on the right-hand side. Note that whether a map viewer or any of the other applications is relevant for learning depends on the specific learning context. We present the use of ActiveTags as a PLE in a scenario to highlight how the coordinated usage of several services may over time evolve into the creation of a PLE.

The scenario is as follows: Peter is interested in learning more about robotics. He starts by searching the Web for "robotics," and the first entry he finds is the Wikipedia page on robotics. In order to track his findings, he decides to store the interesting pages he finds in the Delicious bookmarking system, where he tags them with "robotics." He does the same for the link to the Wikipedia page. Further searches might lead him to the Web site of Academic Earth[3], which collects video lectures from top US universities, where he finds a lecture on robotics.[4] Peter then decides that he wants to follow the lectures of this course to have a structured way of learning more about the field. He stores the lecture as a bookmark. While watching the first lecture, he finds that he wants to take some notes. He decides to use his Google Docs[5] account, where he can write documents, spreadsheets, and presentations. He stores a link to the document in his bookmarks.

When Peter returns after some time, he opens up his bookmark page and sees the resources he has collected. Being an ActiveTags user he sees that the tags he has added are highlighted.

[2] http://mupple08.icamp.eu/
[3] http://academicearth.org
[4] http://academicearth.org/courses/introduction-to-robotics/
[5] http://docs.google.com

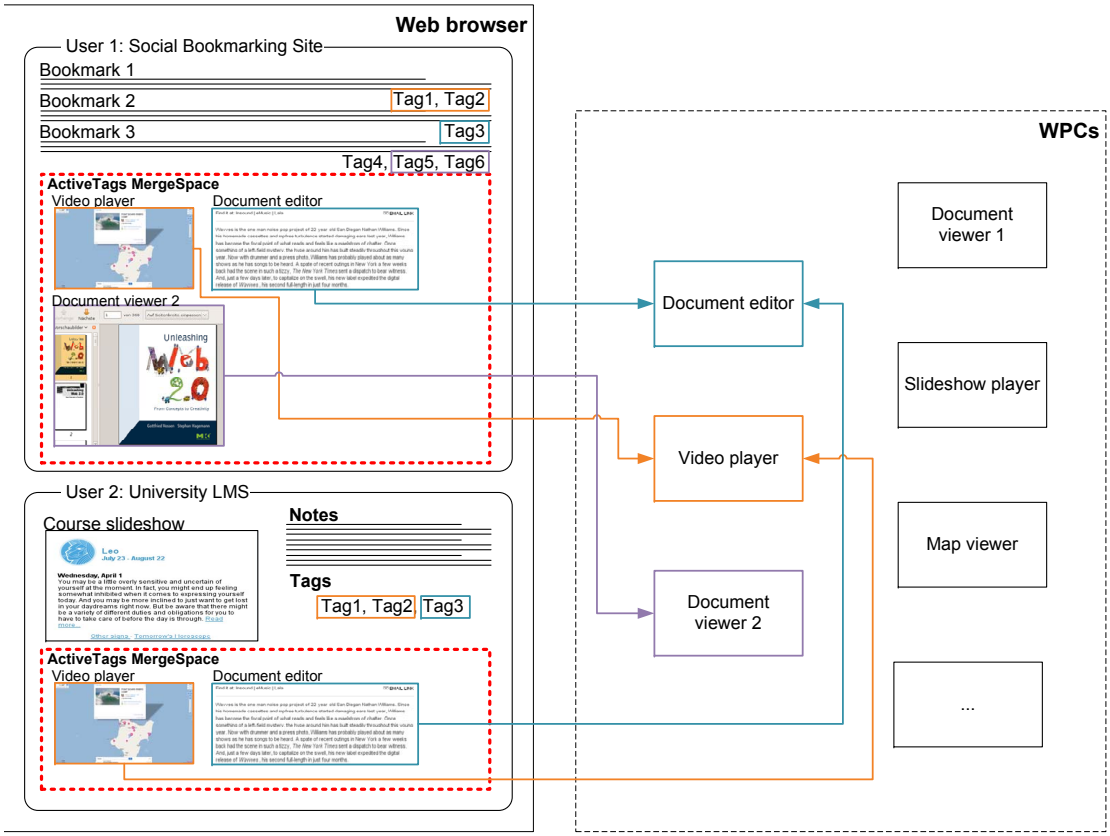

Figure 4: Schematic of two sample uses of ActiveTags for PLEs.

He comes up with the idea that it would be nice to have the content mashed up directly on this page. Checking for relevant mashups, he finds that two are available with which he could integrate the lecture videos and his notes into his bookmarking page. He adds the relevant tags manually, reloads the page, and gets the mashups as intended at the bottom of the page.

So far, Peter has been using tags for two functions: (*i*) He has tagged everything with "robotics" to denote the context, and (*ii*) he has tagged the individual resources with the appropriate tags that activate the mashups. If he does this for many of the lectures, the context may become unclear: Which notes belong to what lecture? How is the structure among lectures or courses? He can deal with this by using more specific tags to denote the context. Instead of using the "robotics" tag he could switch to a combination of tags, such as "robotics, lecture, lecture:number=`i`." Now it becomes clear how contents are organized. This is just one way of enhancing the context and it shows the flexibility of the approach. By using it, Peter ends up with personally tailored learning Web pages, where he sees his notes, the lectures of the course, and bookmarks to further items.

The top left side of Figure 4 (title "User 1: Social bookmarking site") reflects this scenario schematically. The tags on the bookmarked resources lead to mashups of relevant content being shown. Figure 5 shows a screenshot of a browser where this scenario has been implemented with the components mentioned above: Delicious is used for bookmarking, in the top left of the MergeSpace, the video of the lecture can be seen. Next to it are the personal notes. In addition to the above scenario a book on robotics is included here as well.

