Ajax-based Report Pages as Incrementally Rendered Views*

Yupeng Fu
UC San Diego
yupeng@cs.ucsd.edu

Keith Kowalczykowski
app2you, Inc.
keith@app2you.com

Kian Win Ong
UC San Diego
kianwin@cs.ucsd.edu

Yannis Papakonstantinou
app2you, Inc. and
UC San Diego
yannis@cs.ucsd.edu

Kevin Keliang Zhao
UC San Diego
kezhao@cs.ucsd.edu

ABSTRACT

While Ajax-based programming enables faster performance and higher interface quality over pure server-side programming, it is demanding and error prone as each action that partially updates the page requires custom, ad-hoc code. The problem is exacerbated by distributed programming between the browser and server, where the developer uses JavaScript to access the page state and Java/SQL for the database. The FORWARD framework simplifies the development of Ajax pages by treating them as rendered views, where the developer declares a view using an extension of SQL and page units, which map to the view and render the data in the browser. Such a declarative approach leads to significantly less code, as the framework automatically solves performance optimization problems that the developer would otherwise hand-code. Since pages are fueled by views, FORWARD leverages years of database research on incremental view maintenance by creating optimization techniques appropriately extended for the needs of pages (nesting, variability, ordering), thereby achieving performance comparable to hand-coded JavaScript/Java applications.

Categories and Subject Descriptors

H.2.8 [**Information Systems**]: Database Management—*Database Applications*

General Terms

Languages

Keywords

Ajax, SQL, View Maintenance

*Supported by NSF IIS-00917379 and a Google Research Award

SIGMOD'10, June 6–11, 2010, Indianapolis, Indiana, USA.

1. INTRODUCTION

AJAX-based web application pages became popular by Gmail and Google Suggest in 2004. They are now a requirement for professional level Web 2.0 web application pages and a cornerstone of Software-as-a-Service applications, since they enable performance and interface quality that are equivalent to those of desktop applications. AJAX (Asynchronous JavaScript And XML) is a conglomerate of technologies and programming techniques for highly interactive web application pages. Its programming model for producing web application pages differs from the prior pure server-side model of the Web 1.0 era, where the page is produced on its entirety at the server side. Section 1.1 discusses the advantages that Ajax offers over the pure server-side model but also the serious complexities and programming challenges that it introduces.

The FORWARD framework simplifies the programming of data-driven application pages by treating them as rendered views, whose data are declared by the developer using a syntactically minor extension of SQL, while the rendering is delivered by page units, which are responsible for data visualization and interaction with the user. The units map to the views, either by use of an API or by the use of unit visual (configuration) templates that put together the page units and the SQL views that feed them with data.

The paper's key contribution is the introduction of declarative SQL programming for the development of the report part of data-driven Ajax pages. Drawing a parallel to how declarative SQL queries simplified data management during the last 30 years, similar productivity benefits can be delivered by the use of declarative SQL queries for the development of data-driven Ajax pages. As has been the case with SQL in the past, the productivity benefits of the declarative approach are due to the framework automatically solving performance optimization problems and providing common functionalities that would otherwise need to be hand-coded by the developer. In particular, FORWARD leverages years of database research on incremental view maintenance and extends it for the needs of pages, as summarized in the contributions list of Section 1.3. The net effect is that FORWARD relieves the Ajax page developer from having to write mundane data synchronization code in order to reflect the effect of the users' actions on the pages. An online demo of FORWARD is available at `http://www.db.ucsd.edu/forward`.

1.1 Ajax background

In a pre-Ajax, pure server-side application[1] a user action on an html page leads to an http request to the server. The server updates its state, computes a new html page and sends it to the client (i.e., the browser). At a sufficient level of abstraction, the new page computation is a function that inputs the server state, which includes the request data, the main memory state of the application (primarily session data), the database(s) and relevant information from external systems (e.g., a credit card processing system) and outputs html. Unfortunately the user experience in pure server-side applications is interrupted: the browser blocks synchronously and blanks out (i.e. displays a blank window) while it waits for the new page. Even in the common case where the new page is almost identical to the old page, aspects of the browser state, such as the data of non-submitted form elements, the cursor and scroll bar positions, are lost and the user spends time to "anchor" his attention and focus to the new rendering of the page.

An Ajax page relies on browser-side JavaScript code, including extensive use of JavaScript/Ajax library components, such as maps, calendars and tabbed dialogs. A user action leads to the browser running an *event handling* JavaScript function that collects data from the page (e.g., from forms and components relevant to the action), and sends an asynchronous *Xml Http Request (XHR)* with a *response handler* callback function specified. The browser does not blank out: it keeps showing the old page while the request is processed and even allows additional user actions and consequent requests to be issued. Later the response handler receives the server's response and uses it to partially update the page's state. The page state primarily consists of (1) the page DOM (Document Object Model) object and its subobjects, which capture the displayed HTML and the state of the HTML forms (text boxes, radio buttons, etc) and (2) the state of the JavaScript variables of the page, which are often parts of third party JavaScript components (such as maps, calendars, tabbed dialogs). The components typically encapsulate their state by exporting methods that the JavaScript functions have to programmatically use for reading and writing it.

The Ajax advantage The Ajax pages' "desktop application feel" and quick responsiveness is due to three advantages over the pure server model:

1. *Partial update speed*: The request processing and the response are focused on the relatively few operations needed to produce the partial update of the page, in contrast to the pure server model where the whole page must be recomputed. Since today's applications are often fueled by multiple queries (e.g., Amazon's user page is fueled by 100+ queries [25]) the partial update strategy can dramatically decrease the response time.
   For example, consider a proposal reviewing application. On the page shown in Figure 1, the reviewers can see each other's reviews as they are submitted and revised. In a pure server-side model, submitting a review for a proposal will require the entire page to be recomputed, including queries for the reviews and average grades of all proposals. On an Ajax page, however, a developer will typically optimize: an asynchronous request will be issued with its input being the review. The server will issue queries only in order to find the id of the newly inserted review and the average grade of the corresponding proposal. Upon receiving the response, the response handler updates subregions of the page's DOM to reflect the small changes.
2. *Continous action on browser*: In the example, once the reviewer submits a review, he can continue reviewing by moving his cursor and scroll bar to the next proposal, even before the response handling function has updated the page. Such behavior is a major HCI improvement [15] over server side applications, where the request is followed by a loss of the page and of the cursor position. The continuous action is enabled by two factors: First, the asynchronous nature of the request prevents blanking out. In the common case where the response handler leaves most of the page unaffected, the user can keep working uninterrupted on most of the page. Furthermore, the browser's synchronous blocking is reduced to the amount of time needed for the response handling function to update the page (after the response has been received), the components to update their state, and the browser to reflow (i.e, to redraw the modified part of the DOM) [29]. In conjunction with the partial update, which minimizes updates on the page and consequent reflowing operations, this leads to a typically negligible wait period.
3. *JavaScript and Ajax libraries*: Third party libraries provide comprehensive collections of client-side JavaScript and Ajax components (such as maps, calendars and tabbed dialogs) that produce large savings in development time due to code re-use. These component libraries enable more polished and consistent user experience across different web applications, and also mitigate the API incompatibilities between browsers.

The above advantages have led to novel applications and features, many of which were practically impossible previously. Such applications capture and quickly respond to actions of the user on the page.

