UC San Diego
UC San Diego Electronic Theses and Dissertations

Title
Web application creation made easy : a SQL-driven rapid development framework and a Do-It- Yourself platform

Permalink
https://escholarship.org/uc/item/1j06g6p9

Author
Ong, Kian Win

Publication Date
2010

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA, SAN DIEGO

Web Application Creation Made Easy: A SQL-driven Rapid Development Framework and a Do-It-Yourself Platform

A dissertation submitted in partial satisfaction of the
requirements for the degree
Doctor of Philosophy

in

Computer Science

by

Kian Win Ong

Committee in charge:

Professor Alin Deutsch, Chair
Professor Vish Krishnan
Professor Ingolf Krueger
Professor Bertram Ludäscher
Professor Yannis Papakonstantinou

2010

Copyright
Kian Win Ong, 2010
All rights reserved.

The dissertation of Kian Win Ong is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

____

____

____

____

____
Chair

University of California, San Diego

2010

TABLE OF CONTENTS

Signature Page iii

Table of Contents iv

List of Figures vi

List of Tables viii

Acknowledgements ix

Vita x

Abstract of the Dissertation xi

Chapter 1 Introduction 1
1.1 Rapid Application Development Framework 2
1.2 Do-It-Yourself (DIY) Platform 4
1.3 Roadmap 6

Chapter 2 Rapid Application Development Framework 7
2.1 Introduction 7
2.1.1 Ajax background 8
2.1.2 Framework and language contributions 12
2.1.3 System and optimization contributions 14
2.2 The FORWARD framework and scope 15
2.2.1 Running example 15
2.2.2 Architecture 16
2.2.3 Data layer 18
2.2.4 Visual layer 21
2.3 Incremental page refresh 24
2.3.1 Leveraging and extending incremental view maintenance 25
2.3.2 Incremental maintenance of the visual layer . . . 33
2.4 Evaluation 35
2.5 Related work 38
2.6 Summary and Future Work 39

Chapter 3 Data Model 42
3.1 Syntax 44
3.2 Type System 46
3.2.1 Data Trees 46
3.2.2 Constraints 49

3.2.3 Primary keys & Context 52
3.2.4 Data Consistency 55
3.3 Data Diffs 57

Chapter 4 Query Language 60
4.1 Syntax Analysis 62
4.2 Semantic Analysis 68
4.2.1 Variable reference checking 68
4.2.2 Type checking / inference 70
4.2.3 Group by checking 71
4.2.4 Primary key checking / inference 71
4.3 Query Evaluation 72
4.3.1 Decomposition of input data 72
4.3.2 Relational decomposition of query 78
4.3.3 Evaluating the generators 83
4.3.4 Reconstructing the nested output data tree 84
4.4 Data Sources 86
4.4.1 Local versus Remote Data Sources 86
4.4.2 Data Source Catalog 88

Chapter 5 Incremental View Maintenance 90
5.1 Delta Tuples 90
5.2 Leveraging Relational Incremental View Maintenance . . 92
5.3 Rewrite Rules to Create Incremental Queries 93
5.4 Evaluation of incremental query 108
5.5 Reconstruction of the new view 114

Chapter 6 Do-It-Yourself Platform 116
6.1 Introduction 116
6.2 Framework and Scope 124
6.2.1 Reports 125
6.2.2 Contextual Actions 127
6.2.3 User Group Definitions 129
6.3 Do-It-Yourself Design Facility 129
6.3.1 Derived Properties 130
6.3.2 Page-Driven Design 131
6.3.3 Workflow-Driven Design 137
6.3.4 Automated Report Creation 141
6.4 Related Work 147

Bibliography 152

LIST OF FIGURES

Figure 2.1: `Review Proposals` page 15
Figure 2.2: FORWARD architecture 16
Figure 2.3: Page data tree 18
Figure 2.4: Page schema 18
Figure 2.5: Unit tree configuration 23
Figure 2.6: Page state computation 26

Figure 3.1: BNF for data tree 44
Figure 3.2: BNF for schema tree 44
Figure 3.3: Example data tree 45
Figure 3.4: Example schema tree 45
Figure 3.5: Grammar for Data Tree 46
Figure 3.6: Grammar for Schema Tree 47
Figure 3.7: Grammar for Constraints 49
Figure 3.8: Example top-level collection 52
Figure 3.9: Context 54
Figure 3.10: Original Data Tree 58
Figure 3.11: Data Diffs 58

