*Journal of Universal Computer Science, vol. 15, no. 10 (2009), 2109-2137
submitted: 2/3/09, accepted: 25/5/09, appeared: 28/5/09* © *J.UCS*

Application Framework with Demand-Driven Mashup for Selective Browsing

Sohei Ikeda
(Dept. of Computer Science and System Eng.
Grad. School of Eng., Kobe University, Japan
ikeda@cs26.scitec.kobe-u.ac.jp)

Takakazu Nagamine
(Dept. of Computer Science and System Eng.
Grad. School of Eng., Kobe University, Japan
nagamine@cs26.scitec.kobe-u.ac.jp)

Tomio Kamada
(Dept. of Computer Science and System Eng.
Grad. School of Eng., Kobe University, Japan
t_kamada@acm.org)

Abstract: We are developing a new mashup framework for creating flexible applications in which users can selectively browse through mashup items. The framework provides GUI components called widgets through which users can browse mashed-up data selectively, and the system processes demand-driven creation of mashed-up data upon receiving access requests through widgets. The application developer has to only prepare a configuration file that specifies how to combine web services and how to display mashed-up data. This paper proposes a revised widget model for effective data display, and introduces practical applications that allow selective browsing. The revision of the widget model is to accept various GUI components, process user interactions, and provide cooperative widgets. To avoid conflict with lazy data creation, we introduce properties into widgets that are automatically maintained by the system and can be monitored by other widgets. The case study through the applications shows the situations where the initially browsed data helps users to terminate redundant searches, set effective filter settings, or change the importance of the criteria. Some applications display synoptic information through columns, maps, or distribution charts; such information is useful for selective browsing.

Key Words: Web, Web Application, Web Service, Mashup, Ajax
Category: H.4.3, H.3.3, D.2.6

1 Introduction

In recent years, a considerable amount of information has been provided by Web services. Mashup technology allows us to combine different types of information in order to provide a new integrated service. For example, geolocational information can be used to combine real estate services, yellow page services, transit

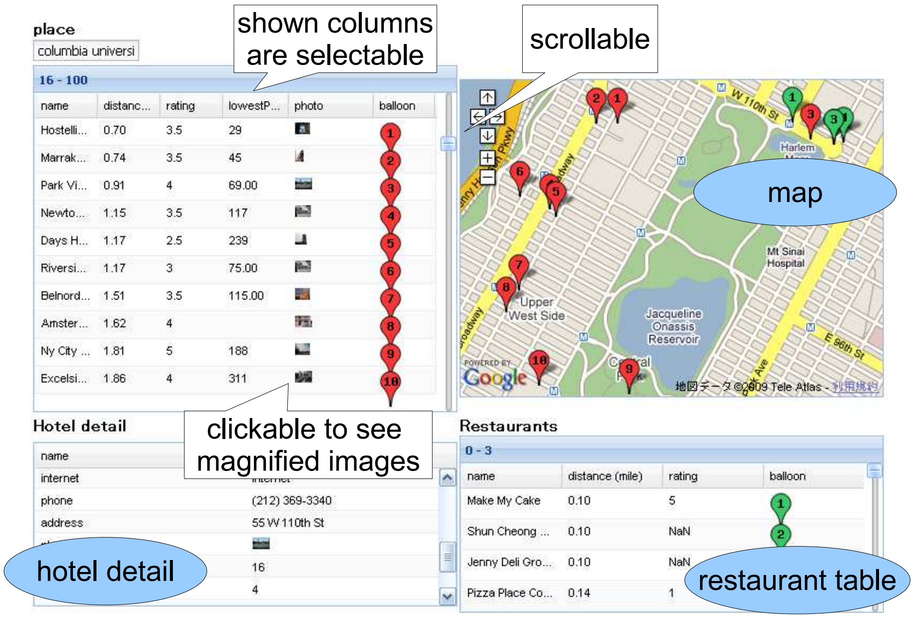

Figure 1: Screenshot of an Sample Mashup Application

information services, photo storage services, etc., and URLs can be used to link Web pages to user review services or blog entries.

Many mashup tools are proposed or developed to help create mashup applications. For example, Yahoo! pipes [Yahoo! Pipes 07] provides GUI environments to create various mashups easily and exhibit them as Web services. Mash Maker [Ennals et al. 07, Ennals and Gay 07, Ennals and Garofalakis 07] allows end users to add various types of information to Web pages by applying mashup fragments that meet their interests. Skilled users can develop and publish mashup fragments for Web pages.

We are developing a new mashup framework for creating flexible applications in which users can selectively browse through mashup items. Our framework provides GUI components called *widgets* that can be used to browse through mashed-up data selectively according to the user's interests, while the system processes demand-driven creation of mashed-up data.

1.1 Application Image

This section introduces a sample application of our mashup framework and illustrates selective browsing of mashed-up data. Figure 1 is a screenshot of the application for a hotel search. The user first inputs the address to specify center point of a search area, and the application displays hotel information near that place. The "hotel table" lists nearby hotels and their information, and the "map" plots their locations. The user can request more information for each hotel by clicking the corresponding row on the table and then get the information from the widgets on the lower part.

Using mashup technologies, the application can prepare various criteria for the hotel selection. For example, the location of the hotel can be used for transit guide services to get transit time from the center place, photo search services to get views near the hotel, or yellow page services to get neighboring restaurants or liquor shops. The hotel name, telephone number, or Web page URL can be keys to make inquiries of user review services or blog services.

To conform to various criteria, our framework provides interactive widgets that allow selection of display targets. For example, our table widget has a facility to dynamically add/change columns to be displayed. In the case of the hotel table, it initially shows the major criteria for the hotel selection and then allows the user to add extra criteria, depending on his/her interests. In Figure 1, the user has added a column of photo thumbnails to show views around hotels. The user can configure the visibility of columns through the popup panel that appears when the header is clicked. Table widgets are useful to compare criteria of listed hotels, such as room rates or distances from the center place. For detailed information, the application shows the information for one hotel at a time for each selected hotel in the hotel table. The lower-left widget shows the "hotel properties," such as addresses, number of rooms, and amenities of the hotel; the lower-right table lists the "neighboring restaurants" of the selected hotel. The application also adopts popup actions for data display. The thumbnail column on the hotel table can be clicked to provide a magnified image with popup viewer.

To cope with interactive and selective browsing of mashed-up data, our system adopts demand-driven data creation of requested data. Receiving add operations of table columns, our system starts creation of the corresponding mashed-up data. When the user clicks the corresponding row to select a hotel, the system processes mashup operations for the hotel to provide the data for lower-placed widgets. If the table has a large number of hotels to be displayed, it displays the data in rows along with a scroll bar. The underlying system first executes mashup operations required to display rows in the initial view; it then incrementally creates data as the user scrolls down the table. Web services that may often produce many results adopt paginated queries in order to return results in multiple pages, and our system fetches paginated results in a demand-driven manner as the user scrolls down the table.

To examine a large number of hotels efficiently, the user can employ filtering facilities based on the values of the specified column, such as room prices or walking distances. The user can activate the facilities through the configuration

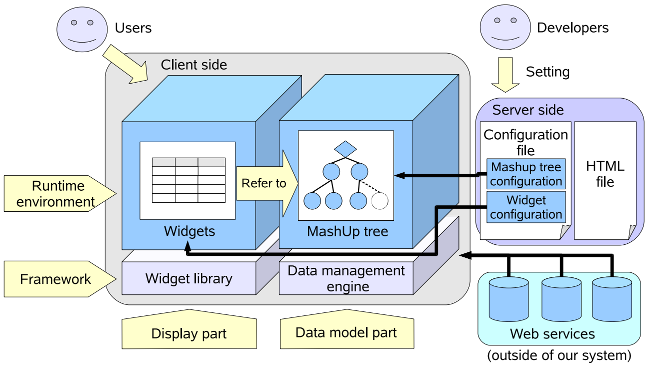

Figure 2: Overview of Our Framework

panel, while the system can avoid needless mashups for hotels that are filtered out.

1.2 Contributions

Our framework adopts demand-driven data creation and interactive widgets to attain selective browsing of mashed-up data. Our system consists of a data management engine and a widget library, as shown in Figure 2. The application developer has to only prepare a configuration file that specifies how to combine Web services and how to display mashed-up data on widgets. Depending on the configurations, widgets monitor user interactions and request mashed-up data from the engine that processes demand-driven data creation, while treating asynchronous calls of Web services or dependencies among the data. These basic concepts of our framework were proposed in [Ikeda et al. 09], but at that point, our framework only assumed table widgets, and it then supported naive coordination among them.