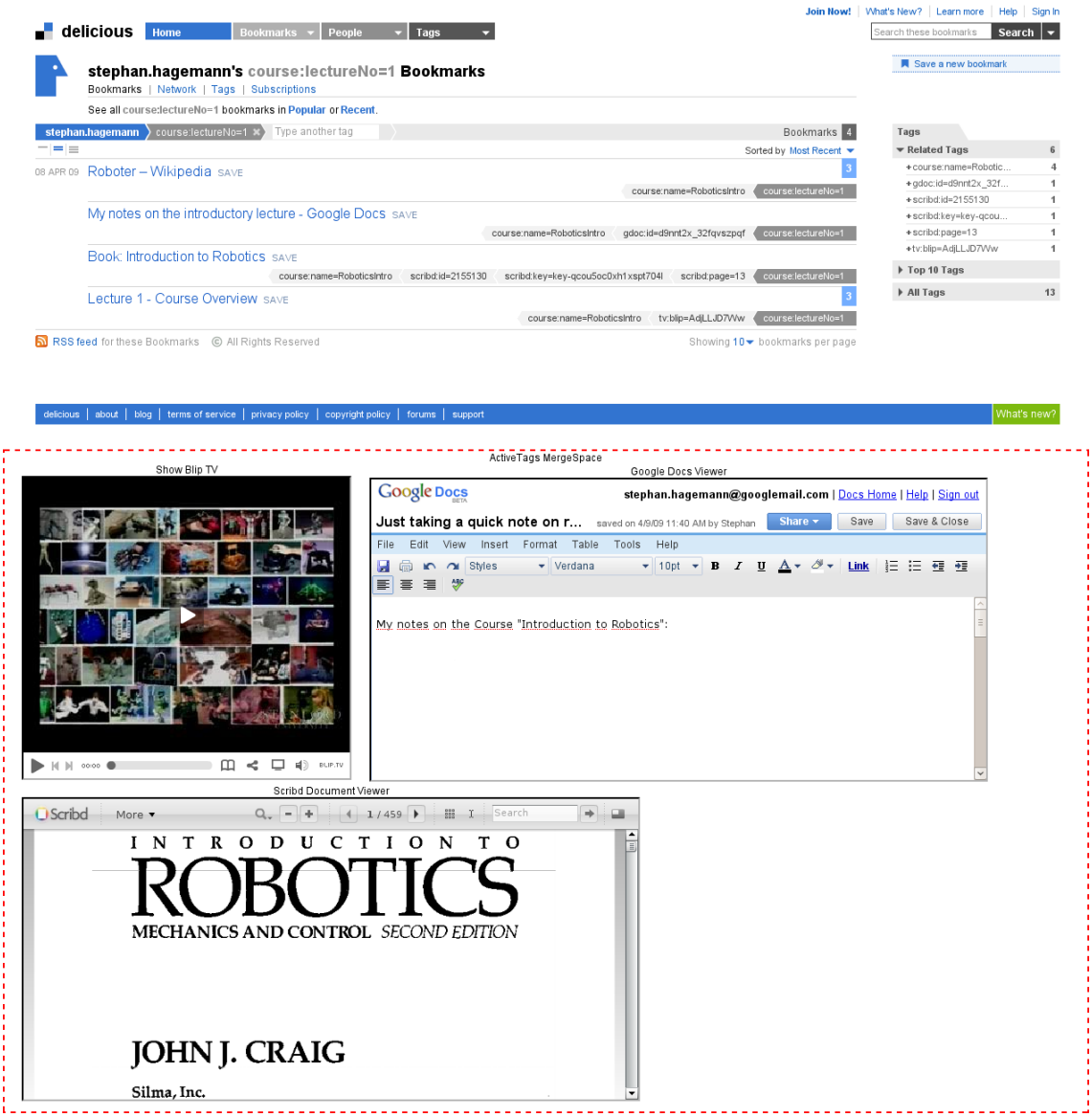

Figure 5: ActiveTags used as a PLE.

Since tagging is so pervasive, the same approach can easily be transferred to other scenarios: A student taking a university course, for which an Learning Management System (LMS) to distribute contents can be enhanced with exactly the same functionality when the tags are copied from Peter's page. Note that while this will call upon the same mashups, the result may be different: For example, the content of the document stored at Google Docs can be accessed only if the Peter has set it to be public or has invited collaborators. This is because the sharing feature of Google Docs directly carries over to the Google Docs mashup. This scenario is schematically depicted in the bottom left of Figure 4 (title "User 2: University LMS").

The process described here is manual: Peter has to enter the relevant tags based on the needs of the mashups. Finding the necessary parameter values might not always be obvious. This falls nicely into the observation of van Harmelen (2008): "While for technically skilled users a browser-based loosely-joined pieces approach works well, the approach presents some difficulties for learners who are less than comfortable with multiple systems."

The difficulties of our loosely-joined approach can be remedied by a tagging support system that complements the mashup capabilities of ActiveTags. The idea is that when the users visit a resource that can be mashed up using a particular ActiveTags mashup, the necessary tags could be generated automatically and offered to the user for copy-and-paste use. This functionality complements TagExtractors in that it extracts tags for the purpose of resource identification in other contexts. Coming back to our scenario one last time, this might allow Peter to add the video lecture to his bookmarks by simply going to the Web page of the lecture and copying the needed tags out of the tagging support extension.

3 Alternatives to Web Application Customization

In this section we look at applications of similar capabilities as ActiveTags, in particular MIT's Piggy Bank and the Intel Mash Maker. It will turn out later that all three, although fundamentally different and design and approach, can be treated in a formal way within the same framework.

3.1 Piggy Bank

The Piggy Bank project[6] states that it "is a Firefox extension that turns your browser into a mashup platform, by allowing you to extract data from different web sites and mix them together." Piggy Bank can improve the Web surfing experience on Web sites that provide their content in Resource Description Framework (RDF) in addition to HTML (cf. Huynh et al., 2007, 2005). Where pages do not come with RDF descriptions of their data, Piggy Bank can employ so-called screen scrapers. These scrapers are similar to the TagExtractors of ActiveTags, but they can be defined to extract any content into an RDF data structure. Scrapers can be site-specific (such as the ones for the ACM Portal or for LinkedIn) or general (such as those for Embedded RDF (eRDF) or Gleaning Resource Descriptions from Dialects

[6] http://simile.mit.edu/wiki/Piggy_Bank

of Languages (GRDDL)).[7] Scrapers are defined in RDF and can be added to an instance of Piggy Bank through the extensions features.

The extension can invoke the extraction of this RDF data and store it locally. Figure 6 shows one view of the data store. Since RDF data from any domain or ontology can be stored, the available views are generic: list, calendar, map, timeline, and graph (the last one is active in Figure 6). Searching for data is supported by a faceted search interface: this allows the user to refine a collection of items by selecting properties. This feature can be seen on the right side of Figure 6. A detailed view presents all the data belonging to an item in a table view. All these features give users a good generic overview over the Semantic Web data they have accumulated.

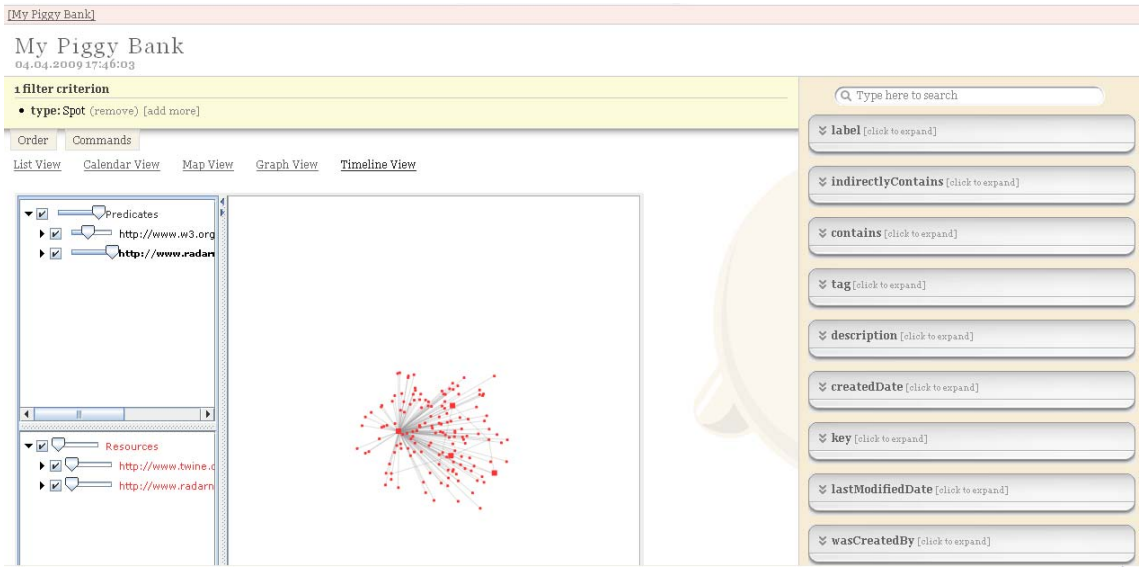

Figure 6: My Piggy Bank.