Figure 4.1: Syntax of query (`SELECT` clause) 63
Figure 4.2: Syntax of query (expressions) 64
Figure 4.3: AST (queries) 65
Figure 4.4: AST (`SELECT` clause) 66
Figure 4.5: AST (expressions) 67
Figure 4.6: Nested schema before decomposition 74
Figure 4.7: Schema after normalization 75
Figure 4.8: Schema after concatenation (decomposed schema) 76
Figure 4.9: Data and Schema Tree for Local / Remote Data Sources 88

Figure 5.1: Example base relations 111
Figure 5.2: Example decomposed old view 111
Figure 5.3: Example flat deltas of base relations 112
Figure 5.4: Example generator tree with incremental queries 112
Figure 5.5: Example incremental view deltas 113

Figure 6.1: `Submit Startup` Page 119
Figure 6.2: `Evaluate Startups` Page 119
Figure 6.3: `Advisor Comments` Page 120
Figure 6.4: TechCrunch50 Workflow Visualization
(in ppt; see video demo for the actual one) 123
Figure 6.5: Page Wizard for `Submit Startup` Page 132
Figure 6.6: WYSIWYG Page Design 134

Figure 6.7: Workflow-Driven Design 139
Figure 6.8: Automated Report Extension on `Evaluate Startups` Page . . 142

LIST OF TABLES

Table 2.1: Strawman implementation 36
Table 2.2: FORWARD implementation 36

ACKNOWLEDGEMENTS

Chapter 2 contains material from "Ajax-based Report Pages as Incrementally Rendered Views", by Yupeng Fu, Keith Kowalczykowski, Kian Win Ong, Kevin Keliang Zhao and Yannis Papakonstantinou, which appears in Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010. The dissertation author was the primary investigator and author of this paper. The material in this chapter is copyright ©2010 by the Association for Computing Machinery, Inc. (ACM). Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page in print or the first screen in digital media. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Publications Dept., ACM, Inc., fax +1 (212) 869-0481, or email `permissions@acm.org`.

Chapters 3, 4 and 5 contain material from the technical report "Technical Specifications of the FORWARD Framework", by Yupeng Fu, Kian Win Ong, Kevin Keliang Zhao, Yannis Papakonstantinou and Michalis Petropoulos. The dissertation author was the primary investigator and author of the relevant chapters in the technical report.

Chapter 6 contains material from "Do-It-Yourself Database-Driven Web Applications", by Keith Kowalczykowski, Kian Win Ong, Kevin Keliang Zhao, Alin Deutsch, Yannis Papakonstantinou and Michalis Petropoulos, which appears in Proceedings of the Fourth Biennial Conference on Innovative Data Systems Research, CIDR 2009. The dissertation author was the primary investigator and author of this paper. The paper is published under a Creative Commons License Agreement (`http://creativecommons.org/licenses/by/3.0/`). Permission is granted to copy, distribute, display, and perform the work, make derivative works and make commercial use of the work, but the work must be attributed to the authors and CIDR 2009.

VITA

2003	B. Eng. in Computer Engineering, National University of Singapore.
2003-2004	Research Assistant, Hong Kong University of Science and Technology.
2004-2010	Graduate Student Researcher, University of California, San Diego.
2010	Ph. D. in Computer Science, University of California, San Diego.

PUBLICATIONS

Y. Fu, K. Kowalczykowski, K. W. Ong, K. K. Zhao, Y. Papakonstantinou. "Ajax-based Report Pages as Incrementally Rendered Views". *ACM Management of Data International Conference (SIGMOD)*, 2010

G. Bhatia, Y. Fu, K. Kowalczykowski, K. W. Ong, K. K. Zhao, A. Deutsch, Y. Papakonstantinou. "FORWARD: Design Specification Techniques for Do-It-Yourself Application Platforms". *International Workshop on the Web and Databases (WebDB)*, 2009.