The goal of this paper is to enhance our framework and provide practical applications, as shown in Section 1.1. To achieve this objective, we have revised our widget model to accept various GUI components, process user interactions, and provide cooperative widgets. In the example described in Section 1.1 above, clicking on one of rows on the hotel table leads to redrawing of lower-placed widgets, and scroll-down actions on the table make the map widget adjust display targets to plot all the hotel locations. To process this kind of widget coordination with conventional programming environments, developers are often forced to define event listeners to trigger continuous processes. However, as our framework handles lazy data creation and asynchronous Web service requests, naive introduction of event listeners would cause difficulties in application development. To enable the easy development of these cooperative widgets, we introduce *properties* into widgets that can be monitored by other widgets. These properties are automatically maintained by the system, and updates of property values are announced to the monitoring widgets. Our system also prepares event listeners that retrieve their argument values at the time the corresponding components being displayed that is prior to the event handling.

This paper describes sample applications that utilize cooperative widgets and allow users to browse mashed-up data selectively. Some applications prepare multiple views that can be switched based on user interests, data categories, or searching steps. Our case study shows that there are situations in which a search result can bring about changes in a user's major criteria for subsequent searches, and interactive browsing is useful in those situations. Another situation shows that mashed-up data can help us to carry out a focused search of probable candidates and thereby reduce the number of detailed checks for candidates.

The rest of this paper is organized as follows. Section 2 introduces basic technologies used for mashup applications. Section 3 describes the models of data and widgets and explains how to build applications in our framework using the example described in Section 1.1. Section 4 presents sample applications and discusses the usability and performance advantages of our framework. Section 5 summarizes related work, and Section 6 concludes this paper.

2 Background

This section introduces some major types of Web services. The first type of Web service adopts HTTP GET interfaces with URL parameters. These web services usually return results in XML or JSON formats. Web services that allow unique identification of resources via URIs without having a state such as session tracking are called RESTful Web services. These services can provide improved response time and reduced server load because of their support for caching of representations. Our framework supports Web services that use HTTP GET interfaces, have no state, and return results in XML format.

Web services that produce many results for a request often return results divided into multiple pages. For example, the application described in Section 1.1 uses Yahoo! Local Web service to retrieve hotel information. In this example, the application requests hotels page by page with changing parameters to indicate the starting position of results to be returned, according to scroll-down action on the table.

Next, we briefly introduce other types of Web services. JSONP (JSON with padding) is a technique to retrieve data asynchronously through dynamic embedding of HTML script tags. It is attracting attention because it allows direct access to Web services from browsers, while HTTP GET interfaces are not allowed with cross-domain access without a proxy server. SOAP (Simple Object

Access Protocol) is a protocol to transport data via XML format, and it is used often in enterprise Web services. It can contain not only simple parameters but also meta-data for routing and security.

Some mashup applications exploit information in Web pages by using Web scraping, which is a technique of extracting information from Web pages and restructuring it. Some tools are provided in [Dapper 06, Yahoo! Pipes 07].

As a client-side technology, some mashup applications use a technique called Ajax (Asynchronous JavaScript and XML) in order to provide rich user experiences. For example, users can operate applications continuously without being blocked by Web service requests or waiting for the result. Ajax enables these behaviors by invoking Web services asynchronously. The applications use callback functions that are activated upon receiving the results, and they display data by dynamic HTML manipulation.

3 Our framework

This section first describes our framework models for data and widgets (in Sections 3.1 and 3.2, respectively) and then explains how to build applications using the sample application shown in Section 1.1. Section 3.3 then introduces our system implementation.

3.1 Mashup Tree

Our framework uses a tree data structure called a *mashup tree* to represent mashed-up data, as shown in Figure 3. Our system creates nodes in a demand-driven manner on the basis of tree construction rules set by the application developer. The behavior of on-demand data creation is described in Section 3.3. The system provides the following four types of operations: input operations (`input`, `inputList`), Web service calls (`request`), extractions of XML elements (`extract`, `extractList`), and user-defined operations (`operation`). Figure 3 represents a mashup tree for the example described in Section 1.1. The `cPlace` node receives user input for the center place, and the `geocoder` node represents a Web service call using that input and holds the return value (an XML document) as its node value. The application extracts `latitude` and `longitude` values from the result, and the `hotelSearch` node calls Web services to get an XML document containing a list of hotels. `Hotel` nodes are constructed through the `extractList` operation. The extractList operation receives the XML document from the `hotelSearch` node and searches all XML subelements with specified tag names to build a list of `hotel` nodes, where each element represents an XML element for each hotel, and its child nodes represent operations or values (`name, rating, restaurantSearch`, and so on) for each hotel.

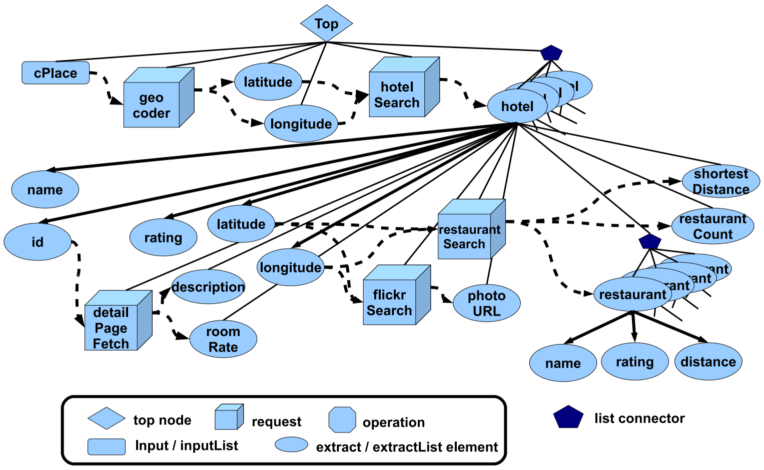

Figure 3: Mashup Tree

```
<mashUpDef>
 <input name='cPlace'/>
 <request name='geocoder'
  baseURL='http://...'>
  <fixedParam key='hl' value='en'/>
  <refParam key='q' src='cPlace'/>
 </request>
 <extract name='latitude' path='lat'
          src='geocoder'/>
 <extract name='longitude' path='lng'
          src='geocoder'/>

 <request name='hotelSearch'
          baseURL='http://...'>
  <refParam key='latitude' src='latitude'/>
  <refParam key='latitude' src='latitude'/>
  <pageParam type='index'>
   <param key='offsetKey' value='start'/>
   <param key='init' value='1'/>
   <param key='countKey' value='results'/>
   <param key='count' value='20'/>
   <param key='correspond'
          value='Result'/>
  </pageParam>
  ...
 </request>

 <extractList name='hotel'
    src='hotelSearch' path='Result'>
  <extract name='name' path='Title'/>
  <extract name='rating' path='Rating'/>
  <request name='restaurantSearch'
           baseURL='...'>
    ...
  </request>
  <extractList name='restaurant'
   src='restaurantSearch' path='Result'>
   <!-- class definitions for children
     nodes of the restaurant nodes. -->
  </extractList>
  ...
 </extractList>
</mashUpDef>
```

Figure 4: Sample Configuration of Mashup Tree

The application developer writes rules for tree construction in XML formats. We are also developing a GUI editor for the rule writing. Figure 4 shows a sample definition, where each rule (called *node class* or *class*) is represented by an XML element with its corresponding tag name. In order to obtain values used for node creation, node classes can access source nodes to get their values. These references to the source nodes are specified at the `src` attribute of the constructor class by using the relative path between node classes. In the path, a parent is denoted by "!" instead of "..". The parent-child relationship between classes is maintained between constructed nodes. As an `extractList` class creates a node list, child classes of the `extractList` class (lines 28–42) are applied to all element nodes of the list.

Specification of Request Classes: Our framework assumes stateless Web services and each request class specifies its parameters within the definition. In case of the `geocoder` (lines 3–7), it specifies a fixed value for parameter `hl` and the node value of `cPlace` for parameter `q`. For Web services with paginated queries, request classes use `pageParam` for page settings. In the case of `hotelSearch` (lines 17–24), it requests 20 items of `Result` elements for each request. The node value of `hotelSearch` represents the concatenation of return XML documents, and `hotel` can list all the hotels in the paginated results. For periodic Web service invocation, the developer can specify the period as `update` attribute values of request classes.