Users can share their data via Semantic Bank. This is a server implementation that accompanies Piggy Bank. Data from the local bank can be published to the server, where all those that have access to it have the possibility of browsing it (just as in the local version) and can choose to download it into their respective local banks.

As of now, Piggy Bank allows mashups based on the extracted data only in a very limited way. Certainly, if several disconnected pages offer their content as RDF and data from these sources is stored in the local bank, the content of these pages is mixed and shown uniformly in Piggy Bank. However, there is no built-in possibility to make use of this information beyond the generic views. This is due to the generic nature of the store, where more specific views are linked to specific domains for which Piggy Bank currently is too general. As Huynh et al. (2007) states, the authors "envision improvements to Piggy Bank that let users incorporate on-demand templates for viewing the retrieved information items." These templates could be the equivalent to mashup definitions of ActiveTags.

[7] `http://simile.mit.edu/wiki/Category:Javascript_screen_scraper`

3.2 Intel Mash Maker

The Intel Mash Maker[8] is available as extensions to Firefox and Internet Explorer (Ennals et al., 2007). It comes with an API for the creation of widgets, which are Mash Maker applications that perform a variety of functions related to extracted data including the display of mashups. Mash Maker is aimed at the typical Web users, who "should be able to easily create applications and interfaces that are specially customized not only for them, but for the exact task they are performing at that moment." Their "mission is mashups for the moment, on demand" (Ennals et al., 2007).

Similar to Piggy Bank, Mash Maker employs an RDF extractor to gather the data from Web pages and allows custom extractors to be built in case no RDF is available. In contrast to the approach of Piggy Bank, extractors are not created in JavaScript and are not themselves available as RDF. Instead they are built using a combination of point-and-click and menu configurations. Figure 7 shows the creation of an extractor. The large shaded area in the center of the figure shows the item that has been selected, the menu on the left shows that it is stored in an "article" type. Optionally, an extractor can directly be specified using XPath. One can expect that Mash Maker's approach will lead to more extractors being created because of the following reasons: (*i*) they are built collaboratively, (*ii*) incrementally, and (*iii*) they immediately benefit all users because they do not have to be installed individually.

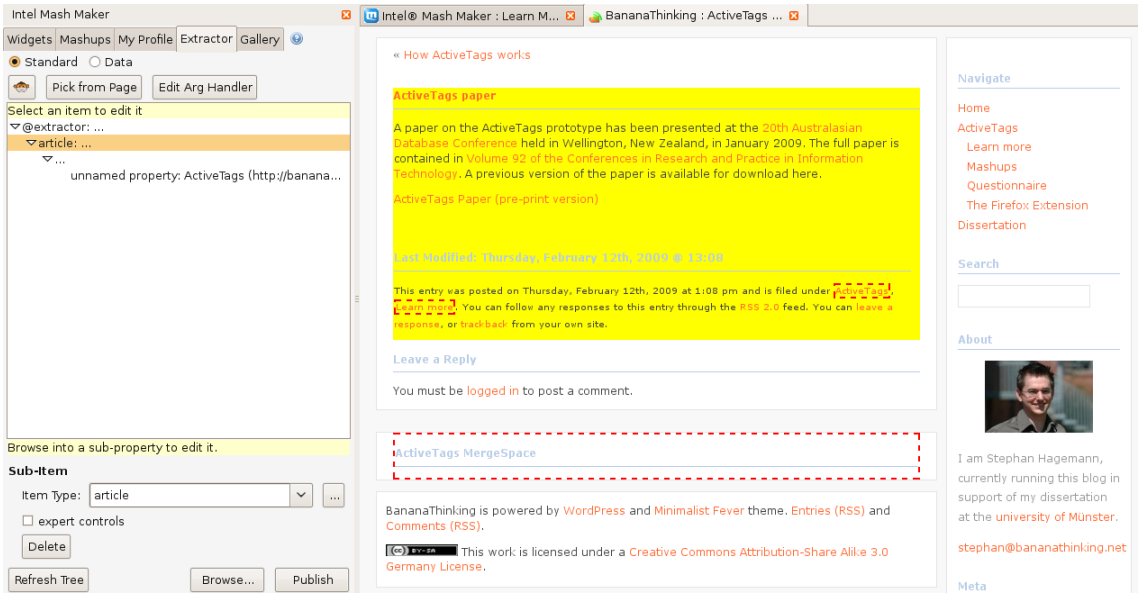

Figure 7: Extractor creation with Mash Maker.

In contrast to the implementation of the ActiveTags server, extractors for Mash Maker are stored on a wiki-like server where any user can edit the extractors of any Web page. To prevent vandalism, individual page extractors can be locked, or extractors can be rolled back to older versions. In addition, the server stores argument handlers that contain information on what parameters of a page are included in the URL. Just like extractors, the argument handlers are specified by the users.

[8] `http://mashmaker.intel.com/`

In spite of all the support, the definition of extractors is a non-trivial process. In combination with the limited spread of RDF, this is the reason why few pages are fully "understood" by Mash Maker. Extractors that do work well include the pages discussed in Ennals et al. (2007) such as housing on craigslist in San Francisco[9] or flight results on expedia.com.[10]

Similar to ActiveTags, the creation of mashups is included in the browser and made part of the ordinary browsing process. To this end, widgets need to be combined meaningfully: "Widgets visualize the data on the page, add additions and action icons to items on the page, provide data to other widgets, and see information that has been extracted from the page by Extractors."[11]

The combination of widgets can lead to a mashup; Figure 8 shows such a case: The source page is a craigslist apartment listing. On top of it, several widgets are shown. The first widget is "Find address," which marks up the individual entries and allows users to trigger the retrieval of that entry's precise address. The "Google Maps" widget shows on a map the addresses that a user picked.