The Ajax challenge The programming of Ajax pages is complex, time consuming and error-prone for many reasons. Indeed, each of the Ajax advantages listed above leads to corresponding programming challenges:

1. Realizing the benefits of partial update requires the developer to program custom logic for each action that partially updates the page. In a pure server-side implementation, the programmer need only write (1) code that produces the report and (2) code for the effect of each individual action on the database. In an Ajax application however, each action also requires (3) server-side code to retrieve a subset of the data needed for refresh (4) JavaScript code to refresh a sub-region of the page. In the running example, (3) and (4) are required for each of submitting a new review, revising an existing review and removing a review. This is obviously laborious and error-prone, as the developer needs to correctly assess the data flow dependencies on the page. For the running example of submitting a review on the page of Figure 1, if the developer had issued a query for `Average Grade`, but not `Reviews`, the page

[1] We also classify as pure server-side applications those that make simple use of JavaScript for UI purposes but without the JavaScript contacting the server, as in Ajax. For example, an application where JavaScript is used to cause submission of a form upon clicking the enter key still qualifies as a pure server-side application for our purposes.

will display inconsistent data if another review had been concurrently inserted into the database.

2. The programmer has to coordinate browser-based JavaScript code with server-side Java and SQL code. That is, developing the Ajax pages requires distributed programming between the browser and server, and involves multiple languages and data models. Furthermore, JavaScript is widely criticized (e.g., see [22]) as too unstructured and error prone. While the lack of strong typing and other conventional programming language features was arguably an advantage when JavaScript was used just for UI purposes, it is a liability nowadays, where JavaScript (thanks to XHR requests) is an integral part of the process that the application implements.
3. While the developer of the pages of a pure server-side application needs to only understand HTML (since the browser automatically parses HTML and turns it into DOM) the developer of Ajax pages needs to understand the DOM, in order to update the displayed HTML, and also understand the component interfaces in order to first write code that initializes components, and then write code that refreshes the components' state based on the nature of each update.

1.2 Framework and language contributions

FORWARD facilitates the development of Ajax pages by treating them as rendered views. The pages consist of a page data tree, which captures the data of the page state at a logical level, and a visual layer, where a *page unit tree* maps to the page data tree and renders its data into an html page, typically including JavaScript and Ajax components also. The *page data tree* is populated with data from an SQL statement, called the *page query*. SQL has been minimally extended with (a) SELECT clause nesting and (b) variability of schemas in SQL's CASE statements so that it creates nested heterogeneous tables that the programmer easily maps to the page unit tree. A user request from the context of a unit leads to the invocation of a server-side program, which updates the server state. In this paper, which is focused on the report part of data-driven pages and applications, we assume that the server state is captured by the state of an SQL database and therefore the server state update is fully captured by respective updates of the tables of the database, which are expressed in SQL. Conceptually, the updates indirectly lead to a new page data tree, which is the result of the page query on the new server state, and consequently to a new rendered page.

FORWARD makes the following contributions towards rapid, declarative programming of Ajax pages:

- A minimal SQL extension that is used to create the page data tree, and a page unit tree that renders the page data tree. The combination enables the developer to avoid multiple language programming (JavaScript, SQL, Java) in order to implement Ajax pages. Instead the developer declaratively describes the reported data and their rendering into Ajax pages.
  We chose SQL over XQuery/XML because (a) SQL has a much larger programmer audience and installed base (b) SQL has a smaller feature set, omitting operators such as `//` and `*` that have created challenges for efficient query processing and view maintenance and do not appear to be necessary for our problem, and (c) existing database research and technology provide a great leverage for implementation and optimization (see Section 1.3), which enables focus on the truly novel research issues without having to re-express already solved problems in XML/XQuery or having to re-implement database server functionality. Our experience in creating commercial level applications and prior academic work in the area (see Section 5) indicate that if the application does not interface with external systems then SQL's expressive power is typically sufficient. We briefly describe in the Future Work (Section 6) the issues arising in interfacing to external systems.
- A FORWARD developer avoids the hassle of programming JavaScript and Ajax components for partial updates. Instead he specifies the unit state using the page data tree, which is a declarative function expressed in the SQL extension over the state of the database. For example, a map unit (which is a wrapper around a Google Maps component) is used by specifying the points that should be shown on the map, without bothering to specify which points are new, which ones are updated, what methods the component offers for modifications, etc.

Roadmap We present the framework in Section 2 with a running example. Section 2.3 presents the data aspects of the framework. Section 2.4 presents the visual layer.

1.3 System and optimization contributions

A naive implementation of the FORWARD's simple programming model would exhibit the crippling performance and interface quality problems of pure server-side applications. Instead FORWARD achieves the performance and interface quality of Ajax pages by solving performance optimization problems that would otherwise need to be handcoded by the developer. In particular:

- Instead of literally creating the new page data tree, unit tree and html/JavaScript page from scratch in each step, FORWARD incrementally computes them using their prior versions. Since the page data tree is typically fueled by our extended SQL queries, FORWARD leverages prior database research on incremental view maintenance, essentially treating the page data tree as a view. We extend prior work on incremental view maintenance to capture (a) nesting, (b) variability of the output tuples and (c) ordering, which has been neglected by prior work focusing on homogeneous sets of tuples.
- FORWARD provides an architecture that enables the use of massive JavaScript/Ajax component libraries (such as Dojo [30]) as page units into FORWARD's framework. The basic data tree incremental maintenance algorithm is modified to account for the fact that a component may not offer methods to implement each possible data tree change. Rather a best-effort approach is enabled for wrapping data tree changes into component method calls.

The net effect is that FORWARD's ease-of-development is accomplished at an acceptable performance penalty over hand-crafted programs. As a data point, revising an existing review and re-rendering the page takes 42 ms in FORWARD, which compares favorably to WAN network latency (50-100 ms and above), and the average human reaction time of 200 ms.

National Science Foundation

Review Proposals

ID	Title	Reviews			Grades	My Review	Avg. Grade
		Comment	Grade	Reviewer			
509	Flying cars	Creative idea.	3 - Good	tom@abc.edu			2.7
		I like it!	4 - Very Good	jane@abc.edu			
		Ridiculous!	1 - Poor	john@abc.edu			
568	Invisible cloak	Promising!	5 - Excellent	jack@abc.edu			4.3
		Interesting idea!	4 - Very Good	patric@abc.edu			
701	Time machine	I don't quite see the value of this project!	2 - Fair	bob@abc.edu		Comment: Good stuff. Grade: 4 - Very Good	3.3
		The concepts are original.	4 - Very Good	jimmy@abc.edu			
810	Perpetual motion machine	Really good proposal!	4 - Very Good	jack@abc.edu			2.3

3 – Good
1 – Poor
2 – Fair
3 – Good
4 – Very Good
5 – Excellent
Edit Remove

Edit Remove

Figure 1: `Review Proposals` page

Roadmap Section 3 presents optimizations for incrementally maintaining the page, with Section 3.1 highlighting the incremental view maintenance of the page data tree, and Section 3.2 presenting the architecture for incrementally refreshing the visual layer. Lastly, Section 4 presents the system implementation and experimental results.

2. THE FORWARD DATABASE-DRIVEN FRAMEWORK

2.1 Running example

The running example is a simplified version of the FastLane web application, which the National Science Foundation (NSF) uses to coordinate the submission and reviewing of proposals among Principal Investigators (PIs), Reviewers and Program Directors.[2] First, a PI visits a `Submit Proposal` page to submit the project description, budget estimates and personnel particulars. After the proposal submission deadline, NSF invites Reviewers to a panel, during which they collaborate on the `Review Proposals` page (Figure 1). Each Reviewer sees the titles of proposals assigned, and can click on them to access proposal details. For each proposal, a Reviewer can submit and revise a review comprising a textual comment and a grade ranging from 1 to 5. In addition, a Reviewer can see others' reviews, a bar chart visualizing the respective grades, and the average grade. Finally, the Program Director uses a `Recommend Proposals` page to peruse all reviews provided and indicate which proposals are recommended for funding.