Inputs and Updates of Mashup Tree: Input nodes are used to receive data from widgets. Input nodes can be used to hold a single string value or an XML document, depending on the accessing widgets. When mashup tree receives updates on some input nodes or request nodes, it discards affected parts that depend on the previous values and maintains data coherency. Data recreation for discarded parts is carried out upon receiving data access requests from widgets.

3.2 Widget

This section describes widget models for data display, access to the mashup tree, and widget coordination. We are working on enhancements of widget libraries, and our framework currently provides table-type widgets and a map widget (as described in Section 1.1), along with some container widgets that will be described in Section 4. The map widget wraps interfaces of Google Maps [Google Maps API 05], and can display maps with location markers specified by the widget settings. To build a widget configuration, the developer has to specify how to display mashed-up data. We first describe widget settings for treating the static area of the mashup tree, and we then introduce the concept of how to change display targets, depending on properties of other widgets or occurrences of events.

Figure 5 shows a widget configuration for the application described in Section 1.1. This application uses one widget for text input, three table-type widgets, and one map widget. These widgets are created statically and placed on the HTML page by being embedded in HTML elements that the developer specifies (lines 2, 6, 29, 36, and 44). The developer can append an ID at the `id` attribute of each widget for identification. In this example, `hotelTable` (hotel table in Figure 1) is given a static display target that covers all the `hotel` nodes of the mashup tree (lines 7–8), while the table may actually display some of

```
1 <widgetDef>
2  <embed htmlID='cPlace'>
3   <textBox id='cPlace' target='TREE'
4            nodeName='cPlace'/>
5  </embed>
6  <embed htmlID='hotelTable'>
7   <table id='hotelTable' target='TREE'
8          foreach='hotel'>
9    <columns>
10    <column header='name' nodeName='name'/>
11    <column header='rating'
12            nodeName='rating'/>
13    <column><renderer>
14     <widget type='thumbnail'
15             nodeName='photoURL'>
16      <listeners><listener event='click'>
17       <popup><widget type='imageViewer'>
18        <args><arg key='URL'>
19         <nodeValue path='.'>
20        </arg></args>
21        </widget></popup>
22      </listener></listeners>
23     </widget>
24    </renderer></column>
25    ...
26   </columns>
27  </table>
28 </embed>
29 <embed htmlID='hotelDetail'>
30  <table id='hotelDetail'
31   target='WIDGET::hotelTable.selected'>
32   <columns>...</columns>
33  </table>
34 </embed>
35
36 <embed htmlID='restaurantTable'>
37  <table id='restaurantTable'
38   target='WIDGET::hotelTable.selected'
39   foreach='restaurant'>
40   <columns>...</columns>
41  </table>
42 </embed>
43
44 <embed htmlID='map'>
45  <map id='map'>
46   <marker lat='latitude' lng='longitude'
47       color='red' target=
48       'WIDGET::hotelTable.onView'>
49   <marker lat='latitude' lng='longitude'
50       color='green' target=
51       'WIDGET::restaurantTable.onView'>
52  </map>
53 </embed>
54 </widgetDef>
```

Figure 5: Sample Configuration of Widgets

these nodes, depending on scroll actions or filter settings by the user. Each row of the table corresponds to a `hotel` node, and the values of child nodes (`name`, `rating`, etc.) are shown in columns (lines 9–26). The column for thumbnail image uses inner widget and event listeners that will be explained later. On the basis of the configuration, the table widget manages user interactions, such as paging/scrolling, switching of columns, and filter setting; and it requests the required node values from the data management engine.

Our framework introduces properties into widgets that can be monitored by other widgets and allows easy description of some widget coordination. In this application, hotel table has a `selected` property that denotes the corresponding `hotel` node of the currently selected row, and tables for hotel detail and neighboring restaurants (30–33 and 37–41, respectively) change their display targets to the selected hotel node. In lines 31 and 38, the `target` attributes of the `hotelDetail` and `restaurantTable` are set to `WIDGET::hotelTable.selected`, which means that these tables monitor the `selected` property of the hotel table (`hotelTable`), and these table display child node values of the selected hotel on their rows and columns. When there is a change in the property, the monitoring table treats the newly selected node and its descendant nodes as display targets. The hotel table also employes an `onView` property that is monitored by the map widget (lines 45–48) that plots locations of hotels that are currently displayed on table views by adjusting the scale and center location of maps to cover hotels on

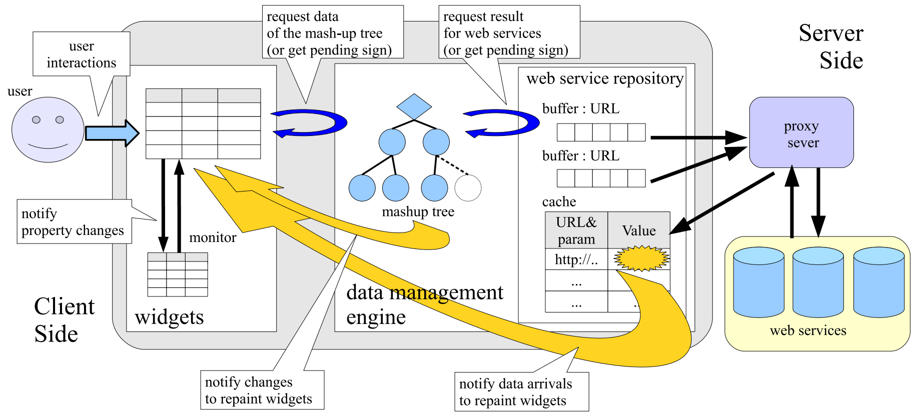

Figure 6: System Implementation

the table view. These properties are basically maintained by the system. Modifications of the values caused by user interactions or updates in the mashup tree are automatically communicated to the monitoring widgets to repaint/redraw the widgets.

To treat widget coordination caused by events, our framework also prepares event listeners that are used to create widgets, change properties of widgets, or set values of input nodes in the data model. In the previous example, the application uses an event listener to handle click events on a thumbnail viewer and create a pop-up window containing an image viewer (lines 16–22). In this example, a listener uses node values for image URLs (line 19: '.' represents the relative path from the `photoURL` node specified at line 15). To avoid conflict with lazy data creation (described in Section 3.3), our system first acquires node values before activating the corresponding listeners. Section 4 will introduce listeners that change widget properties of container widgets to switch display targets or listeners that set values of input nodes to keep probable candidates for the selection.

3.3 System Implementation

Our framework enables demand-driven data creation according to selective browsing, with cooperation between widgets and the data management engine. The engine has a submodule that consists of a Web service repository to treat asynchronous Web service requests. This section describes how we implement demand-driven data creation and the cooperation of these elements (Figure 6). Most of the runtime system works on client sides and uses Ajax technologies.

In our system, widgets manage user interactions and determine which nodes are to be accessed. For example, when a table widget has many rows with a scroll bar or has columns that are turned off, it is sufficient for the widget to request the value of nodes that should be displayed. When a filter setting is applied to the table, it first requests the condition values of rows and then requests values for other columns by skipping filtered rows, until rows that meet the requested conditions fulfill the current view. If a map widget receives many marker entries that include some locations outside the currently displayed area, the widget can skip data access for outside markers.

The data management engine manages demand-driven data creation and asynchronous Web service requests. It returns node values according to requests from widgets. If a node has not been created yet, the engine creates a node after obtaining argument values from the source nodes. When some source nodes are not created yet, the engine processes the creation of nodes recursively. When some node creation requires Web service requests, the engine first invokes asynchronous Web service requests and immediately returns special values to the widgets as an announcement that the node is under construction now. Widgets that receive the announcement will skip the data display or display under construction signs and continue data access for the remaining parts of the display. When the engine receives the return values of the asynchronous requests, it notifies the waiting widgets to redraw their view, and the widgets will request node values again.

The data management engine prepares a mechanism of partial data reconstruction upon receiving changes in input node values. The engine first discards nodes that can be traversed from the input node through data dependency paths (represented by arrows in Figure 3), while data reconstruction is processed in a demand-driven manner, based on widget requests. Our system manages the display targets of widgets, and when the mashup tree discards affected parts of input changes, the system notifies monitoring widgets of the affected parts to redraw the corresponding parts. For widget coordination basing on properties, when some widgets receive user interactions (e.g., row selection or scroll down actions) and change some properties, the widgets communicate the changes to monitoring widgets.