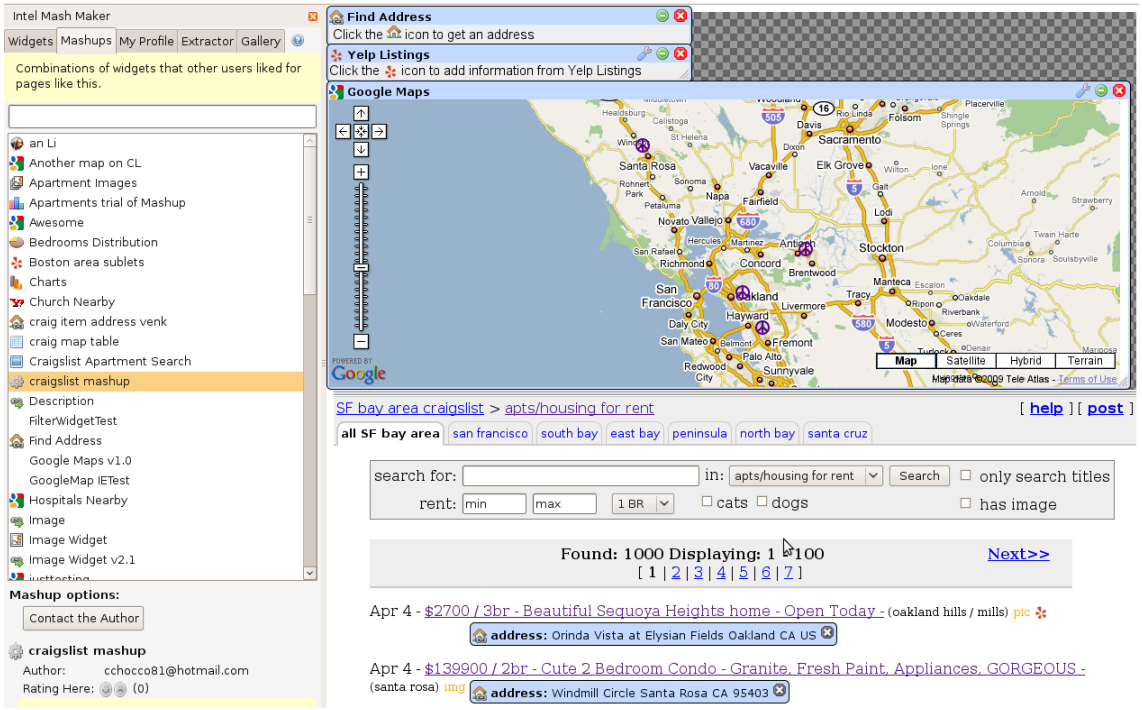

Figure 8: Mash Maker mashup showing craigslist entries on a map.

Mash Maker utilizes all the data from Web pages, either through RDF or through custom extractors. The wiki approach to extractor sharing seems a very promising way of quickly building up a database. That the database is not large yet does not speak against this approach. The widgets approach, combined with the idea to make mashups "on demand" is currently the critical point: its complexity seems to produce many inoperable mashups.

[9] http://sfbay.craigslist.org/apa/
[10] http://www.expedia.com/
[11] http://mashmaker.intel.com/web/techinfo/gs_widgets.html

Finally, Mash Maker does not enable mashups based on tags as they are enabled by ActiveTags. The extractors are too site-specific, and invocation criteria for a mashup based on the content of tags cannot be specified. Invocation is based on the presence of certain properties and not their contents. Nevertheless, it seems to be possible to integrate the functionality of ActiveTags into Mash Maker, since the necessary information is potentially retrievable.

4 A Formalization: Mashup Creation as Meta-Programming

As noted by Hagemann and Vossen (2010), a straightforward analogy can be drawn between the request of a Web page and the execution of a program or a method. In order for a method to be called, it is typically referenced by its name. In addition, parameters may be passed to the method, which specify the particular execution. As depicted in the first row of Fig. 9, the method that requests a Web page is the HTTP GET method. The URL can be seen as its parameter. The execution of the HTTP request leads to the result, i.e., a Web page. When client-side mashups augment the surfing experience, there is an additional step in this process. As the second row in Fig. 9 depicts, the Web page and the extractors define what mashups are to be executed and appended to the Web page. The role of the extractors is thus to specify which portions of the Web page are to be interpreted as the specification of additional "method calls." The result of these executions is again a Web page, but augmented with the results from the execution of mashups.

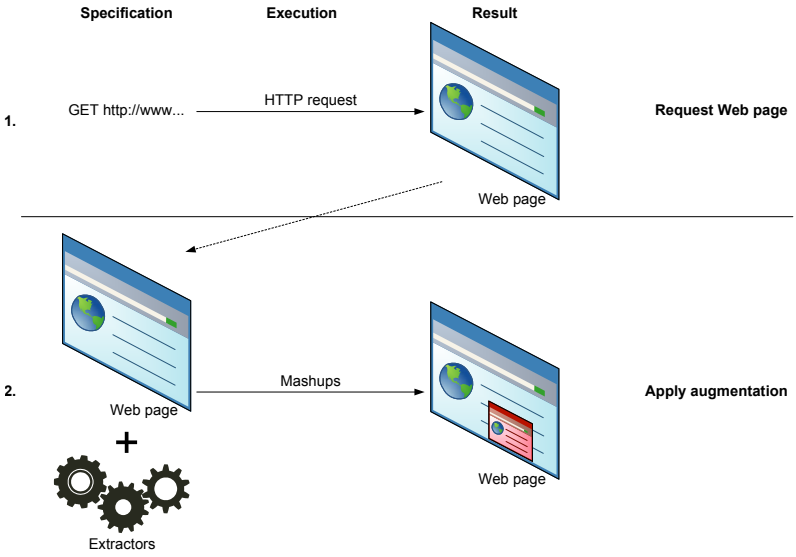

Figure 9: Web page retrieval and meta-querying analogy.

It turns out that the analogy already contains the essence of *meta-querying*: Queries that execute or manipulate other queries as their data. Client-side mashups turn the *data* that is the original Web page into a *query* when extractors are applied and mashups are subsequently executed. One can hence say that a Web page *implicitly* includes the execution information, which is made *explicit* by mashups and extractors. In this section we study this analogy, by reformulating the application of client-side mashups as meta-querying.

Using meta-querying for the realization of the analogy highlights that, once again, seemingly unrelated approaches can be ascribed to relational technology. This holds even for the most recent developments (mashups) and again shows the robustness of the relational model. Such an advancement is in line with the call for a broader application of existing mechanisms from Abiteboul et al. (2005), while at the same time supporting the engineering of Web information systems by contributing a conceptual viewpoint that highlights the need for further developments.

4.1 A Relational View of the Web

In order to be able to analyze the meta-querying analogy more formally, an appropriate language and setting must be chosen first. Any language that allows for meta-programming or even reflection can be used. The question is: How to choose among the multitude of languages that exist with this property? As noted in van den Bussche et al. (1996),

> ... adding reflection to a computationally complete programming language will not enhance its expressive power, the features are typically meant to allow for a more natural or succinct expression of certain advanced programming constructions.

Indeed, computationally complete languages obviously allow for the representation of client-side mashups, since they have been built as such. The question thus boils down to whether a more limited language can be used to model our approach. That this is indeed the case is shown by the WebSQL project Arocena et al. (1997), at least for ordinary Web queries, since it has defined an "SQL-like query language for extracting information from the web."[12]

However, reflection is neither native to Structured Query Language (SQL) nor to the relational world, but has been introduced and analyzed in the literature: Programs as data have been analyzed during the early 1980s already Stonebraker et al. (1984). van den Bussche et al. (1996) presented an approach where relational algebra programs are stored in specific program relations. Dalkilic et al. (1996) introduced Reflective SQL (RSQL), a design and implementation of the latter approach using SQL. The construction of these program relations was shown to be effective, yet quite intricate, but led to a version of SQL that allowed for queries ignoring precise schema information Masermann and Vossen (2000). Building on Stonebraker et al. (1984), a complementary approach was presented in Neven et al. (1999) to create the relational meta algebra, which allows relational algebra expression as a data type in relations.

The culmination of these efforts can be seen in *Meta-SQL* van den Bussche et al. (2005) where the idea of the "program as a data type" is transferred to SQL and to commercial database systems (in particular IBM's DB2). "Practical meta-querying", as it is called, is enabled here by storing queries in an Extensible Markup Language (XML) notation and defining appropriate functions that enable reflection in SQL. A working prototype of this has been developed.[13] In this section we use the results from work on WebSQL and Meta-SQL to remodel the approach of client-side mashup applications from a relational standpoint.