2.2 Architecture

Figure 2 shows an overview of the architecture of the FORWARD framework. In FORWARD, each page is described by a unit tree that has a corresponding *page schema*. The unit tree synchronizes between the *page data tree*, which conforms to the page schema, and the *browser page state*, which includes the state of JavaScript components and HTML DOM. As a user interacts with a page, events can happen which are triggered by either a direct user action (e.g., clicking a button) or other mechanisms such as timers, and leads to an invocation of a server-side program that updates the server state. A program has access to (1) the context and form data[3] of the program invocation, and (2) a SQL

[2]A demo of the original Fastlane application is available at `https://www.fldemo.nsf.gov/`

[3] If the page contains HTML forms that are both initialized by the query and updatable by the user, interesting challenges arise around the programs requiring unified access to both the database and the user provided data. The details of such unified access mechanisms are beyond the scope of this paper and are briefly discussed in the Future Work section.

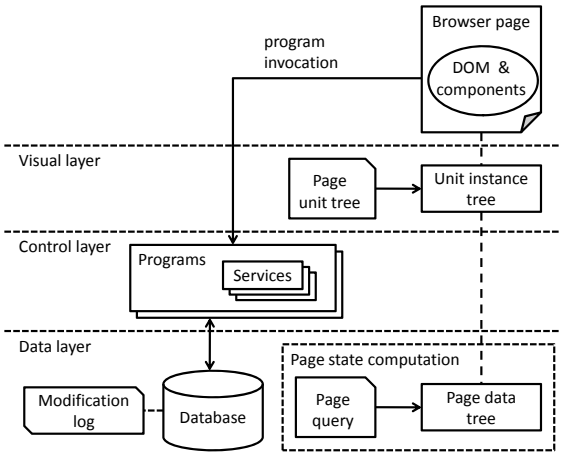

Figure 2: FORWARD architecture

database. Using these data, a program can issue `INSERT/UPDATE/DELETE` commands on the database. In FORWARD the server state is completely captured by the state of the database and therefore the server state update is fully captured by a *modification log* that stores all DML commands on the database's tables. After the program invocation, a *page state computation* module creates the data tree of the new browser page state.

In order to support the Ajax incremental update of a page, the respective renderers of units translate the data difference between the old and the new page data trees to method calls of JavaScript components, as well as updates to the HTML DOM. Furthermore, the data difference is automatically computed in an incremental way without recomputing the page state from scratch. This is possible because the computation of a page's data is specified using a query, called the *page query*. As a result, the page data tree is essentially a view over base tables. The framework logs the modifications to the state of the base tables in the same application, and employs incremental view maintenance techniques to obtain the changes to the view. Incremental page update is the core focus of this paper, and technical issues are discussed in detail in Section 3.

2.3 Data layer

The *page data tree* captures the page's state at a logical level using a minimal extension of SQL's data model with the following features that will facilitate mapping to the page's data from the unit tree. First, the data tree has both sets and lists to indicate whether the ordering of tuples matters; e.g., the grade options in Figure 1 and the proposals are a list while the reviews of the proposal form a set. Second, it has nested relations; e.g., nested reviews within proposals. Finally, it allows heterogeneous schemas for the tuples of a collection; e.g., a tuple corresponding to the input mode of `My Review` also carries the nested list of grade options while a tuple corresponding to the display mode only carries comments and grades.

The schema of a data tree is captured by a *schema tree*.

```
list(
 tuple(
  proposal_id  : 509 ,
  title        : Flying Cars,
  average_grade: 4.5,
  reviews      :set(
   tuple(
    review_id :  1,
    comment   :  Creative...,
    grade     :  3 - Good,
    reviewer  :  tom@abc.edu)
    ...),
  grades       : list(
   tuple(
    bar_id : 1,
    value  : 3)),
   my_review   : switch(
    input_tuple: tuple(
     comment : null,
     grade   : tuple(
      grade_ref    : null,
      grade_options:list(
       tuple(
        grade_id   : 1,
        grade_label: 1 - Poor)
        ...)))
    ))
 ...)
```

Figure 3: Page data tree

```
list(
 tuple(
  proposal_id  : int,
  title        : string,
  average_grade: float,
  reviews      : set(
   tuple(
    review_id : int,
    comment   : string,
    grade     : string,
    reviewer  : string)
    ),
  grades       : list(
   tuple(
    bar_id : int,
    value  : int)),
  my_review    : switch(
   input_tuple: tuple(
    comment : string,
     grade  : tuple(
      grade_ref    :int,
      grade_options:list(
       tuple(
        grade_id   :int,
        grade_label:string)
      ))),
   display_tuple : tuple(
    comment : string
    grade   : string
   )))
 )
```

Figure 4: Page schema

Each data node (also called *value*) maps to a schema node, and the data tree is homomorphic to the schema tree. A value can be one of the following:

1. an *atomic value*, such as string, integer, boolean etc.
2. a *tuple*, which is a mapping $[a_1 : v_1, \ldots, a_n : v_n]$ of attribute names to values, whereas attributes $a_1, \ldots, a_n$ are distinct.
3. a *collection*, which contains tuples $v_1, \ldots, v_n$ such that $v_1, \ldots, v_n$ are homogeneous, i.e. each maps to the same schema node. Furthermore, the schema node specifies a primary key constraint, such that tuples in the collection are uniquely identifiable. An unordered collection is a *set*, whereas an ordered collection is a *list*. The primary keys will play a key role in the incremental maintenance of the page.
4. a *switch value*, which is a tagged union $[c_1 : v_1, \ldots, c_n : v_n]$ of case / value pairs, such that only one pair can be selected. While the switch value represents only the selected pair, the switch schema represents the schema of all possible case/value pairs.
5. a *null value*.

For example, Figure 3 shows the page data tree that represents the list of proposals at the `Review Proposals` page. In a proposal tuple, `title` is an atomic string value, `reviews` is a nested set of review tuples, and `my_review` is a switch value where the `input_tuple` case is selected. Note however that the corresponding switch schema in Figure 4 contains both `display_tuple` and `input_tuple` cases to indicate that a reviewer's review can be either in display mode or input mode.

We extend SQL for nesting (in the spirit of OQL [5]) and variability. Furthermore, a query without an `ORDER BY` clause produces a set while a query with `ORDER BY` produces a list. The following query produces the `Review Proposals` page data tree:

```
1  SELECT P.proposal_id, P.title,
2      (
3      SELECT   *
4      FROM     reviews R
5      WHERE    R.proposal_ref = P.proposal_id
6      AND      R.reviewer <> S.user
7      ) AS other_reviews,
8
9      (
10     SELECT   R.review_id AS bar_id, R.grade AS value
11     FROM     reviews R
12     WHERE    R.proposal_ref = P.proposal_id
13     ORDER BY R.grade DESC
14     ) AS grades,
15
16     (
17     SELECT   CASE
18         WHEN (D.mode = 'input') THEN 'input_tuple' :
19             SELECT
20                 D.comment, Tuple(
21                     D.grade_ref,
22                     ( SELECT * FROM grade_options )
23                     AS grade_options
24                 ) AS grade
25         ELSE 'display_tuple' :
26             SELECT  R.comment, R.grade
27             FROM    reviews R
28             WHERE   R.proposal_ref = P.proposal_id
29             AND     R.reviewer = S.user
30         END
31     FROM     draft_reviews D
32     WHERE    D.proposal_ref = P.proposal_id
33     AND      D.reviewer = S.user
34     ) AS my_review,
35
36     (
37     SELECT   AVG(R.grade)
38     FROM     reviews R
39     WHERE    R.proposal_ref = P.proposal_id
40     ) AS average_grade
41
42 FROM  proposals P, current_session S
43 WHERE EXISTS (
44     SELECT   *
45     FROM     assignments A
46     WHERE    A.proposal_ref = P.proposal_id
47     AND      A.reviewer = S.user
48 )
49 ORDER BY P.proposal_id
```

The query operates over four database tables: `proposals`, `assignments` to reviewers, `reviews` that have been submitted and `draft_reviews` that have been saved. It also operates over a special collection `current_session`, which provides a single tuple of HTTP session attributes.