We have summarized the above activities for the application described in Section 1.1. When a user inputs the center place, the data management engine discards affected parts (almost the entire mashup tree), and notifies the hotel table widgets. The hotel table tries to access the list connector for `hotel` nodes, and receives an under construction announcement, while the data management engine invokes an asynchronous hotel search request to get the first page of the result. Upon receiving the announcement, the hotel table resets its properties and notifies monitoring widgets of the new properties. When the Web service repository receives the results of the previous request, it notifies the table widget of data arrival. The table then starts data access to the hotel nodes, changing its properties. The table scans data models to fulfill the current view, while the system invokes asynchronous Web service requests required to create nodes to be accessed. The Web service repository eventually will receive the results, and table widgets will fulfill its view.

The Web service repository manages asynchronous calls of Web services and caches results for previous requests. To avoid excessive access to Web services, we prepare buffering queues for each service site, and the current system keep at least 500 msec interval between invocations. As our system assumes Web services on HTTP GET requests, it prepares proxy services to enable cross-domain Web requests. We hope that these cross-domain requests can use direct connections in the near future.

Finally we discuss some design issues. Our framework does not allow widgets to hold direct references to tree nodes as first class values, but widgets receive node values only from the data management engine. For specification of the display target, the widget can indicate a static area of the mashup tree or subtrees rooted by nodes referenced through property values of widgets. This arrangement avoids ambiguities of lazy data creation when the mashup tree updates its components. If a widget has a dangling reference to nodes that are discarded by the update and no longer belong to the mashup tree, the data access through the reference might cause lazy creation of child nodes using partly new and partly old source values. To avoid these ambiguities, our framework maintains lazy data creation and prevents the existence of "outdated" nodes.

4 Case Studies

4.1 Basic Evaluation

This section first measures some performance values on our data creation engine and widget library using the application described in Section 1.1, then checks the extensibility of the application, and finally discusses the advantages of selective browsing while showing some usage scenarios for the application.

4.1.1 Basic Performance

To measure the basic performance of lazy data creation, we also built a tiny version of the application described in Section 1.1 that only displays the hotel table while removing maps and lower-placed tables. While the hotel table of the original version displays photo thumbnails, the tiny version displays photo URLs as texts to exclude the influence of rendering the images. We used Firefox 3.0.6 and Firebug 1.3.3 for the measurement. The browser runs on a PC with AMD

Table 1: Performance Measurements

	shown columns (`time`: walking time)	# of invocations G	H	R	P	W	time taken to display (sec)	time required for comm. (sec)
1 :	input new place							
	`rate, time`	1	1	10	0	10	14.6	13.8
2 :	input new place							
	`rate, time, photo`	1	1	10	10	10	19.2	18.4
3 :	scroll down by 10 rows†							
	`rate, time`	0	0	10	0	8	9.3	8.9
4 :	input new place with filter††							
	`rate, time`	1	1	10	0	14	21.8	19.9

† The action is done at a situation where the table alrady display first 10 rows and the system has gotton first 20 items in H.

†† The walking time (`time`)is used as a filter condition.

Table 2: Used Web Services

	information	Web services	message size (KB)
G	geocoder	Google Maps	0.4
H	basic hotel information†	Yahoo! Local	14
R	room rate††	Web scraping	1.5
P	photo	Flickr	7.5†††
W	walking time	HopStop	2.7

† The service returns information for 20 hotels at one request

†† The room rate is scraped form a Web page via Yahoo! pipes

††† The size for retrieving a photo is excluded

Athlon 64 X2 3800+ 2.01 GHz, 2048 MB memory and Windows Vista Business (32bit).

Table 1 shows the performance values for several user actions in the tiny version of the application. We measured the numbers of Web service invocations in the system, and the time taken to display all the displayed cells after the actions. The table displays 2 or 3 columns and lists 10 rows of hotels. Web services used in the measurement are listed in Table 2 (we denote each Web service as G, H, R, P, or W in the following parts).

For the number of Web service invocations, our system requests only data needed for columns (comparing action 1 and 2) and rows (check action 1 and 3) currently shown. As some hotels share their locations, the requests of W can use cached values (2 times for action 3, and 2 times for action 4). In the case of

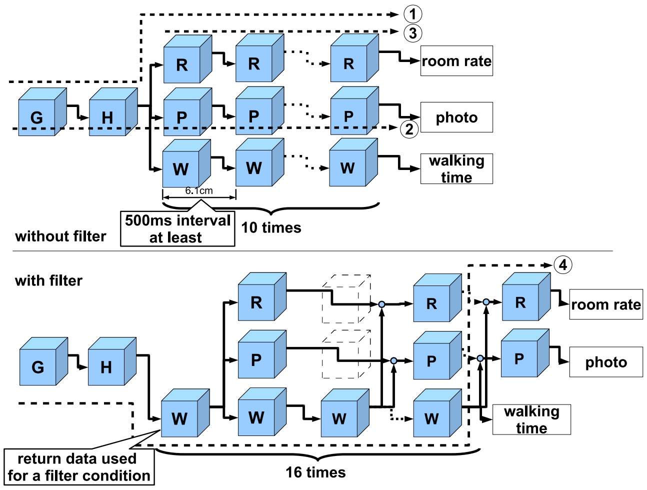

Figure 7: Sequences of Web Service Requests

action 4, the table accesses 16 nodes for the filter condition until the number of unfiltered rows reaches 10, while it accesses 10 nodes for each of other columns. Our system reduces the network traffic for a filtered row from 12.4 KB to 3.4 KB.

Next, we evaluated the response time. Figure 7 shows the sequences of Web service requests inside the system (Some W requests uses cached values). The arrows 1, 2, 3, and 4 in Figure 7 represent critical paths for the corresponding cases up to showing the last cell. The times required for communication in Table 1 show the times for the critical paths, and include response times for Web service invocations on the path and waiting times that may occur to keep at least 500 msec intervals between invocations of the same site. In cases 1, 2, and 3, the times for communication take up more than 90% of the times taken to display all cells. Moreover, our system manages to show initial responses in 1.58–2.62 sec for cases 1 and 2 and 0.53–1.87 sec for case 3. The total execution time is mainly dominated by the time for the communication critical path. In our system, invocations for different sites are processed concurrently. For case 4 with filtering, our system first evaluates condition values to avoid Web service requests for rows being filtered. This lengthens the critical path for communication by only one step. The time for communication is more than 90% of the time taken to display all cells.

Finally, we show some performance values for cooperative widgets using the original version described in section 1.1. We click a row of the hotel table to select a hotel, and measure the time taken to display all the cells of the lower-placed tables. To fill the cells of the hotel detail table, the system has to get the

Figure 8: Screenshot of Extended Application for Tour Planning

reputation and the number of inbound links from blogs by invoking two Web services, in addition to the information already displayed in the hotel columns. The neighboring restaurant table shows information for 4 hotels, which requires only one Web service call. As a result, the time taken for display is 2.5 sec. This time includes the response time of the Web services; the longest response time of the above three services is 1.9 sec in this measurement.

4.1.2 Extending the Application

We extended the hotel search application so that it can treat various categories of places to visit for tour planning and keep probable candidates during the selection. Figure 8 shows a screenshot of the application, while the original version treats only hotel searches. The user can search places to visit in the "place table" by specifying target categories (such as hotels, tourist attractions, museums and so on). The table prepares a different set of initially shown columns for each selected category. As in the original version, the user can click a row to check detailed information and neighboring facilities in lower-placed tables. In the extended version, the user can also specify categories of neighboring facilities.

Now, the application prepares buttons to "pick up" the candidates for visits. The picked-up candidates are listed in the "planning table," and the user can also check information on the places in lower-placed tables by clicking the corresponding rows. The planning table prepares selectable columns to show information, and it has button columns to re-order or remove items in rows.