[12] `http://www.cs.toronto.edu/~websql/`
[13] `http://alpha.uhasselt.be/~lucg5855/meta_impl/`

Following Arocena et al. (1997), the Web can essentially be modeled using two relations *Document* and *Anchor* (see Tab. 1). In the *Document* relation we assume, without loss of generality, that the content of the `text` column contains HTML in the form of Extensible Hypertext Markup Language (XHTML). This ensures that we can use XML operations without restrictions. The `url` column identifies each document. All other columns of the document relation contain document metadata. The *Anchor* relation stores the links between documents: One, signified by the `url` column, points to the other denoted in the `href` column with the label contained in the column `label`.

Table 1: The Document and Anchor relations.

url	title	text	length	type	modif
http://www.acme.com/a	title 1	`text 1`	1,234	text	09-01-01
http://www.acme.com/b	title 2	`text 2`	2,691	text	09-01-01
http://www.acme.com/c	title 3	`text 3`	6,372	text	09-01-01
		⋮			

url	label	href
http://www.acme.com/a	label 1	http://www.acme.com/b
http://www.acme.com/a	label 2	http://www.acme.com/c
http://www.acme.com/c	label 3	http://www.acme.com/b
	⋮	

Surfing the Web then becomes requesting a row from the *Document* table. An individual Web page is retrieved from *Document* by selecting on the `url` attribute:

```
select d.title, d.text
from Document d
where url="$SOME_URL",
```

The commutative diagram in Fig. 10 shows the effect of this relational mapping of the Web: In the relational depiction of the Web (denoted as *rel*) an HTTP request resulting in a Web page becomes an SQL query resulting in a "Web tuple" as defined by the query above. This tuple is the same as one would get when applying the "tuplezing" function *tup* to a Web page. This function simply projects the title and the text of a Web page into a tuple.

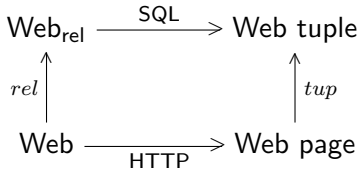

Figure 10: Commutative diagram of the Web and Web pages.

4.2 A Meta-SQL Implementation of Client-Side Mashups

In order to transfer the functionality of client-side mashups into the relational world, we need a way to incorporate it into the relational view and into the way this "relational Web" is queried. As the discussion above has highlighted, it is the extractors and mashup definitions that perform the implicit meta-querying contained in the Web pages. Therefore, essentially these two need to be represented in relations as well. To this end, we introduce two new relations for storing extractors and mashups, resp.

The *Extractor* relation (see Tab. 2) has two columns, with `selector` storing the XPath expression that selects all the tags on a page and `urlPattern` defining, in the form of a regular expression, the URLs on which the extractor operates. This captures the essential properties of TagExtractors from ActiveTags Hagemann and Vossen (2009).

Table 2: The Extractor relation.

selector	urlPattern
`//a[contains(concat('␣',@rel,'␣'),'␣tag␣')]`	`.*`
`//div[contains(@id,'tagdiv')]/a[@class='Plain']`	`^.*(www\.)?flickr\.com`
`//a[contains(@class,␣'spTagboxLink')]`	`^.*www\.spiegel\.de.*$`
⋮	

Tab. 3 shows relation *Mashup*, which contains mashup definitions. The `id` column identifies mashups. `url` typically specifies which document the mashup loads when it is called, note that this column can contain null values. `title` gives the name of the mashup. `tags` contains an XML document which defines the tags required for the execution of the mashup. The root element in this document is always `tags`, in which there are an arbitrary number of `tag` elements.

The tags contained in this XML tree are those necessary for the execution of the mashup. `urlPattern` is a regular expression that is evaluated with the URL of a concrete Web page to test whether the mashup is to be executed. Column `template` contains a template of what the mashup will insert into a Web page should it be executed. Notice that not all mashups have a template. The last column, `program`, contains the SQL code of the mashup. This is the code that needs to be executed to evaluate the effect of a mashup. The depiction shown here is a simplification: We later want to apply Meta-SQL functions, which actually operate on an XML representation of the SQL queries van den Bussche et al. (2005). We have opted for standard SQL syntax here for its compactness in representation.

Querying the mashed-up Web needs to incorporate the contents from the two additional tables. To this end, we need to define how extractors work on documents and how mashups are appended; this done next.

4.2.1 Tag Extraction

Before mashups can be applied, extractors need to be applied in order to determine which tags are present on a Web page. It was stated above that Web pages are stored in XML format.

Table 3: The Mashup relation.

id	url	title	tags	urlPattern	template	program
1	`http://upcoming.yahoo.com/...`	Upcoming event	`<tags> <tag>upcoming:event</tag> </tags>`	`.*`	`<a href="{at:url}"> <at:title/> </a>  `	`select applyTemplate( m.template, m.title, m.url) from Mashup m where m.id = 1,`
2	`http://www.lastfm.de/...`	Last.fm event	`<tags> <tag>lastfm:event</tag> </tags>`	`.*`	`<a href="{at:url}"> <at:title/> </a>  `	`select applyTemplate( m.template, m.title, m.url) from Mashup m where m.id = 2,`
3	`http://dblp.mpi-inf.mpg.de/...`	DBLP search	`<tags> <tag>DBLPsearch</tag> </tags>`	`http:\/\/.* bibsonomy...`	`<h4> <at:title/> </h4> <iframe src="{at:url}"/>`	`select applyTemplate( m.template, m.title, m.url) from Mashup m where m.id = 3,`
4	`NULL`	DBLP & Events	`<tags> <tag>DBLPsearch</tag> <tag>upcoming:event</tag> </tags>`	`.*`	NULL	`select CMB(mashed) from Mashup m, mashed in UEVAL(m.program) where m.id = 1 or m.id = 3,`
			⋮			

Listing 1: XSLT function "extractTags"

```
FUNCTION extractTags PARAM selector STRING RETURNS XML
begin
  <xsl:param name = "selector" />

  <xsl:template match = "/">
   <tags>
    <xsl:for-each select = "$selector">
     <tag><xsl:value-of select="."/></tag>
    </xsl:for-each>
   </tags>
  </xsl:template>
end
```

Listing 2: View definition "DocumentWithTags"

```
1  create view DocumentWithTags as
2  select
3    d.*, t.tags
4  from
5    Document d,
6    (
7      select CMB(extractTags(d.text, te.selector)) as tags
8      from TagExtractor te
9      where regexp(te.urlPattern, d.url) = 1
10   ) t,
```

We can thus use Extensible Stylesheet Language Transformations (XSLT) to make extractors operational. Listing 1 contains the XSLT code of function *extractTags*, which will operate on the XML content of a Web page. Since the extractor selectors are XPath expressions, it suffices to extract all elements from the Web page that are returned by that expression, which is precisely what this function does.

Listing 2 defines view *DocumentWithTags* that uses the *extractTags* function to add a column with the extracted tags to the other information from the document relation. It uses the lateral table syntax to include the text and the `url` of a Web page into the subquery (lines 7–9) that applies *extractTags*. The function *regexp* is used in this subquery to ensure that only applicable extractors are actually executed. It checks that the `url` of the document matches the regular expression defined in column `urlPattern` of the extractor. Not all database systems contain a regular expression function; however, if user-defined functions are allowed (which is the case in most current database systems), it can be defined by implementing such a function Stolze (2003).