Lines 1-49 is the outer query that produces the list of proposals. Lines 3-7 shows a sub-query that produces the set of reviews by reviewers other than the current user. It is a conventional SQL sub-query that is parameterized by tuple variables P and S from the outer query. By allowing nested queries in the `SELECT` clause the query language can construct results that have nested collections. The `bar_chart` sub-query (lines 9-14) is similar except for the `ORDER BY` clause, which makes the result a list instead of a set. The `my_review` query (16-34) features a SQL `CASE WHEN ... ELSE ... END` conditional expression that determines if a reviewer's review is an input tuple or display tuple based on whether the corresponding draft review is valid or not. The extension for heterogeneity allows the `CASE` expression to become a constructor for switch values, whenever each branch evaluates to a (potentially heterogeneous)

`case:value` pair. In general, various constructor functions / operators provide convenience syntax for creating values of the data model: another example is the tuple constructor on line 20. Lastly, the `average_grade` sub-query (lines 36-40) uses the `AVG` aggregation function to calculate the average review grade for a proposal, and the existential sub-query (lines 43-48) filters for proposals that have been assigned to the current user.

2.4 Visual layer

Units capture the logical structure of a page, including the program invocation requests that may be issued from it. Furthermore FORWARD units enable the incorporation of components from JavaScript libraries (such as Dojo [30] and YUI [35]) into the FORWARD framework.

Each page has a *unit tree* (comprising units), and each instance of a page has a corresponding *unit instance tree* (comprising unit instances) that conforms to the unit tree similar to how data trees conform to schema trees. Each unit has a *unit schema*, and a homomorphic mapping from *unit schema attributes* to nodes in the page schema tree. Intuitively, the unit schema captures the exported state of the unit and its descendants. The unit mapping induces a mapping from the corresponding unit instances to the nodes in the page data tree, which are termed the *unit instance's data*.

The FORWARD framework provides a textual template syntax for configuring a unit tree. For example, Figure 5 shows the unit tree for `Review Proposals`, with mappings to the page schema tree of Figure 4. Each XML element that is in the `unit` namespace encloses a *unit configuration*, which contains (1) XML elements in the default namespace for the unit schema attributes, and (2) nested unit configurations for children units. The template also allows HTML elements in the `html` namespace, thus a developer can configure all visual aspects of a page in a single unified syntax. The `bind` attribute is used to map unit schema attributes to schema nodes. For example, the (root attribute of the) `dropdown` unit maps to the `grade` tuple of Figure 4, and its `ref` and `options` attributes map respectively to the `grade_ref` and `grade_options` attributes.

A unit can be associated with one or more server-side programs. When a program is invoked, it has access to (1) the *invocation context*, which is the data node mapped from the unit instance of the program invocation (2) the data of form units, such as textboxes and dropdowns (3) the SQL database. Using these data, a program invocation issues `INSERT/UPDATE/DELETE` commands on the database, which are captured by the application's modification log. For example, the button unit in Figure 5 line 42 associates the `click` event with a `save_review` program. When the `save_review` program is invoked, it uses the corresponding proposal from the invocation context, the grade from `unit:dropdown`, the review from `unit:textbox`, and the current user from the session, and issues an `INSERT` command on the `reviews` table.

After program invocation, the page is incrementally refreshed in an efficient manner, the details of which will be fully described in Section 3.

```
1  <html:html>
2  <html:h1>List of all proposals.</html:h1>
3
4  <unit:table bind="page_state">
5
6      <column header="Title">
7          <unit:print bind="title />
8      </column>
9
10     <column header="Other reviews">
11         <unit:table bind="other_reviews">
12             <column header="Comment">
13                 <unit:print bind="comment" />
14             </column>
15             <column header="Grade">
16                 <unit:print bind="grade" />
17             </column>
18             <column header="Reviewer">
19                 <unit:print bind="reviewer_id" />
20             </column>
21         </unit:table>
22     </column>
23
24     <column header="grades">
25         <unit:barchart bind="grades" />
26     </column>
27
28     <column header="My Review">
29         <unit:switch bind="my_review">
30             <case bind="input_tuple">
31                 <html:div>
32                     <unit:dropdown bind="grade">
33                         <ref bind="grade_ref" />
34                         <options bind="grade_options">
35                             <option bind="grade_option" />
36                         </options>
37                     </unit:dropdown>
38
39                     <unit:textbox bind="comment" />
40
41                     <unit:button text="Submit"
42                      on_click="save_review"/>
43                 </html:div>
44             </case>
45
46             <case bind="display_tuple">
47                 <html:div>
48                     Comment:
49                     <unit:print bind="comment" />
50                     Grade:
51                     <unit:print bind="grade" />
52
53                     <unit:button text="Edit"
54                      on_click="edit_review" />
55                     <unit:button text="Remove"
56                      on_click="remove_review" />
57                 </html:div>
58             </case>
59         </unit:switch>
60     </column>
61
62     <column header="Avg. Grade">
63         <unit:print bind="average_grade" />
64     </column>
65
66 </unit:table>
67 </html:html>
```

Figure 5: Unit tree configuration

3. INCREMENTAL PAGE REFRESH

Pure server-side applications often suffer from long response time, due to the expensive recomputation at the data layer and the visual layer, and unfriendly user experience due to the browser blanking-out. Ajax solves both problems (see Section 1.1), but at the cost of significantly complicating web programming.

The following strawman approach employs Ajax to avoid the blanking out, while the programmer only provides a query that populates the report's data without having to provide separate queries and programs for incremental re-

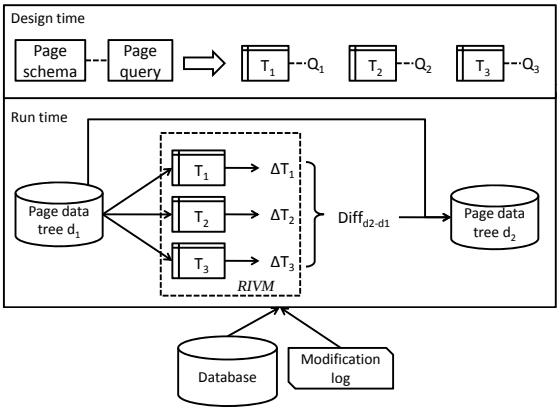

Figure 6: Page state computation

fresh. When a refresh of the page is needed, the page makes an asynchronous XHR JavaScript call that fetches the new page in its entirety from the server. The response handler replaces the old page DOM with the new one. The straw man approach achieves Ajax behavior only superficially. Compared to the pure server-side model, the server side computation stays the same, and the browser still needs to reinitialize all JavaScript components and re-render the entire page. Furthermore, users will still experience loss of focus, of cursor position and of data entered in non-submitted forms.

FORWARD combines the best of both the Ajax model and the pure server-side model by offering the development advantage of modeling pages as rendered views so that the developers need not specify any extra update logic, while the framework automates the incremental page refresh in both the data layer and the visual layer to achieve the Ajax performance and preservation of focus, scroll, cursor positions and form data. This section describes how incremental page refresh is handled in the data layer (Section 3.1) and the visual layer (Section 3.2).

3.1 Leveraging and extending incremental view maintenance

The Page State Computation Module (PSC) of FORWARD (see Figure 2) treats the page data tree as a view. During page refresh it uses the log of modifications that happened to the database data, and possibly the database data itself to incrementally maintain the old view instance to the new view instance.