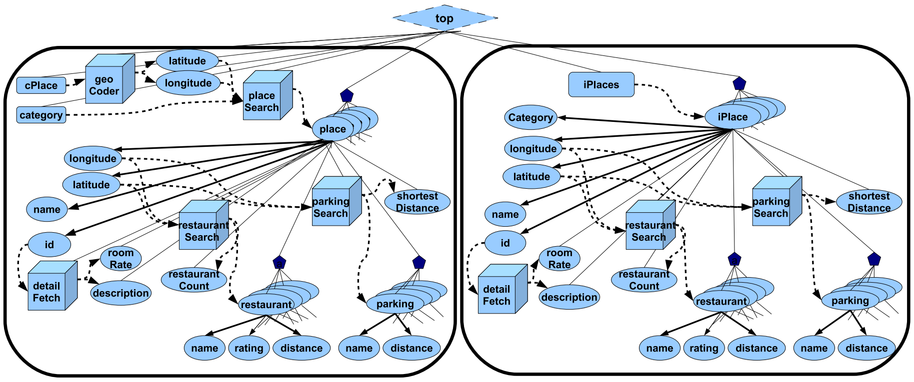

Figure 9: Mashup Tree for Tour Planning

We now explain how to extend the definition of the mashup tree and the widget settings. Figure 9 shows the mashup tree of the extended version. The left half of the tree corresponds to the original mashup tree except that the extended version newly prepares input node of `category` for the place table, and many child nodes of places for neighboring facility search in various categories. The right half of the tree is used to keep data for the planning table, while the left half of the tree changes its contents for every place search. `iPlaces` (`inputList`) is a concatenation of `iPlace` nodes where each `iPlace` corresponds to a selected `place` in the left half tree, and holds the same XML element and the specified category that are passed by the "pickup" button action. `iPlace` prepares almost the same descendant nodes as `place`, by copying the configuration for descendant nodes.

Figure 10 shows the widget configuration for the application. To treat search results for various categories, we can use `switchPanel` widgets that can contain multiple widgets and display one of them depending on the conditions. For example, the place table (lines 17–55) switches the initial setting of the columns to be displayed according to the selected category. In this case, we can prepare tables with different column setting for categories (lines 29–54), and switch them depending on the property value of the `category` widget (lines 18–20 and 7–13). The property values of switchPanel are defined to have the same values as the widget currently being displayed.

Lower-placed tables switch their display target depending on the latest user selection in the place table or the planning table. To find out in which table the latest selection occurred, we prepare event listeners for both tables (lines 21–28 and 61–68), and set the property value of `outerDetailPanel` widget (lines 74–96) to `place` (line 76) or `planning` (line 90). The detail panel also has to

```
1 <widgetDef>
2  <embed htmlID='cPlace'>
3   <inputBox id='cPlace' target='TREE'
4             nodeName='cPlace'/>
5  </embed>
6  <embed htmlID='category'>
7   <select id='category' target='TREE'
8           nodeName='category'>
9    <option value='96926236'
10           label='Restaurants'/>
11   <option value='96926026'
12           label='Tourist Attraction'/>
13   ... </select>
14 </embed>
15
16 <embed htmlID='placeTable'>
17  <switchPanel id='placeTable'>
18   <switch>
19    <widgetValue target='category.@value'/>
20   </switch>
21   <listeners><listener event='select'>
22    <widgetAction method='switch'>
23     <widgetRef target='outerDetailPanel'/>
24      <args><arg key='val'>
25       <fixedValue value='place'/>
26      </arg></args>
27     </widgetAction>
28    </listener></listeners>
29   <panels>
30    <panel case='96926236'>
31     <table id='rSearch' target='TREE'
32            foreach='place'>
33      <columns>
34       <!-- skip some columns-->
35       <column header='pickup'
36               type='button' label='pickup'>
37        <listeners><listener event='click'>
38         <nodeAction method='addItem'>
39          <nodeRef target='TREE'
40                   path='iPlaces'/>
41          <args>
42           <arg key='place'>
43            <nodeValue path='.'/></arg>
44           <arg key='category'>
45            <widgetValue target=
46               'category.@value'/></arg>
47          </args>
48         </nodeAction>
49        </listener></listeners>
50       </column></columns>
51      </table>
52     </panel>
53     <!-- panels for other categories -->
54     </panels>
55   </switchPanel>
56 </embed>
57
58 <embed htmlID='plannigTable'>
59  <table id='planningTable' target='TREE'
60         foreach='iPlace'>
61   <listeners><listener event='select'>
62    <widgetAction method='switch'>
63     <widgetRef target='outerDetailPanel'/>
64     <args>
65      <arg key='val'><fixedValue
66           value='planning'/></arg>
67     </args></widgetAction>
68   </listener></listeners>
69   <columns>  ... </columns>
70  </table>
71 </embed>
72
73 <embed htmlID='detailPanel'>
74  <switchPanel id='outerDetailPanel'>
75   <panels>
76    <panel case='place'>
77     <switchPanel id='dtailPanel4place'>
78      <switch><widgetValue
79              target='category.@value'/>
80      </switch>
81      <panels>
82       <panel case='96926236'>
83        <!-- panel for restaurant seach/>
84       </panel>
85       <!-- panels for search results
86         in other categories -->
87      </panels>
88     </switchingPanel>
89    </panel>
90    <panel case='planning'>
91      <!-- panel for browsing picked up
92        items in the planning table,
93        and also uses switchingPanel.-->
94    </panel>
95   </panels>
96  </switchPanel>
97 </embed>
98 </widgetDef>
```

Figure 10: Widget Configuration of Tour Planning Application

switch its display target depending on the category of the selected place, and we use nested switchPanels in which the inner switchPanel (lines 77–88) monitors category selection (lines 78–80) and switches the panels (lines 81–87). For the neighboring facility table, the switchPanel monitors the selected property of the detail panel and obtains the latest selection. In this case, event listeners are used to handle history-sensitive events that are not bound to widget/tree status.

To provide the facility to keep probable candidates, the place table displays a column containing buttons (lines 35–50) with click event listeners that invokes the `addItem` method of the `iPlaces` node with parameter values (lines 37–49).

The `place` parameter is given the value of the corresponding `place` node, and the `category` parameter is set to the property value of the `category` widget. These event listeners get query results for `place` from the mashup tree, and store them into the right part of the mashup tree. For the descendant nodes of the `place`, node values are not copied into the right part of the tree, and data creation will be needed for their accesses, while descendent nodes of the left and right tree shares Web service invocations and the cache facility of the Web service repository will eliminate redundant Web service invocations.

4.1.3 Usage Scenarios

This part introduces some usage scenarios for the application, and shows situations where interactive browsing creates opportunities to avoid needless data creation, or where synoptic display of mashed-up data helps smooth decisions by the user.

Alice is planning to attend an academic conference to be held in Manhattan to demonstrate digital gadgets. Alice wants to stay in an inexpensive hotel with a good view. She also wants to avoid walking long distances because she is carrying heavy baggage that contains many digital gadgets required for the demonstration. In this application, Alice can turn on the columns that show room rates and photos given by Flickr, and search hotels within walking distance from the conference hall.

When Alice finds a small number of hotels within walking distance after browsing the first few pages that list the hotels in order of distance (e.g. Columbia University has 7 hotels within 1 km as this application returns only hotels that are rated 4 and above), she first checks these hotels. If she likes one of them, she will not browse the remaining results any further, and the system avoids needless mashups for that part.

If the hotels within walking distance do not appeal to her, she has to search for hotels in a wider area. She decides to take a bus or train to reach the conference hall and browses more hotels showing the time taken for walking during transit. As she finds that some hotels require too much walking from the nearest station, she sets a filter by the time taken for walking and the system avoids needless mashups for filtered rows in subsequent results. These two examples show situations where the user decides on actions such as quitting browsing or using filtering facilities after viewing some search results. Such a situation occurs often. For example, if there were many good hotels in the first few pages, Alice could use filtering by a condition such as the amenities provided by the hotels. The application creates only the initial items that have been browsed by the user and allows users to choose actions, and then avoids needless data creation according to the actions.

The following scenario uses the facilities of the extended version. After selecting a hotel, Alice starts making a plan to visit tourist attractions on the day after the conference. Alice searches tourist attractions and chooses some famous places to visit. In addition, as Alice is especially interested in museums, she searches museums by specifying the category and chooses one of them. To visit these places efficiently, she finds their locations on the center map, decides on the order in which to visit the places, and sets up a schedule. Furthermore, Alice wants to search for restaurants around the Statue of Liberty area, which she will visit around lunch-time in her plan; however, she finds few restaurants there.

Then, Alice uses the planning table to choose a place to have lunch at. As Alice becomes interested in having lunch at a good-quality restaurant, she adds columns that show the number of highly rated restaurants near each place; she selects the places with many such restaurants and is then able to see them listed in the neighboring facility table. In this example, she first gathers rough information for each place and then narrows down the places to be inspected in detail. This kind of searching style will reduce the user's checking cost.

The number of highly rated restaurants comes not from the attributes of the place but from the attributes of related factors. This kind of information may be helpful, but the number of attributes will increase. Even if the application prepares many criteria, our system avoids creation of data that is not requested by the user.

4.2 Application Ranges

In this section, we introduce two more applications for recognizing the application ranges of selective browsing.