The *CMB* function ("combine") is taken from van den Bussche et al. (2005): It is an XML aggregation function that leads from a set of XML documents to a single one. It does so by adding a new root node, which is always `cmb`, and including the source documents $d_1, \ldots, d_n$ as subelements. Column `tags` in view *DocumentWithTags* therefore always contains an XML document that has `cmb` as its root node and several trees of `tags` as subelements.

While we have spoken of "tag extraction" in this section, the extension towards more elaborate information is straightforward. Extracting more information from a page can be implemented by defining several extractors for a page. If other data structures (instead of only tags) are to be allowed, another column may be introduced into the extractor relation, which specifies how the data is to be stored. Mashups may then make use of this. However, even all this may be encoded into tags, for example, by using structured (also called *machine*) tags Straup Cope (2007).

4.2.2 Mashup Inclusion

With view *DocumentWithTags* adding tags to documents we have the first component for the "mashed-up relational Web" in place. The second component needed is the actual combination of documents with mashups. For this we will create another view, but first we need another function, which can check whether a set of tags (eventually coming from a Web page) is sufficient for the execution of a particular mashup. This is what function *checkTagSufficiency* depicted in Listing 3 does.

All `tag` nodes can be selected from the `tags` column in *DocumentWithTags* with the XPath expression `//tag`. This path is used by *checkTagSufficiency* to construct the `tagsOnPageKey` key of all the tags in a source document. The function then checks for all tags in the mashup definition whether a key with the same name exists. This effectively checks if all the required tags are present in (and thus form a subset of) the tags of a document. Only if all required tags are found, the output document will be the empty string; otherwise it will contain one or more "f" characters.

To make *checkTagSufficiency* work with the documents, we construct another view based on *DocumentWithTags*, which is called *DocumentWithMashups* and which is depicted in Listing 4. Essentially, this view copies the columns from *DocumentWithTags*, but performs the necessary transformations on the `title` and the `text` column to include mashups. A note is appended to the title (line 4), making it clear that this document has been processed by ActiveTags. The mashups are added to the text in lines 5–13. The subquery (lines 8–11) executes and prepares the mashups. Two checks are performed for each mashup: (1: line 11) whether the URL matches that of the `urlPattern` (which we have already seen for TagExtractors) and

Listing 3: XSLT function "checkTagSufficiency"

```
1 FUNCTION checkTagSufficiency PARAM tagsOnPage XML RETURNS XML
2 begin
3   <xsl:param name = "tagsOnPage" />
4
5   <xsl:key name="tagsOnPageKey" match="$tagsOnPage//tag" use="." />
6
7   <xsl:template match = "/">
8     <xsl:for-each select = "/tags/tag">
9       <xsl:if test="count(key('tagsOnPageKey',string(.)))=0">f</xsl:if>
10    </xsl:for-each>
11  </xsl:template>
12 end
```

Listing 4: View definition DocumentWithMashups

```
1  create view DocumentWithMashups as
2  select
3    dwt.url,
4    concat(dwt.title, ' (with ActiveTags mashups)') as title,
5    appendMashups(
6        dwt.text,
7        (
8          select CMB(mashup) mashups
9          from Mashups m, mashup in UEVAL(m.program)
10         where checkTagSufficiency(m.tags, dwt.tags) = '' and
11               regexp(m.urlPattern, dwt.url) = 1
12       )
13     ) as text,
14   dwt.length, dwt.type, dwt.modif
15 from
16   DocumentWithTags dwt,
```

(2: line 10) whether the tags in the document are sufficient for execution by checking that *checkTagSufficiency* returns an empty string.

The from-clause uses another Meta-SQL expression from van den Bussche et al. (2005), namely UEVAL. This function is one of two in Meta-SQL that allow for semantical meta-querying. UEVAL dynamically executes SQL expressions (or more correctly, their XML representations) which can be loaded from table columns and returns the table resulting from the execution as a set of XML documents. UEVAL can appear anywhere in an SQL expression where a table reference can be used.

Using again *CMB*, the mashups are combined so that one XML document is returned. Function *appendMashups* is the final one to be applied here: It combines the original document text with the combined mashups.

Listing 5 shows the definition of function *appendMashups*. This function copies all the contents of the original document into the output. When the `body` element is discovered it is copied to the output, too, but in addition the XHTML code generated for the mashups is appended.

4.2.3 Mashup Execution

The final component for querying the "mashed-up relational Web" is the execution of mashups. Looking again at Tab. 3, we see that the first three mashups all contain a template and use function *applyTemplate* in their `program` column. The template contains elements of the form `at:`*parameter*, which the applyTemplate function replaces by the current parameter values. The notation we use is that of Diamond (2002), who provides a full-fledged templating solution for XSLT. This is usable for our implementation also, which is why we do not provide a more specific implementation for *applyTemplate* here.

Looking at the templates in Tab. 3, one can see that those of Mashups 1 and 2 define simple link mashups Hagemann and Vossen (2009): They create a link to the additional information. The

Listing 5: XSLT function "appendMashups"

```
FUNCTION appendMashups PARAM mashups XML RETURNS XML
begin
  <xsl:param name="mashups" />

  <xsl:template match="*">
    <xsl:copy>
      <xsl:copy-of select="@*"/>
      <xsl:apply-templates />
    </xsl:copy>
  </xsl:template>

  <xsl:template match="body">
    <xsl:copy>
      <xsl:copy-of select="@*"/>
      <xsl:apply-templates />
      <h1>ActiveTags mashups</h1>
      <xsl:apply-templates select = "\$mashups/CMB" />
    </xsl:copy>
  </xsl:template>
end
```

respective programs "reselect" the row of the mashup definition to retrieve the other columns as parameters and apply `url` and `title` to the template. The result of *applyTemplate* is the result of the query. Mashup 3 works similarly, but it creates an iframe mashup Hagemann and Vossen (2009): The related information is loaded into the current page as an iframe.

While Mashups 1–3 use essentially the same program, Mashup 4 is defined quite differently and shows the capabilities of this approach. This mashup has `NULL`-valued `url` and `template` columns, which it does not need, as it combines the functionality of Mashups 1 and 3 by selecting their definitions from the mashup database, executing them, and combining their results. This combination leads to a new mashup, which is semantically different from the individual mashups since it is only called, when *all* the necessary tags are present. Note that `url` and `template` do not have to be `NULL`-valued for mashups to use other mashup definitions: more intricate mashup definitions might use both.

4.2.4 The Meta Web Query

All functions and relations needed for querying the "Meta Web" are in now in place. Querying a document from the Web augmented with mashups boils down to the query shown below. Since views have been created to define the necessary steps for tag extraction and mashup inclusion, all that needs to be changed as opposed to the query for the original Web is that the query needs to get documents from view *DocumentWithMashups* instead of from *Documents* directly.

```
select d.title, d.text
from DocumentWithMashups d
where url="$SOME_URL",
```

This last observation brings us back to the commutative diagram of Fig. 10, which has shown the correct abstraction of a Web request in the relational depiction. Figure 11 shows a similar diagram for the mashed-up Web and its relational equivalent. With client-side mashup applications requesting a Web page, they no longer only execute an HTTP request, but also apply client-side mashups, denoted as HTTP_M. The relational equivalent we have constructed in the previous sections is denoted as SQL_M. Note that the results of these functions are still *Web page* and *Web tuple*: ActiveTags produces Web pages (albeit augmented) and the adapted query produces a tuple of the same arity as the original query. This is why the *tup* function can remain unchanged.