Figure 6 shows an overview of how PSC incrementally maintains the page data tree d_1 of the old page state s_1 to the page data tree d_2 of the new page state s_2. Recall that a page data tree is computed as the result of the page query, which is a nested query in FORWARD's extended SQL language. At design time, PSC decomposes the nested page schema into flat relational views denoted by T_1, T_2 etc, and rewrites the page query into standalone SQL queries q_1, q_2 etc that define the flat views respectively. At run time, the old page data tree d_1 is first transformed to instances of the flat relational views. Then PSC uses a Relational Incremental View Maintenance algorithm (*RIVM*) on each flat view utilizing the modification log and possibly database data. The incremental changes to the flat views computed by *RIVM* are translated to the data difference $Diff_{d_2-d_1}$ that can be combined with the old page data tree d_1 to calculate the new page data tree d_2. The current implementation of *RIVM* in PSC is built on top of an off-the-shelf relational database without modification to the database engine. The framework monitors all data modification in an application to maintain the modification log, and expands the log to capture data changes of each database table. Both the modification log and page data trees are stored in main memory resident tables of DBMS for fast access.

The data difference $Diff_{d_2-d_1}$ is encoded as the series of data modification commands that turn d_1 into d_2. The data modification commands are insert, remove and update. The remove command $remove(t)$ locates the node identified by t in a data tree and removes the subtree rooted at it. The update command $update(t, N)$ replaces the old subtree rooted at t with the new subtree N. The insert command has a set version and a list version for the two different types of collections. The set-version $insert_s(r, N)$ inserts a subtree N rooted at a tuple into the targeted set identified by r. The list-version $insert_l(r, N, t_0)$ takes one more parameter t_0 which is the adjacent node after which the new subtree should be inserted.

Section 3.1.1 lists the benefits of PSC for page refresh using an example. Section 3.1.2 provides background on relational techniques that are leveraged. Section 3.1.3 describes how a page schema can be decomposed into flat views, and how to obtain the SQL queries that define each flat view. Section 3.1.4 discusses the handling of order through the view maintenance process. Finally, Section 3.1.5 describes how maintenance results on flat views can be translated and applied to the nested data tree.

3.1.1 Benefits and example

PSC drastically reduces the number of SQL queries that must run in order to refresh the page by detecting the following opportunities:

- *Non-modification:* PSC may statically prove that certain data tree nodes are unaffected by the data modifications. Therefore data of these nodes need not be recomputed.
- *Page state self-sufficiency:* In this case, d_2 can be computed as a function of d_1 and the modification log, without access to the proper database tables. Since PSC stores the modification log and d_1 in main memory, fast computation is achieved by avoiding disk access. Furthermore, as the OLAP and view maintenance literature has shown self-sufficiency opportunities can be greatly increased by the inclusion of a small amount of additional data in a view. For example, an average view is not self-maintainable if it only has the averages but it is maintainable if it also has the count.
- *Incremental maintenance:* When the previous two cases are not available, PSC may need to access database tables to compute $Diff_{d_2-d_1}$ and consequently d_2. However, computing $Diff_{d_2-d_1}$ is usually faster than running the page query from scratch, with the help of modification log and the cached old page data set.

In practice, PSC utilizes more than one opportunities from above to maintain a page data tree, with each applied to different parts of the page. Suppose the current reviewer Ken updates the grade and the comment of review `#2` of proposal `509` of Figure 1. Therefore the modification log includes (a) an update to the comment and rating of the (`509`, `#2`) tuple in the `reviews` table, and (b) an update of the mode value

of the (509, #2) tuple in the draft_review table. Suppose that the modification log also happens to have the following changes by other users which happened right before the submission of the update by Ken: (c) an insertion of new review #3 for proposal 509, (d) an update of a review of another proposal 456, and (e) an insertion of a recommendation on another page. Notice that proposal 456 does not appear on the page since it is not assigned to Ken. Given the modification log PSC can statically determine that the change (e) does not affect the review page (i.e., the non-modification case) since the current page does not show recommendations at all. It also determines that the change (d) does not affect the page because proposal 456 does not appear in the page data tree shown to Ken. The other changes in the modification log correspond to either the page state self-sufficiency case or the incremental maintenance case. In particular, because of (a) and (b), the input tuple (form) at proposal 509 will disappear since the switch node will revert to the display case, and the display tuple at proposal 509 will be set according to the grade and review submitted by Ken. A new review tuple is inserted into the list of other reviews for (c). Finally, because of (a) and (c), the average can be incrementally recomputed from the old average, the count and the modifications, if the count is also included in the view as additional data.

3.1.2 Leveraging relational research

The relational model literature [6, 19, 17, 18, 26] has described methods for efficiently maintaining a materialized SQL view $V = q(R_1, R_2, \ldots, R_n)$, where q is a SQL query and $\{R_1, R_2, \ldots, R_n\}$ are base tables. One approach implemented by many existing solutions and also by PSC in FORWARD is to model data changes as *differential tables*. Let the old view instance be V_1 and the new view instance be V_2. Between V_1 and V_2, the differential tables for a base relation R_i are Δ^+R_i containing tuples inserted into R_i, and Δ^-R_i containing tuples deleted from R_i. Δ^+R_i and Δ^-R_i are captured in the modification log. In the same way Δ^+V and Δ^-V can be defined. Tuple update is treated as a combination of insertion and deletion. The view maintenance algorithm *RIVM* in this approach runs two queries q_{Δ^+} and q_{Δ^-} over $\{R_1, \cdots, R_n, \Delta^+R_1, \cdots, \Delta^+R_n, \Delta^-R_1, \cdots, \Delta^-R_n\}$ that produce Δ^+V and Δ^-V respectively.

PSC focuses on the deferred view maintenance [9, 27] that works with after-modification database tables and the modification log. The reason is that in data-driven web applications, although a data modification can affect data trees seen by multiple users, the page view maintenance for a user can be deferred until the user requests for the page again, so that the system throughput can be maximized. PSC implements *RIVM* on top of the open source PostgreSQL, which does not have native support of materialized view. In general, other implementations of *RIVM* can be used in PSC as well.

3.1.3 Reduction of nested queries and switches

PSC uses *RIVM* as a building block to manage nesting and switches. It transforms a nested page schema into flat relational views $T_1, \cdots, T_n$, and the corresponding page query into SQL queries $q_1, \cdots, q_n$, such that each T_i is defined by q_i.

Given a page query q and corresponding page schema V, PSC takes the first step to create flat relations $S_1, \cdots, S_n$ as follows to represent the decomposition of V with respect to q. The outer-most collection of V is represented as the relation S_1. Each sub-query in the SELECT clause is mapped to a new relation. Each case sub-query in CASE WHEN conditional statements is also mapped to a new relation. Notice that sub-queries in the WHERE clauses are unaffected. Let the corresponding sub-query for each S_i be p_i. If p_a is the parent (sub-)query of p_b, then expand S_b to contain the foreign key attributes referencing tuples in S_a, and call S_a the parent of S_b. Finally, the primary key attributes of each S_i are made to include the foreign key attributes to its parent relation, if it exists, in addition to the S_i's original primary key attributes.

At this point, each flat relation S_i after decomposition corresponds to a sub-query p_i that may use values from its ancestor sub-queries. PSC modifies each S_i to get flat view T_i and creates its defining query q_i based on p_i, so that each q_i is a standalone SQL query without correlation. First, each T_i is designed to have S_i's schema and also to contain the attributes whose values are referred by any p_k that is a descendant of p_i. Then the defining query q_i of each flat view T_i is modified from p_i by adding T_j, where p_j is the parent of p_i, as an additional input table in p_i's WHERE clause. The original references to values in p_j can then be changed in q_i to the joined attributes from T_j. Finally, in order to ease the discussion, we still call that a flat relation T_a is the parent of T_b if p_a is the parent of p_b.