4.2.1 Application with Interactive Map

The first application searches for rental apartments for users who commute to their offices or schools by train. This application uses map widgets just like the previous applications, but it has some different features as well. In this application, the user searches for apartments in two steps: first, the user checks stations to decide the residence area by using reference information such as commute time or the living environment around the stations, and then, he/she searches for apartments around the stations. The application allows the user to switch between the modes for these steps using tab function (Figure 11).

The "station" mode plots stations on the left map using markers. These markers represent the average rent of apartments around the stations by using the color variation. Further, detailed information such as commute time is tabulated on the right. The user can input his/her requirements; these requirements

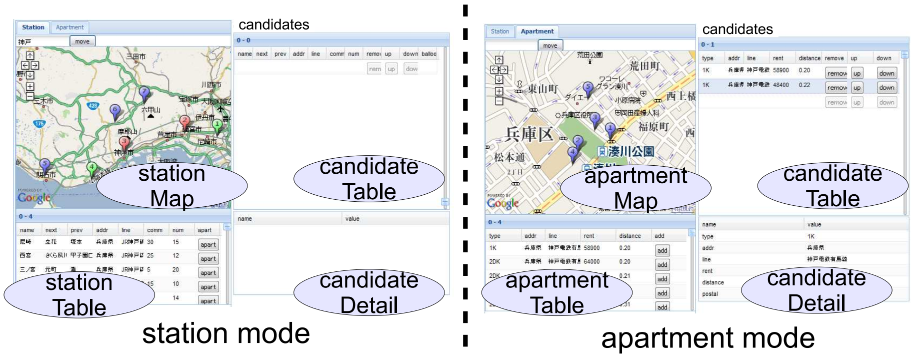

Figure 11: Screenshot of Rental Apartment Search Application

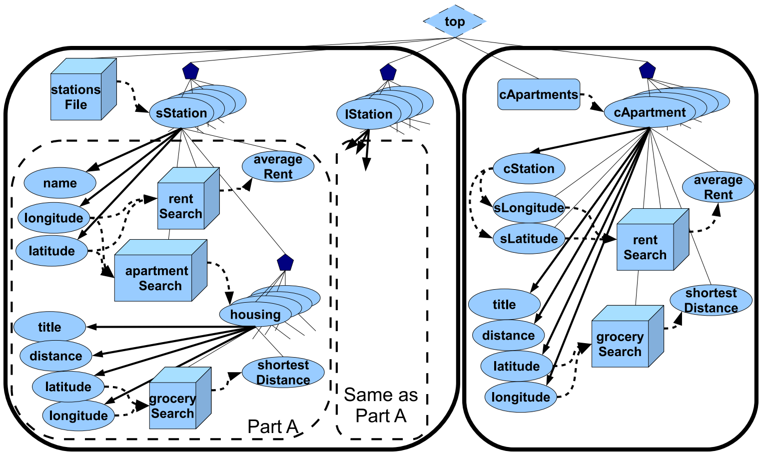

Figure 12: Mashup Tree for Rental Apartment Search

are used as the criteria to search for apartments or to calculate the average rent. As the calculation of the average rent requires Web service invocations, the application plots coarsely-sampled stations at low zoom levels. The user can check the rough distribution of rents in the large-scale map and then zoom in on the area in which he/she is interested. The user can check all the stations in the map zoomed-in map as well as the detailed reference information tabulated on the right. When the user finds an interesting station, he/she just needs to click a button in order to switch to the "apartment" mode and search for apartments around the station. The user can search for apartments in the latter mode and

```
1 <widgetDef>
2  ...
3  <tabPanel id='tabPanel'><panels>
4   <panel label='station'
5          layout='horizontal'>
6    ...
7    <maps id='stationMap'>
8     <markers>
9      <marker id='rMarker' target='TREE'
10        foreach='sStation'
11        lat='latitude' lng='longitude'
12        zoom='5' color='gray'>
13      <properties>
14       <property target='color'>
15        <call fucn='convert2color'><args>
16         <arg key='val'>
17          <nodeValue path='averageRent'/>
18         </arg>
19        </args></call>
20       </property>
21      </properties>
22     </marker>
23      <marker id='lMarker'
24       target='TREE' foreach='lStation'
25       lat='latitude' lng='longitude'
26       zoom='11' color='gray'>
27       ...
28      </marker>
29     </markers>
30    </maps>
31    <table id='stationTable'
32      target='WIDGET::lMarker.onView'>
33      ...
34    </table>
35   </panel>
36   <panel label='apartment'
37          layout='horizontal'>
38     ...
39   </panel>
40  </panels></tabPanel>
41  <table id='candidateTable' target='TREE'
42        foreach='cApartment'>
43   ...
44  </table>
45  ...
46 </widgetDef>
```

Figure 13: Widget Configuration for Rental Apartment Search Application

keep probable candidates in the "candidate table" that can be accessed in both modes. When the user wants to check other stations, he/she can switch back to the station mode to continue the previous activities.

In this application, the user focuses his/her scope in three steps: decisions of area, stations, and apartments. The system displays information according to the search level, while reducing the data creation for mashups.

Next, we explain how to build the application. Figure 12 shows a mashup tree that contains three node lists to hold `sStation, lStation`, and `cApartment` nodes. `sStation` denotes stations that represent each district, and `lStation` represents other stations. We manually prepares a set of `sStation` for this trial. `cApartment` is used for keeping the user's selection of the apartments and the nearest station `cStation`.

Figure 13 shows the widget configuration for the application. This application uses a `tabPanel` widget, which is similar to `switchPanel`, for switching between modes (lines 3–40). The map of the station mode (`stationMap`) uses two sets of markers (lines 9–22 and 23–28) to change display targets according to the zoom level. Each marker monitors `rStation` and `lStation` nodes and sets the threshold zoom level (`zoom` attribute) to specify whether to display the target. The marker checks the values of the `averageRent` node and changes its color depending on the rent (lines 14–20); the color is calculated using a JavaScript function (`convert2color`). The table on the right monitors the `onView` properties of the map in order to show detailed information of stations that are currently displayed on the map (lines 31–34). This relation of monitored-monitoring widgets is opposite to the previous application. According to the configurations

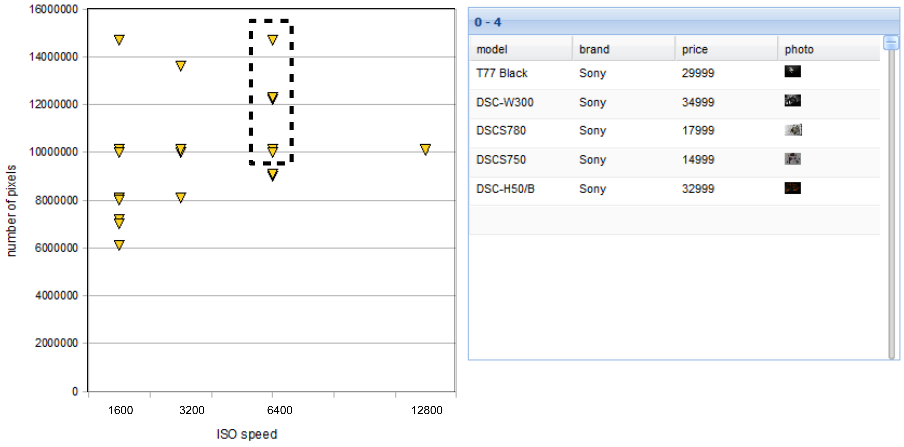

Figure 14: Screenshot of Camera Search Application

```
<widgetDef>
 ...
 <selectableDChart id='cameraChart'>
  <points>
   <point id='cpoint' target='TREE'
          foreach='camera'
          x='iso' y='pixel'/>
  </points>
 </selectableDChart>
 <table id='cTable'
   target='WIDGET::cpoint.regionSelection'
   foreach='camera'>
   ...
 </table>
 ...
</widgetDef>
```

Figure 15: Widget Configuration of Camera Search Application

described above, the application creates the data to be displayed.

4.2.2 Application with Distribution Chart

Next, we introduce a sample application using a distribution chart that we are now developing. The application presents information about digital cameras. Figure 14 is a screenshot image of the application. The chart on the left plots cameras using the ISO speed as X-coordinate and the number of pixels as y-coordinate, and the table on the right shows detailed information of the cameras along with mashed-up data such as the photos taken using these cameras. By selecting a rectangle region on the distribution chart, the user can focus on the camera he/she wants to check. In this application, the user can limit the scope of his/her searches using the distribution chart and selectively browse for only limited products. This type of search method has high affinity with demand-driven data creation.