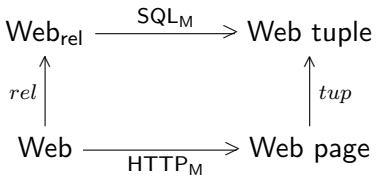

Figure 11: Commutative diagram of ActiveTags-augmented Web and Web pages.

4.3 On the Expressive Power of Meta Web Queries

The formalization presented above uses the analogy with ActiveTags. The question is: What does it say about the alternative approaches to client side mashups, or what is the expressive power of the approach outlined above? As, in terms of functionality related to client-side mashups, Piggy Bank's functionality is a subset of that of Mash Maker, we only need to discuss the latter here.

Intel Mash Maker can operate on any information extracted from a Web page. This includes tags, but can (potentially) encompass more than ActiveTags can work with. The following informal argument shows that this information could also be stored in tags, of which the number is proportional to the size of the Web page.

The content of any Web page can be represented as a tree structure (the DOM tree of the page). The number of nodes in that tree is linearly proportional to the number of HTML tags in the original document. In turn, any tree structure can be represented as a set of RDF triplets. In addition to triplets storing the content (per node in the tree), additional triplets are needed to denote the parent triplet of each triplet and the order of the triplets. In effect, the number of triplets is still linearly proportional to the number of HTML tags in the original document with a small constant factor. Tags have been called "'Poor Man's RDF"' (Andersen, 2005), when it was observed that tags with structure could actually store triplets in a way similar to RDF triplets. So, with some mild convention on how to delimit the parts of a triplet (for example, the machine tag convention (Straup Cope, 2007)), triplets can be written as tags.

The argument now is that, with respect to information that can be used, the approaches of ActiveTags and Intel Mash Maker are hence equivalent. Thus, the above formalization has the expressive power to incorporate Mash Maker's approach. In fact, the above argument

even gives an algorithm for making the entire content of a page processable by the client-side mashup implementation.

5 Conclusions

This paper has presented the Web application customization capabilities of ActiveTags, Piggy Bank, and Intel Mash Maker. Taking the first as an example it was shown that using the new mechanisms entire Web pages are being repurposed and put to new uses. This gives users greater flexibility as to how they can use tagged data. Web sites benefit from more satisfied users and may use ActiveTags as an incubator for new features. The contribution of ActiveTags is therefore beneficiary for consumers and producers. Also, while more elaborate solutions are conceivable and are indeed built, ActiveTags has been shown to offer a quick and simple solution. The example also shows that there is room for more than just the delivery of simple mashups.

At the same time, this paper has proposed a relational meta-querying analogy to client-side mashups. The analogy has been based on ActiveTags, and it has been shown that this does not pose an essential limitation. As such, the analogy has shown the similarity of the approaches of several client-side mashup applications. Additionally, it has explained how these can be elegantly modeled using SQL based on a construction similar to that of WebSQL and with the help of constructs from Meta-SQL. This shows that the meta-querying analogy is indeed helpful in understanding the contribution of client-side mashups.

While in this work the analogy has been used for explanatory purposes only, it is possible to implement it as a running system, since all the components are available. This is a direction for future work which may unify the different approaches. Through a unification of extractor and mashup definitions, this may concentrate the efforts and thus give the approaches greater reach and increase the potential for a wider adoption of client-side mashups.

The combination or unification of the existing solutions can also provide a starting point for the implementation of what has been termed ActiveMetadata in Hagemann (2009). The idea is that the approaches presented here only cover a limited portion of systems that might benefit from integrated mashups. Why not integrate them into the file managers of operating systems or into enterprise resource planning systems?

References

Abiteboul, S., Agrawal, R., Bernstein, P. A., Carey, M. J., Ceri, S., Croft, W. B., DeWitt, D. J., Franklin, M. J., Garcia-Molina, H., Gawlick, D., Gray, J., Haas, L. M., Halevy, A. Y., Hellerstein, J. M., Ioannidis, Y. E., Kersten, M. L., Pazzani, M. J., Lesk, M., Maier, D., Naughton, J. F., Schek, H.-J., Sellis, T. K., Silberschatz, A., Stonebraker, M., Snodgrass, R. T., Ullman, J. D., Weikum, G., Widom, J., and Zdonik, S. B. (2005). The Lowell database research self-assessment. *Commun. ACM*, 48(5):111–118.

Ames, M. and Naaman, M. (2007). Why we tag: motivations for annotation in mobile and online media. In *CHI '07: Proc. of the SIGCHI Conf. on Human factors in computing systems*, pages 971–980, New York, NY, USA. ACM Press.

Andersen, B. (2005). Meta Tags: The Poor Man's RDF? Sci-Fi Hi-Fi Blog. `http://weblog.scifihifi.com/2005/08/05/meta-tags-the-poor-mans-rdf` (2009-04-22).

Arocena, G. O., Mendelzon, A. O., and Mihaila, G. A. (1997). Applications of a Web Query Language. *Computer Networks*, 29(8-13):1305–1315.

Catt, D. (2006). Advanced Tagging and TripleTags. geobloggers Blog. `http://geobloggers.com/archives/2006/01/11/advanced-tagging-and-tripletags/` (2009-04-22).

Chatti, M. A., Jarke, M., and Frosch-Wilke, D. (2007). The future of e-learning: a shift to knowledge networking and social software. *Int. J. of Knowledge and Learning*, 3(4/5):404–420.

Dalkilic, M. M., Jain, M., van Gucht, D., and Mendhekar, A. (1996). Design and Implementation of Reflective SQL (Extended Abstract). Technical report, Indiana University School of informatics.

Diamond, J. (2002). Template Languages in XSLT. O'Reilly xml.com – xml from the inside out. `http://www.xml.com/pub/a/2002/03/27/templatexslt.html` (2009-04-22).

Ennals, R., Brewer, E. A., Garofalakis, M. N., Shadle, M., and Gandhi, P. (2007). Intel Mash Maker: join the web. *SIGMOD Rec.*, 36(4):27–33.

Hagemann, S. (2009). *A Framework for the Consistent Usage of Tag-based Mashups*. PhD thesis, The Münster School of Business and Economics, University of Münster, Münster.

Hagemann, S. and Vossen, G. (2009). ActiveTags: Making tags more useful anywhere on the Web. In Lin, X. and Bouguettaya, A., editors, *20th Australasian Database Conf. (ADC 2009), Wellington, New Zealand*, volume 92 of *Conferences in Research and Practice in Information Technology (CRPIT)*.

Hagemann, S. and Vossen, G. (2010). Web Page Augmentation with Client-Side Mashups as Meta-Querying. to appear in Proc. 2nd Asian Conference on Intelligent Information and Database Systems (ACIIDS) 2010, Hue, Vietnam.

Hinchcliffe, D. (2007). The 10 top challenges facing enterprise mashups. Enterprise Web 2.0 Blog. `http://blogs.zdnet.com/Hinchcliffe/?p=141` (2009-04-22).

Hippel, E. V. (2005). *Democratizing Innovation*. MIT Press, Cambridge, Cambridge, MA, USA.