During run time, PSC traverses q_i from top down to incrementally maintain each T_i as follows: First for the top level view T_1 defined by q_1, PSC runs Δ^+T_1 and Δ^-T_1 using *RIVM*. If T_a is the parent of T_b, when T_b is maintained by PSC, its parent table T_a would have already been maintained, so that Δ^+T_a and Δ^-T_a are available, which is necessary since T_a is an input relation to T_b's definition q_b.

For example, the following SQL statements defines some of the flat views corresponding to the result of relational decomposition of the page schema and the page query in the running example.

```
CREATE VIEW T1_proposals AS
SELECT   R.proposal_id, P.title, S.user
FROM     proposals P, current_session S
WHERE    EXISTS (
    SELECT   *
    FROM     assignments A
    WHERE    A.proposal_ref = P.proposal_id
    AND      A.reviewer = S.user
)
ORDER BY P.proposal_id

CREATE VIEW T2_other_reviews AS
SELECT   R.comment, R.grade, R.review_id, T1.proposal_id
FROM     reviews R, T1_proposals T1
WHERE    R.proposal_ref = T1.proposal_id
AND      R.reviewer <> T1.user

CREATE VIEW T3_average_grade AS
SELECT   T1.proposal_id, AVG(R.grade)
FROM     reviews R, T1_proposals T1
WHERE    R.proposal_ref = T1.proposal_id
GROUP BY T1.proposal_id
```

Consider the modifications described in Section 3.1.1. Since neither proposals nor current_session is changed between the previous and the current page states, T_1 is not changed and Δ^+T_1 and Δ^-T_1 are empty. Because change (c) brings a non-empty $\Delta^+R_{\text{reviews}}$, *RIVM* is able to compute Δ^+T_2 by joining $\Delta^+R_{\text{reviews}}$ and T_1 and then doing the selection. The

defining queries of all the decomposed flat views of the running example can be incrementally maintained by *RIVM*.

3.1.4 Lists and reordering

Since most prior work on relational view maintenance assume bag or set semantics only, PSC is extended to support ordered list semantics by embedding order information as data and simulating list-version operators using order-insensitive ones.

The support of list in the data model of FORWARD allows list-version operators in the query language's algebra like the FLWR in XQuery where the inputs, outputs and intermediate results are treated as lists. Many of these operators only need to preserve the order of tuples from the input to the output, such as the list-version selection and projection. For the view maintenance purpose, such operators can be simulated by their relational counterparts, with order information embedded as data inside the *order specifying attributes*. For example, a list can be encoded as a set of tuples $\{(1, \text{tom}), (2, \text{ken}), (3, \text{jane})\}$ with the auxiliary first attribute being the order specifying attribute. In practice, such system-generated attributes use encodings like LexKey described in [10] in order to support efficient repositioning. In this way, these operators can treat order like data and need not explicitly maintain it.

The ORDER BY operator that creates order is handled by PSC by statically marking the order-by attributes as the order specifying attributes. At run time, only the inserted data changes are sorted, while the reordering of the entire view is deferred until the final result, where the size of data is usually small as limited by the nature of a web page. Order-sensitive operators, such as Top-k and MEDIAN, are often expensive to maintain incrementally. For example, a deletion of tuple from the input list of a Top-k operator may incur scanning of the input list if the deleted tuple was among the top k tuples. Maintenance of Top-k views has been studied in [34]. How the embedded order information is restored when the modification is applied to the nested view is discussed next.

3.1.5 Updating the page data tree

After the data changes Δ^+T_i and Δ^-T_i are obtained for each T_i, the view maintenance result of the flat views are translated to $Diff_{d_2-d_1}$ as a series of data modification commands and then applied to the old page data tree d_1 to obtain the new data tree d_2. The changes to different T_i are applied in a top-down order, so that when changes to a child data node is applied, its parent data node is guaranteed to exist. Since every T_i has primary key attributes defined to contain ancestors' primary keys in the corresponding data tree, it is simple to navigate in the data tree to locate the target of each change in Δ^+T_i and Δ^-T_i. Notice that a relation in the data tree can be either a list or a set. If it is a list, tuples from Δ^+T_i need to be translated into list-version insert commands of the data tree which require adjacent tuples to be specified. Such adjacent tuples can be located efficiently by using binary-search over the order specifying attributes because the previous sorted list is materialized and cached as part of the data tree d_1. The handling of other cases is straightforward.

3.2 Incremental maintenance of the visual layer

Given the data layer difference $Diff_{d_2-d_1}$ from the PSC, the incremental maintenance visual layer (IMVL) refreshes the page through unit instances that translate the data layer difference into updates of DOM elements and JavaScript components. The IMVL is based on the observation that the browser page state can be divided into fragments, where each fragment corresponds to the rendering of a unit instance, which in turn depends on one or more corresponding data tree nodes. Only a unit instance corresponding to an updated data tree node needs to be re-rendered.

Incremental maintenance of unit instance tree Incremental maintenance of the page is facilitated by the unit instance tree, which is a data structure residing on the browser. Each unit instance maintains pointers to its underlying DOM elements and JavaScript components, so that only pertinent elements / components are re-rendered. From the data layer difference, which is encoded as a sequence of insert, update and delete commands on the page data tree, the IMVL uses the unit tree to produce *unit instance differences*, which are corresponding encodings for each unit instance. Each insert command that spans multiple units will be fragmented into insert commands for the respective unit instances; similarly so for each delete command. Each update command that spans multiple units will be fragmented into an update command for the top unit instance, delete commands for existing descendant unit instances, and insert command for new descendant unit instances. When initializing a new page instance, the IMVL will create the unit instance tree from scratch. However, given an existing page instance, the IMVL will use the unit instance differences to incrementally maintain the unit instance tree, in order to preserve existing DOM elements and JavaScript components.

Incremental rendering of units With an updated unit instance tree, the IMVL will invoke in turn each unit instance's *incremental renderer* (or *renderer*), which translates the unit instance difference into updates of the underlying DOM element or method calls of the underlying JavaScript component. Note that these renderers are implemented by unit authors, and are automatically utilized by the framework without any effort from the developer. Essentially, renderers modularize and encapsulate the partial update logic necessary to utilize JavaScript components, so that developers do not have to provide such custom logic for each page.

Mediating between unit differences and JavaScript components Consider the number of possible combinations for a unit instance difference: (1) any of the unit schema's attributes can be the root of the data diff (2) the data diff can be encoded as any of the three insert, update and delete commands. For each $(attribute, command)$ pair, a unit can be associated with a renderer. Since FORWARD units utilize components from existing JavaScript libraries, the number of possible renderers typically exceed that of available refresh methods on components. Therefore, given a unit difference, if the most specific renderer for the $(attribute, command)$ pair is not implemented, the framework will attempt to simulate it on a best-effort basis with other available renderers. Any renderer can be simulated by a `update` renderer of an ancestor attribute, while an `update` renderer on a tuple can also be simulated by a combination of `insert` and `delete` renderers on the same tuple. Minimally, a unit needs to be associated with an `insert` and `delete` renderer on the unit schema root attribute.

For example, consider the bar chart unit used on the `Review Proposals` page, and a reviewer modifying his grade on a review. If the underlying JavaScript component supports changing the value of a particular bar, and a `update` renderer has been implemented for the `value` attribute, the bar chart will be incrementally refreshed where only the affected bar grows/shrinks. Otherwise, the entire bar chart has to be refreshed. Implementing specific renderers improves performance for units that are expensive to initialize (e.g. a map unit), and avoids overwriting user-provided values in units that collect values (e.g. a textbox).

4. IMPLEMENTATION AND EVALUATION

FORWARD operates as an interpreter of an application specification, with static analysis taking place the first time an application is loaded by the system. A proof-of-concept prototype has been implemented as a Java servlet running in the Jetty servlet container. Queries are parsed and translated into conventional SQL statements, which are executed in a PostgreSQL relational database.