Now, we explain how the developer can build the application (Figure 15), while the chart widget is at the moment under construction, and hence, the application cannot be run at present. In order to plot cameras, the developer registers `cpoint` that covers all the `camera` nodes in the mashup tree and use `iso` and `pixel` values as `x/y`-coordinates, respectively (lines 5–7). The `selectableDChart` widget provides a facility of region selection with the `regionSelection` property that returns the list of tree nodes in the selection. Thus, the right table only has to monitor the property with describing point ID for the specification of display targets (lines 10–14).

4.3 Summary and Future Work

Our system allows selective browsing of mashed-up data while avoiding data creation for irrelevant parts. The user can select the browsing area interactively and incrementally, and there are situations where the initially browsed data helps users to terminate redundant searches, set effective filter settings, or change the importance of the criteria. The application can provide synoptic information through columns, maps, or distribution charts; such information is helpful for focusing probable candidates. Cooperative widgets are useful for the above-mentioned multiple-step search activities. We are planning to provide various types of widgets for displaying information effectively.

This paper enhances the widget model of our framework, and we are planning enhancements on the data model and its implementation as future work. For example, in order to memorize information for probable candidates, we have to duplicate parts of the mashup tree without sharing the common parts of the subtrees. Our current system only succeeds in preventing duplicate Web service calls. In order to treat such shared data structures, we are planning to introduce definitions of re-usable mashup fragments, hash-based data structures, or join operators and enable the sharing of common data. We also want to improve the implementation of the filter facility provided by widgets. Our system allows dynamic application of filter conditions but does not utilize select conditions of queries that some web services provide. Our data model will need some revisions in treatments of request classes and filter facilities.

5 Related Work and Some Comparisons

This section first summarize related work, and then compares our framework with other mashup tools through the application described in Section 1.1 and 4.

5.1 Related Work

In this section, we first summarize some categories of mashup tools and frameworks, and then introduce some researches that utilize lazy evaluation for Web data processing.

5.1.1 Building Connections for Mashups

Yahoo! pipes [Yahoo! Pipes 07] and Popfly [Microsoft Popfly 07] provide GUI environments to easily create a connection network for mashups, and allow presenting the mashup as a Web service or a Web application, respectively.

The user of Yahoo! pipes structures a connection graph of operators, and the system processes data along the graph edges. Yahoo! pipes provides rich operations such as the location extractor, which analyzes texts and extracts sufficient geolocation for each article. When the user wants to apply operations to each element of a data list, the user can define a subroutine by organizing the operations into a sub graph, and apply the subroutine to each list element using a loop operator. Yahoo! pipes is primarily used to integrate data into an RSS feed that generally has a fixed form. The system process operations on the server side, and does not assume interactive data creation with users.

Popfly provides a GUI environment similar to Yahoo! pipes wherein the user connects operators called *Blocks* to create mashups. The user can allocate UI Blocks to display mashed-up data, and some UI Blocks provide impressive visualization options (e.g. screens to display photo collections in the form of a rotating carousel). Unlike Yahoo! pipes, Popfly can re-evaluate some part of the operation graph in response to changes of user input or notification from the timer operator. UI Blocks can take on behaviors, such as redrawing the whole screen or appending recreated data to existing data, that are different for each UI Block being applied.

Damia [Altinel et al. 07, Simmen et al. 08] provides a GUI environment where the user connects operations to integrate data into a feed, like Yahoo! pipes. As it targets enterprise business users, it can deal with enterprise sources and data types such as email, Excel spreadsheets, and corporate relational databases, in addition to Web feeds. It also enables secure access to those data.

Plagger [Miyagawa 06] is a RSS/Atom feed aggregator written in Perl. It allows users to handle feeds by connecting Perl plug-ins, and supports various forms of data processing, such as sending the content of a feed as an email, in addition to just integrating feeds.

MARIO [Riabov et al. 08] is a tool for simplifying data composition; the user can define mashups without describing explicit connections between operations. It prepares a catalog of data sources and operations that are described with tags, and allows users to explore the space of potentially compassable data mashups and preview composition results as they iteratively select the tags.

The Calo Desktop Assistant [Blythe et al. 08] provides an environment for creating information integration applications. The user does not have to describe explicit connections of web data sources, but attaches semantic types in an ontology to the sources. Given the user query, the system automatically generates the connection of procedures that carries out the query.

5.1.2 Creating Mashups with Data Manipulation

C3W [Fujima et al. 04a, Fujima et al. 04b] allows users to connect Web applications together in a spread sheet view. To use Web applications, the user can bind cells to Web form parameters of the applications, and clip data from the output page to store in cells. The user can also use spread sheet style operators. To duplicate set of operations and connections to Web applications, the user only has to duplicate set of cells, and can apply some customization afterwards. Marmite [Wong and Hong 06, Wong and Hong 07] also presents mashed-up data in a spread-sheet style. The user creates mashups by stepwise addition of new columns. The user can filter rows by using the values of the specified columns. It provides map views to display place information and also provides a function for exporting mashed-up data to text files and databases. As these tools use spread-sheet views, it is hard to treat nested data such as multiple hotels with a list of nearby restaurants for each.

On the other hand, QED Wiki [IBM QED Wiki 07], Afrous [Afrous 07], and EzWeb [MORFEO EzWeb 08] allow users to create mashups by assembling a collection of widgets prepared in their catalogs. The user can put widgets together on a dashboard screen using a drag-and-drop user interface. These tools prepare widgets such as a table widget, a map widget, and a video player widget. These tools also support coordination among widgets like our framework. Some widgets can be used to display data already displayed in other widgets in different formats, for example, a map widget plots locations of the hotels in a table widget. Some widgets can be used to receive user inputs. These tools do not support demand-driven data creation contralled by user actions on them.

5.1.3 Mash Maker

Mash Maker [Ennals et al. 07, Ennals and Gay 07, Ennals and Garofalakis 07] allows the end user to create mashups interactively, browsing mashed-up results. The user interactively applies mashup fragments called widgets to the Web page being browsed. For example, when a user is browsing through a page that contains a list of houses available to rent, with each entry in the list containing a hyperlink to the detailed Web page, Mash Maker can provide a widget that attaches address entries in the Web page, by retrieving each detailed page to extract address information. After attaching address information, the user can apply another widget that searches for facilities near the locations. Mash Maker also provides widgets to visualize data such as a table and a map. The system can suggest major widgets applicable to the page. The user can also keep and share the created mashup that will be applied to the same type of pages.

Skilled users can create widgets, and share them with other users. Mash Maker provides a programming environment for creating widgets. It uses a structured tree to represent mashed-up data, and provides a functional programming model, the Mash Maker language [Ennals and Gay 07]. The programmer can write programs that invoke a sequence of Web services in order to attach results to the tree. Mash Maker provides a mechanism that automatically propagates data changes in the tree on the basis of data dependency given by widgets, and the programmer can define widgets without complicated coding for continuations of asynchronous Web service calls or effects of data changes. The system adopts lazy creation of the tree structures, depending on the adoption of widgets. However, the system does not support lazy data creation according to the behaviors of GUI components like our framework.

5.1.4 Tools for Developing Web Applications

Google Mashup Editor [Google Mashup Editor 07] allows the developer to use enhanced HTML tags for simple mashup operations with other standard technologies such as HTML, XML, CSS, and JavaScript. Using these tags, the developer can retrieve RSS feeds, execute loop operations, and display the result in HTML texts. For more complicated mashups, the programmer can write JavaScript programs.

Flex [Adobe Flex 04] is an environment that facilitates creating Flash applications. The developer defines user interface designs in an XML format called MXML and describes the operation logic using ActionScript wherein Web services can be invoked through easy descriptions. Flex is not a mashup tool, but many components or libraries are provided to utilize mashups (e.g. mashup with Google Maps [Google Maps API 05]).

5.1.5 Lazy Evaluation

Lazy evaluation itself is a well-known concept in functional programming languages, and is supported by some systems (e.g., Haskell). Mash Maker also adopts lazy evaluation of expressions.

AXML [Abiteboul et al. 04a, Abiteboul et al. 04b] (Active XML documents) are XML documents with embedded calls to Web services that will be invoked to generate data for their replacements. AXML has a query language, and algorithms for lazy query evaluation are proposed to avoid materialization of information irrelevant to the query.