Hotho, A., Jäschke, R., Schmitz, C., and Stumme, G. (2006). Emergent Semantics in BibSonomy. In Hochberger, C. and Liskowsky, R., editors, *Informatik 2006 - Informatik für Menschen, Band 2, Beiträge der 36. Jahrestagung der Gesellschaft für Informatik e.V. (GI), 2.-6. Oktober 2006 in Dresden*, volume 94 of *LNI*, pages 305–312. GI.

Huynh, D., Mazzocchi, S., and Karger, D. (2007). Piggy Bank: Experience the Semantic Web inside your web browser. *Web Semant.*, 5(1):16–27.

Huynh, D., Mazzocchi, S., and Karger, D. R. (2005). Piggy Bank: Experience the Semantic Web Inside Your Web Browser. In Gil, Y., Motta, E., Benjamins, V. R., and Musen, M. A., editors, *The Semantic Web - ISWC 2005, 4th Int. Semantic Web Conf., ISWC 2005, Galway, Ireland, November 6-10, 2005, Proc.*, volume 3729 of *Lecture Notes in Computer Science*, pages 413–430. Springer-Verlag.

Johnson, M., Liber, O., Wilson, S., Milligan, C., Beauvoir, P., and Sharples, P. (2006). The Personal Learning Environment: A report on the JISC CETIS PLE project. Technical report, JISC CETIS.

Karger, D. R., Ostler, S., and Lee, R. (2009). The web page as a WYSIWYG end-user customizable database-backed information management application. In *UIST '09: Proceedings of the 22nd annual ACM symposium on User interface software and technology*, pages 257–260, New York, NY, USA. ACM.

Lambiotte, R. and Ausloos, M. (2006). Collaborative Tagging as a Tripartite Network. In Alexandrov, V. N., van Albada, G. D., Sloot, P. M. A., and Dongarra, J., editors, *Computational Science - ICCS 2006, 6th Int. Conf., Reading, UK, May 28-31, 2006, Proc., Part III*, volume 3993 of *Lecture Notes in Computer Science*, pages 1114–1117. Springer-Verlag.

Lee, S. E., Son, D. K., and Han, S. S. (2007). Qtag: tagging as a means of rating, opinion-expressing, sharing and visualizing. In *SIGDOC '07: Proc. of the 25th annual ACM Int. Conf. on Design of communication*, pages 189–195, New York, NY, USA. ACM Press.

Lutteroth, C. and Weber, G. (2008). End-user GUI customization. In Marshall, S. and Masoodian, M., editors, *CHINZ*, ACM International Conference Proceeding Series, pages 1–8. ACM.

Macías, J. A. and Paternò, F. (2008). Customization of Web applications through an intelligent environment exploiting logical interface descriptions. *Interacting with Computers*, 20(1):29–47.

Marlow, C., Naaman, M., boyd, d., and Davis, M. (2006). HT06, tagging paper, taxonomy, Flickr, academic article, to read. In Wiil, U. K., Nürnberg, P. J., and Rubart, J., editors, *HYPERTEXT 2006, Proc. of the 17th ACM Conf. on Hypertext and Hypermedia, August 22-25, 2006, Odense, Denmark*, pages 31–40. ACM Press.

Masermann, U. and Vossen, G. (2000). SISQL: Schema-Independent Database Querying (On and Off the Web). In Desai, B. C., Kiyoki, Y., and Toyama, M., editors, *IDEAS*, pages 55–64. IEEE Computer Society.

Musser, J. and O'Reilly, T. (2007). *Web 2.0 - Principles and Best Practices*. O'Reilly Media, Sebastopol, CA, USA.

Neven, F., van den Bussche, J., van Gucht, D., and Vossen, G. (1999). Typed Query Languages for Databases Containing Queries. *Inf. Syst.*, 24(7):569–595.

Rodier, M. (2009). Mashups slowly Gain Traction on Wall Street. Wall Street and Technology. `http://www.wallstreetandtech.com/data-management/showArticle.jhtml?articleID=216401169` (2009-04-22).

Severance, C., Hardin, J., and Whyte, A. (2008). The coming functionality mash-up in Personal Learning Environments. *Interactive Learning Environments*, 16(1):p47 – 62.

Stolze, K. (2003). Bringing the Power of Regular Expression Matching to SQL. IBM developerWorks. `http://www.ibm.com/developerworks/data/library/techarticle/0301stolze/0301stolze.html` (2009-04-22).

Stonebraker, M., Anderson, E., Hanson, E., and Rubenstein, B. (1984). QUEL as a data type. In *SIGMOD '84: Proc. of the 1984 ACM SIGMOD Int. Conf. on Management of data*, pages 208–214, New York, NY, USA. ACM Press.

Straup Cope, A. (2007). Machine tags. Flickr API / Discuss. `http://www.flickr.com/groups/api/discuss/72157594497877875/` (2009-04-22).

Thom-Santelli, J., Muller, M. J., and Millen, D. R. (2008). Social tagging roles: publishers, evangelists, leaders. In *CHI '08: Proc. of the twenty-sixth annual SIGCHI Conf. on Human factors in computing systems*, pages 1041–1044, New York, NY, USA. ACM Press.

van den Bussche, J., van Gucht, D., and Vossen, G. (1996). Reflective Programming in the Relational Algebra. *J. Comput. Syst. Sci.*, 52(3):537–549.

van den Bussche, J., Vansummeren, S., and Vossen, G. (2005). Towards practical meta-querying. *Inf. Syst.*, 30(4):317–332.

van Harmelen, M. (2008). Design trajectories: four experiments in PLE implementation. *Interactive Learning Environments*, 16(1):p35 – 46.

Vander Wal, T. (2005). Explaining and Showing Broad and Narrow Folksonomies. vanderwal.net Blog. `http://www.vanderwal.net/random/entrysel.php?blog=1635` (2009-04-22).

Voss, J. (2007). Tagging, Folksonomy & Co - Renaissance of Manual Indexing? Talk at the 10th International Symposium for Information Science, Cologne, Germany. `http://arxiv.org/abs/cs/0701072` (2009-04-22).

Working Papers, ERCIS

Nr. 1 Becker, J.; Backhaus, K.; Grob, H. L.; Hoeren, T.; Klein, S.; Kuchen, H.; Müller-Funk, U.; Thonemann, U. W.; Vossen, G.; European Research Center for Information Systems (ERCIS). Gründungsveranstaltung Münster, 12. Oktober 2004. October 2004.

Nr. 2 Teubner, R. A.: The IT21 Checkup for IT Fitness: Experiences and Empirical Evidence from 4 Years of Evalutation Practice. March 2005.

Nr. 3 Teubner, R. A.; Mocker, M.: Strategic Information Planning – Insights from an Action Research Project in the Financial Services Industry. June 2005.

Nr. 4 Gottfried Vossen, Stephan Hagemann: From Version 1.0 to Version 2.0: A Brief History Of the Web. January 2007.

Nr. 5 Hagemann, S.; Letz, C.; Vossen, G.: Web Service Discovery – Reality Check 2.0. July 2007.

Nr. 7 Ciechanowicz, P.; Poldner, M.; Kuchen, H.: The Münster Skeleton Library Muesli – A Comprehensive Overview. 2009.

ERCIS – European Research Center for Information Systems
Westfälische Wilhelms-Universität Münster
Leonardo-Campus 3 ▪ 48149 Münster ▪ Germany
Tel: +49 (0)251 83-38100 ▪ Fax: +49 (0)251 83-38109
info@ercis.org ▪ http://www.ercis.org/

ISSN 1614-7448