To illustrate the performance characteristics of the prototype, we consider the running example where the database stores 20,000 proposals, each proposal has 6 reviews, the page displays 20 proposals, and a reviewer submits a revision to his review for one displayed proposal. Only two other database modifications have been made since the reviewer's page was last refreshed: one for them is a review update (by another reviewer) for the same proposal, whereas the other is a review update for a proposal that is not displayed on the current page. Consequently, the page refreshes with two additional reviews on the page.

All measurements are performed on an Intel Core 2 Quad 2.7 GHz desktop running Windows Vista 64-bit. The server runs under Java VM 1.6 under server mode, whereas pages are loaded in the Firefox 3.5 browser. Since the JVM's JIT compiles hotspot bytecode into native code based on runtime profiling of multiple method invocations, the initial 20-30 readings for each experiment are discarded until steady readings can be obtained, in order to approximate a long-running server. The average of 10 readings are then taken to smooth out CPU spikes from the JVM's garbage collection. To simulate a database server where proposals are not already cached in memory buffers, the currently logged-in user (and hence the proposals retrieved) is randomly selected for each reading.

For network measurements, the servlet container enables gzip compression by default, which accounts for an order of magnitude reduction in response size. To estimate the time needed for real network traffic, we assume a coast-to-coast network round-trip time of 100 ms [20], and the average US upload and download bandwidths of 1 Mbps and 5 Mbps respectively [24].

To demonstrate the end-to-end performance of a server roundtrip, Table 1 presents itemized activities and their latencies in the strawman implementation described in the beginning of Section 3, starting from the time the submit button is clicked till the browser fully refreshes. Table 2 presents the same activities and their latencies in FORWARD, which employs the incremental maintenance techniques of Section 3.

In Table 1, (1) is the time spent by JavaScript code in the browser collecting the data for the invocation context, (2-3) is the time to transmit the request, and (4) is the time to invoke the program for updating the review in the database. Note that (1-4) are outside the scope of the incremental maintenance optimizations, and therefore have identical values in Table 2. (5) is the time to evaluate the query to generate the page data tree. Indexes have been created on foreign key columns so that PostgreSQL can efficiently join tables, but the query is expensive due to the disk accesses incurred. (6) is the time to encode the entire page data tree in JSON. (7-8) is network time. Finally (9) is the time to create a new unit instance tree and render the DOM elements and JavaScript components from scratch. We omitted browser reflow time, as browsers do not provide programmatic mechanisms to reliably measure it, and the reflow time for the running example is too fast to be measured manually with a stopwatch.

	System	Description	Time	Size
1	Browser	Invoke request	14 ms	
2	Network	Request latency	50 ms	
3		Request transfer time	2 ms	0.2 KB
4	Server	Update review	5 ms	
5		Generate page data tree	210 ms	
6		Response I/O	13 ms	
7	Network	Response latency	50 ms	
8		Response transfer time	9 ms	6 KB
9	Browser	Rendering	38 ms	
		Total	**391 ms**	

Table 1: Strawman implementation

	System	Description	Time	Size
1	Browser	Invoke request	14 ms	
2	Network	Request latency	50 ms	
3		Request transfer time	2 ms	0.2 KB
4	Server	Update review	5 ms	
5		View maintenance	7 ms	
6		Response I/O	5 ms	
7	Network	Response latency	50 ms	
8		Response transfer time	1 ms	0.4 KB
9	Browser	Incremental rendering	8 ms	
		Total	**142 ms**	

Table 2: FORWARD implementation

In Table 2, note that (4) remains the same as in Table 1, showing that storing the modification log in main memory has no measurable performance penalty. (5) demonstrates the efficacy of the incremental view maintenance of Section 3.1. Since there are no proposal updates in the modification log, the `proposals` collection falls in the non-modification case, therefore no SQL queries need to be issued. Other nested collections, such as `other_reviews` and `grades`, fall in the incremental maintenance case, where both the modification log and database need to be accessed to compute $\texttt{proposals} \bowtie \Delta^{+}\texttt{reviews}$. As compared to the full query which requires a join on the 120,000-row `reviews` table, the incremental query uses the 3-row modification log to yield a 30x speed up. (6) shows that the data layer difference is more efficient to encode than the entire page data tree. Similarly (8) shows that the data layer difference is also more efficient to transmit. Lastly, (9) shows that incremental maintenance of the visual layer (Section 3.2) produces a 4.75x speed up. The speed up can be attributed to less DOM elements being created and less JavaScript components being

initialized. In addition to the speed up, FORWARD's incremental refresh preserves values that the user may have entered in other form units, thus providing a user experience superior to that of the strawman implementation's full refresh.

5. RELATED WORK

The data management research community has created database-driven frameworks for web site [12] and pure server side web application [8, 33] development. In WebML [8] the unit structure of a page tracks the database's E/R schema and it is easy to create pages that report/update entities and navigate across them. While these frameworks do not work with Ajax components, they still provide an important target, which FORWARD pursued: maintain their clarity of specifying applications despite the fact that Ajax applications require distributed programming, multiple languages and tedious combination of component initialization and refresh. Generally browser side code was neglected (except for the recent [14] that describes how to run XQuery on the browser).

Echo2 [11], ZK [38], Backbase [4] and ICEfaces [21] are Ajax frameworks that also provide to the programmer the ease of programming in a single language (typically Java) and exclusively at the server. They mirror the page state by caching it in its entirety on the server and they keep the browser and server page states in sync automatically. However, since the languages of these frameworks are imperative (instead of FORWARD's SQL-based language), they cannot perform automatic incremental maintenance of the page. Therefore one has to program both for the initialization and the refresh of the components. To the best of our knowledge FORWARD is the only framework that employs automatic incremental maintenance of the page.

In the same spirit Microsoft's ASP.NET [31] is a pure server-side framework that provides mirroring of page state by always sending the page state from the browser to the server in a hidden form field. It shares a drawback with the Ajax frameworks listed above in that the page state includes styling properties and implementation details and therefore it has a high memory footprint and slow mirroring. For example, our measurements have shown that an Echo2 page for the page of Figure 1 occupies about 300KB, or three times more memory. FORWARD's structuring of the page across the MVC architecture and the sending of the forms parts of the page data tree is obviously sufficient and more efficient.

Google's GWT [16] and Cornell's Hilda [32] achieve the single language property with the same fundamental technique: they distribute the processing between the browser and the server. In GWT's case this is accomplished by translating Java (which is the single programming language) into JavaScript. We believe that the high engineering complexity of distribution is unnecessary since mirroring can be very performant, as we showed.

For use in pre-Ajax web application infrastructure, [7] shows how to manage cached dynamic pages by invalidating out-dated views in the cache upon relevant updates to the base tables. Ajax provides a finer-grained opportunity, which FORWARD exploits: Instead of invalidating the whole page, incrementally update its invalidated parts.

Relational incremental view maintenance received high attention in the mid 90s, in the context of efficient data warehouse maintenance (for example, see [37, 2, 23]). More recently, [1, 36, 10, 3, 28, 13] proposed solutions to the view maintenance problem for query languages and data models that support nesting and ordering. However, these techniques have limited applicability for FORWARD as they specialize in immediate view maintenance only, do not support the sets of required update operations or apply to less expressive query languages.

6. CONCLUSIONS AND FUTURE WORK

FORWARD allows the development of Ajax data-driven pages by declaratively describing their data (using an appropriately extended SQL) and consequently rendering them in FORWARD's page unit structure. The pages are treated as automatically refreshed rendered views. We showed that the rendered views approach increases productivity since the page can be succinctly expressed with a combination of SQL and (visual) page units while the mundane data synchronization issues of the page are automatically resolved.