Nanafusi [Yamanaka et al. 05] is a programming environment for demand-driven processing of streaming XML data. This work aims at efficient handling of large-scale XML data while hiding network latencies. Nanafusi converts XML streams into the corresponding tree structures in a demand-driven manner, and also provides join operators that can return an initial response quickly. Nanafusi assumes streaming input data and sequential access for the output, while our framework treats discrete or paginated Web service requests for the input and accepts random data access for the output on receiving user requests through widgets.

5.2 Comparison

We now compare our framework to Popfly and Mash Maker, which offer high interactivity among mashup tools discribed in section 5.1.

The application in section 1.1 and its extended version use Web services that return paginated results. For such Web services (in the case of Mash Maker, Web pages), Popfly can create a different page for each paginated result, and Mash Maker can apply mashups to each page. Then, the user can visit only those pages in which he/she is interested and avoid needless mashups. However, the user cannot compare paginated results in the same view (e.g., the map of the application described in Section 1.1), and the systems do not share common mashed-up data among multiple pages.

These tools can collect all the paginated results and treat them as united data. However, if we want to avoid needless mashups on these tools, we have to rely on end users to manage the progress of mashed-up data creation. Popfly and Mash Maker can create special GUI components that can be used for requesting the next paginated result of Web services and mashup operations as the result. However, such levels of coordination between a GUI and Web service invocation are not common in the case of these tools.

Popfly and Mash Maker provide GUI components with filtering facilities for browsing through created data, but these applications cannot avoid needless mashups. In order to avoid needless mashups, we have to filter items before applying mashups, and Popfly and Mash Maker have to prelocate filters in the mashup definitions. However, the user often wants to set filter conditions after browsing through some search results, and these tools require a redefinition of mashups in order to avoid needless data creations. In contrast, when the user changes the filter condition of our widgets, these widgets dynamically change the evaluation order of the display target, and can avoid needless mashup in subsequent browsing attempts.

Popfly and Mash Maker do not provide functions for memorizing user inputs. Memorization is useful in order to connect the items found from different search results. For example, by using inputList for memorization, our application allows the user to choose places to visit from different search results and to compare these choices in the planning table with mashed-up data.

In the application for Real-Estate search described in Section 4.2, the user can move the view of a map and zoom in, while checking detailed information for only the apartments currently in view. As mentioned above, our framework allows the user to focus on probable candidates through rich user interactions and create data for only focused candidates, while Popfly and Mash Maker can create required data in response to simple user interactions such as inputting new data or clicking on links.

6 Conclusion

We are developping a new mashup framework for creating flexible applications that enable selective browsing by the users, and this paper proposes a revised widget model for effective data display, and evaluates the effectiveness of selective browsing through applications. The revision of the widget model is to accept various GUI components, process user interactions, and provide cooperative widgets. To avoid conflict with the demand-driven data creation of our framework, we introduce properties into widgets that are automatically maintenanced by the system and can be monitored by other widgets to change their display targets. Our system also prepares event listeners that perform prior access to the data model.

This paper introduces three applications that allow selective browsing by the users. The case study through the applications shows the situations where the initially browsed data helps users to terminate redundant searches, set effective filter settings, or change the importance of the criteria. Some applications display synoptic information through columns, maps, or distribution charts; such information is useful for selective browsing. When the user reduces the amount of mashed-up data to be checked by selective browsing, our system can avoid needless Web service requests for the data creation.

Acknowledgements

This research was partially supported by the Ministry of Education, Science, Sports and Culture, Grant-in-Aid for Young Scientists (B), 18700029, 2008.

References

[Abiteboul et al. 04a] Abiteboul, S., Benjelloun, O., Cautis, B., Manolescu, I., Milo, T., Preda, N.: "Lazy query evaluation for active xml"; SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data; 227–238; ACM, New York, NY, USA, 2004a.

[Abiteboul et al. 04b] Abiteboul, S., Benjelloun, O., Milo, T.: "Positive active xml"; PODS '04: Proceedings of the twenty-third ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems; 35–45; ACM, New York, NY, USA, 2004b.

[Adobe Flex 04] Adobe Flex: `http://www.adobe.com/jp/products/flex/` (2004).

[Afrous 07] Afrous: `http://afrous.com/` (2007).

[Altinel et al. 07] Altinel, M., Brown, P., Cline, S., Kartha, R., Louie, E., Markl, V., Mau, L., Ng, Y.-H., Simmen, D., Singh, A.: "Damia: a data mashup fabric for intranet applications"; VLDB '07: Proceedings of the 33rd international conference on Very large data bases; 1370–1373; VLDB Endowment, 2007.

[Blythe et al. 08] Blythe, J., Kapoor, D., Knoblock, C. A., Lerman, K., Minton, S.: "Information integration for the masses"; Journal of Universal Computer Science; Vol. 14 (2008), Issue 11, 1811–1837.

[Dapper 06] Dapper: `http://www.dapper.net/` (2006).

[Ennals et al. 07] Ennals, R., Brewer, E., Garofalakis, M., Shadle, M., Gandhi, P.: "Intel Mash Maker: join the web"; SIGMOD Record; Vol. 36 (2007), No. 4, 27–33.

[Ennals and Gay 07] Ennals, R., Gay, D.: "User-friendly functional programming for web mashups"; in the 2007 ACM SIGPLAN international conference on Functional programming (ICFP '07); 223–234; ACM, 2007.

[Ennals and Garofalakis 07] Ennals, R. J., Garofalakis, M. N.: "Mashmaker: mashups for the masses"; in the 2007 ACM SIGMOD international conference on Management of data (SIGMOD '07); 1116–1118; ACM, 2007.

[Fujima et al. 04a] Fujima, J., Lunzer, A., Hornbæ k, K., Tanaka, Y.: "C3w: clipping, connecting and cloning for the web"; WWW Alt. '04: Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters; 444–445; ACM, New York, NY, USA, 2004a.

[Fujima et al. 04b] Fujima, J., Lunzer, A., Hornbæ k, K., Tanaka, Y.: "Clip, connect, clone: combining application elements to build custom interfaces for information access"; UIST '04: Proceedings of the 17th annual ACM symposium on User interface software and technology; 175–184; ACM, New York, NY, USA, 2004b.

[Google Maps API 05] Google Maps API: `http://code.google.com/apis/maps/` (2005).

[Google Mashup Editor 07] Google Mashup Editor: `http://code.google.com/gme/` (2007).

[IBM QED Wiki 07] IBM QED Wiki: `http://services.alphaworks.ibm.com/graduated/qedwiki.html` (2007).

[Ikeda et al. 09] Ikeda, S., Kusano, N., Nagamine, T., Kamada, T.: "Demand-Driven MashUp Framework with AJAX Widget Viewer"; Computer Software; Vol. 26 (2009), No. 3, 20–33; (written in Japanese).

[Microsoft Popfly 07] Microsoft Popfly: `http://www.popfly.com/` (2007).

[Miyagawa 06] Miyagawa, T.: "Plagger"; `http://plagger.org/` (2006).

[MORFEO EzWeb 08] MORFEO EzWeb: `http://ezweb.morfeo-project.org/` (2008).

[Riabov et al. 08] Riabov, A. V., Boillet, E., Feblowitz, M. D., Liu, Z., Ranganathan, A.: "Wishful search: interactive composition of data mashups"; WWW '08: Proceeding of the 17th international conference on World Wide Web; 775–784; ACM, New York, NY, USA, 2008.

[Simmen et al. 08] Simmen, D. E., Altinel, M., Markl, V., Padmanabhan, S., Singh, A.: "Damia: data mashups for intranet applications"; SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data; 1171–1182; ACM, New York, NY, USA, 2008.

[Wong and Hong 06] Wong, J., Hong, J.: "Marmite: end-user programming for the web"; CHI '06: CHI '06 extended abstracts on Human factors in computing systems; 1541–1546; ACM, New York, NY, USA, 2006.

[Wong and Hong 07] Wong, J., Hong, J. I.: "Making mashups with marmite: towards end-user programming for the web"; CHI '07: Proceedings of the SIGCHI conference on Human factors in computing systems; 1435–1444; ACM, New York, NY, USA, 2007.

[Yahoo! Pipes 07] Yahoo! Pipes: `http://pipes.yahoo.com/ pipes/` (2007).

[Yamanaka et al. 05] Yamanaka, M., Niimura, K., Kamada, T.: "A programming environment for demand-driven processing of network xml data and its performance evaluation"; in the 2005 ACM symposium on Document engineering (DocEng '05); 207–216; ACM, 2005.