At the core of the described solution has been the "data-driven page" assumption, which technically means that the server state is effectively fully captured by its database state and the page's data at a logical level can be described with a query over the database state. Notice that a partial failure of this assumption does not lead to a wholesale dismissal of our approach. Rather, the developer can use FORWARD just for the parts of pages that are data-driven while he will need to resort to conventional Ajax programming techniques for the rest.

The larger task of simplifying Ajax web application programming entails a number of additional challenges, described next, emerging once we remove the data-driven page assumption. Some of those challenges will require additional research often at the intersection of software engineering and data management. Others (notably integration issues) will keep being resolved by the developers' code in practice.

Extending to non-data-driven pages An extension to the rendered view paradigm may need to be taken when the user interaction on the Ajax page leads to changes on the page itself and such page changes are most succinctly expressed as a direct function of the user interaction. One way, but not always the best, to accomplish such page changes is to first turn the user interaction's effects into database insert/delete/updates so that the page view can be automatically and incrementally maintained by capturing them. Another direction towards addressing such cases is the extension of FORWARD to allow the page queries to (a) use the current page state also as an input database and (b) extend the page query semantics to allow the expression of a partial modification of itself. We believe that an extension in this direction can bridge the rendered view paradigm with a style of programming based on explicitly specifying the effect of users' interactions on parts of the page.

Integration between the relationally-driven page and server side OO components If the server state includes objects that either do not have an underlying database or they are exported by OO components that encapsulate their underlying database (therefore making it unavailable to the SQL query fueling the rendered view) a conventional integration problem of objects to relational data emerges. Part of the problem is mitigated by the fact that the FORWARD page schemas already include nesting and variability. Nevertheless, we need to work to provide tools, in the spirit

of object-relational mappers such as Hibernate, to facilitate this integration, while keeping in mind that the long history of the OO/relational interfacing problem indicates that a magic bullet is not found and developers will need to apply the best practices and methodologies of OO/relational integration in this domain.

7. REFERENCES

[1] S. Abiteboul, J. McHugh, M. Rys, V. Vassalos, and J. L. Wiener. Incremental maintenance for materialized views over semistructured data. In *VLDB*, pages 38–49, 1998.

[2] D. Agrawal, A. E. Abbadi, A. K. Singh, and T. Yurek. Efficient view maintenance at data warehouses. In *SIGMOD*, pages 417–427, 1997.

[3] M. A. Ali, A. A. A. Fernandes, and N. W. Paton. Movie: An incremental maintenance system for materialized object views. *Data Knowl. Eng.*, 47(2):131–166, 2003.

[4] Backbase enterprise ajax framework, 2009. `http://www.backbase.com/products/enterprise-ajax/`.

[5] F. Bancilhon, S. Cluet, and C. Delobel. A query language for the o_2 object-oriented database system. In *DBPL*, pages 122–138, 1989.

[6] J. A. Blakeley, P.-Å. Larson, and F. W. Tompa. Efficiently updating materialized views. In *SIGMOD*, pages 61–71, 1986.

[7] K. S. Candan, D. Agrawal, W.-S. Li, O. Po, and W.-P. Hsiung. View invalidation for dynamic content caching in multitiered architectures. In *VLDB*, pages 562–573, 2002.

[8] S. Ceri, P. Fraternali, and A. Bongio. Web modeling language (webml): a modeling language for designing web sites. *Computer Networks*, 33(1-6):137–157, 2000.

[9] L. S. Colby, T. Griffin, L. Libkin, I. S. Mumick, and H. Trickey. Algorithms for deferred view maintenance. In *SIGMOD*, pages 469–480, 1996.

[10] K. Dimitrova, M. El-Sayed, and E. A. Rundensteiner. Order-sensitive view maintenance of materialized xquery views. In *ER*, pages 144–157, 2003.

[11] Echo web framework, 2009. `http://echo.nextapp.com/site/`.

[12] M. F. Fernández, D. Florescu, A. Y. Levy, and D. Suciu. Declarative specification of web sites with strudel. *VLDB J.*, 9(1):38–55, 2000.

[13] J. N. Foster, R. Konuru, J. Siméon, and L. Villard. An algebraic approach to view maintenance for xquery. In *PLAN-X*, 2008.

[14] G. Fourny, M. Pilman, D. Florescu, D. Kossmann, T. Kraska, and D. McBeath. Xquery in the browser. In *WWW*, pages 1011–1020, 2009.

[15] J. J. Garrett. Ajax: A new approach to web applications. `http://adaptivepath.com/ideas/essays/archives/000385.php`, February 2005. [Online; Stand 18.03.2008].

[16] Google widget toolkit, 2009. `http://code.google.com/webtoolkit/`.

[17] T. Griffin and L. Libkin. Incremental maintenance of views with duplicates. In M. J. Carey and D. A. Schneider, editors, *SIGMOD*, pages 328–339. ACM Press, 1995.

[18] A. Gupta and I. S. Mumick. Maintenance of materialized views: Problems, techniques, and applications. *IEEE Data Eng. Bull.*, 18(2):3–18, 1995.

[19] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In P. Buneman and S. Jajodia, editors, *SIGMOD*, pages 157–166. ACM Press, 1993.

[20] M. A. Habib and M. Abrams. Analysis of sources of latency in downloading web pages. In *WebNet*, pages 227–232, 2000.

[21] Icefaces, 2009. `http://www.icefaces.org/main/home/`.

[22] B. Johnson. Reveling in constraints. *Queue*, 7(6):30–37, 2009.

[23] I. S. Mumick, D. Quass, and B. S. Mumick. Maintenance of data cubes and summary tables in a warehouse. In *SIGMOD*, pages 100–111, 1997.

[24] C. W. of America. 2009 report on internet speeds in all 50 states, 2009. `http://cwafiles.org/speedmatters/state_reports_2009/CWA_Report_on_Internet_Speeds_2009.pdf`.

[25] C. O'Hanlon. A conversation with werner vogels. *Queue*, 4(4):14–22, 2006.

[26] K. A. Ross, D. Srivastava, and S. Sudarshan. Materialized view maintenance and integrity constraint checking: Trading space for time. In *SIGMOD*, pages 447–458, 1996.

[27] K. Salem, K. S. Beyer, R. Cochrane, and B. G. Lindsay. How to roll a join: Asynchronous incremental view maintenance. In *SIGMOD*, pages 129–140, 2000.

[28] A. Sawires, J. Tatemura, O. Po, D. Agrawal, and K. S. Candan. Incremental maintenance of path expression views. In *SIGMOD*, pages 443–454, 2005.

[29] L. Simon. Minimizing browser reflow. `http://code.google.com/speed/articles/reflow.html`, June 2009.

[30] The dojo toolkit, 2009. `http://www.dojotoolkit.org/`.

[31] Wikipedia. Asp.net, 2009. Accessed Nov 04 2009. `http://en.wikipedia.org/w/index.php?title=ASP.NET&oldid=323456166`.

[32] F. Yang, N. Gupta, N. Gerner, X. Qi, A. J. Demers, J. Gehrke, and J. Shanmugasundaram. A unified platform for data driven web applications with automatic client-server partitioning. In *WWW*, pages 341–350, 2007.

[33] F. Yang, J. Shanmugasundaram, M. Riedewald, and J. Gehrke. Hilda: A high-level language for data-drivenweb applications. In *ICDE*, page 32, 2006.

[34] K. Yi, H. Yu, J. Yang, G. Xia, and Y. Chen. Efficient maintenance of materialized top-k views. In *ICDE*, pages 189–200, 2003.

[35] Yui library, 2009. `http://developer.yahoo.com/yui/`.

[36] Y. Zhuge and H. Garcia-Molina. Graph structured views and their incremental maintenance. In *ICDE*, pages 116–125, 1998.

[37] Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom. View maintenance in a warehousing environment. In *SIGMOD*, pages 316–327, 1995.

[38] Zk direct ria, 2009. `http://www.zkoss.org/`.