K. Kowalczykowski, K. W. Ong, K. K. Zhao, Y. Papakonstantinou, M. Petropoulos, A. Deustsch. "Do-It-Yourself Custom Forms-Driven Workflow Applications". *Conference on Innovative Data Systems Research (CIDR)*, 2009.

Q. Chen, A. Lim, K. W. Ong. "Enabling Structural Summaries for Efficient Update and Workload Adaptation". *Data & Knowledge Engineering (DKE)*, 64(3), 2008.

Q. Chen, A. Lim, K. W. Ong, J. Q. Tang. "Indexing XML Documents for XPath Query Processing in External Memory". *Data & Knowledge Engineering (DKE)*, 59(3), 2006.

Q. Chen, A. Lim, K. W. Ong. "Indexing Graph-Structured XML Data for Efficient Structural Join Operation". *Data & Knowledge Engineering (DKE)*, 58(2), 2006.

Q. Chen, A. Lim, K. W. Ong. "D(K)-Index: An Adaptive Structural Summary for Graph-Structured Data". *ACM Management of Data International Conference (SIGMOD)*, 2003.

ABSTRACT OF THE DISSERTATION

Web Application Creation Made Easy:
A SQL-driven Rapid Development Framework and
a Do-It-Yourself Platform

by

Kian Win Ong

Doctor of Philosophy in Computer Science

University of California, San Diego, 2010

Professor Alin Deutsch, Chair

Building, installing and evolving a custom web application, even one which comprises only Create, Read, Update and Delete (CRUD) pages accessing a single database, is time consuming and expensive. We present two complementary systems that enable the rapid creation, customization and evolution of such a database-driven web application and its pages thereof: a rapid application development framework called **FORWARD**, and a *Do-It-Yourself (DIY) platform* called **app2you**.

FORWARD simplifies the development of AJAX web pages by treating them as rendered views, where the programmer specifies a view using visual units

and (minimally extended) SQL. Such a declarative approach leads to significantly less code, as the framework automatically solves performance optimization problems that the programmer would otherwise hand-code. Since the pages are fueled by views, FORWARD leverages years of database research on incremental view maintenance by creating optimization techniques appropriately extended for the need of pages (nesting, variability, ordering), thereby achieving performance comparable to hand-coded applications.

app2you builds on FORWARD by empowering non-programmer business process owners to create and customize application pages, without programming or database design in a conventional sense. app2you provides a WYSIWYG design facility where the owner specifies the application by manipulating visual aspects of it, responding to questions posed by wizards and setting configuration options. In response, the design facility infers the necessary FORWARD application, including database schema, data structures and code, and immediately produces a revision of the application for the owner's evaluation. The software development cycle is shortened to literally seconds.

Chapter 1

Introduction

Database-driven web applications are the most disciplined and efficient way by which organizations exchange data and run their workflows. For example, a professor, her TAs and her students can collaborate on a database-driven web application where the students submit their projects, the TAs and instructor assign comments and grades, etc. Or a home development company and the home buyers of a particular real estate development project can collaborate, where the buyers issue requests for special features of their homes, the employees take action on the requests and the buyers can check the status of the requests at any point. Or a startup conference where companies apply to present, the organizers review, arrange interviews with selected companies, approve or decline, and finally the approved presenters register. In the absence of the requisite web applications, people run their workflows using inefficient ad-hoc combinations of phone calls, email and file attachments. Confusion, files that are out-of-synch and out-of-date, and communication overhead then become typical maladies [Kra05].

However, building, installing and evolving web applications that are custom to an organization's workflow is work-intensive and time consuming. Consider a custom *human-centric* [TV07] *database-driven web application*, i.e., an application whose:

- entire state is captured by a single database (i.e. there is no mediation / integration across multiple databases, or maintenance of out-of-database

state due to interfacing with external systems)

- state changes (and correspondingly the business process progresses) exclusively in response to user actions on the web pages (i.e. there are no data feeds that are machine-generated, or programs that react to events triggered by other systems)
- pages provide only user actions that perform basic CRUD (Create, Read, Update and Delete) operations

Even though studies [AVFY98, DMS+05, YSRG06, CFB00] have demonstrated that the business process of a human-centric database-driven web application can be specified in terms of its pages using small, restricted languages, its development will still incur months of expensive effort by a team of professional programmers [ABB+07].

In light of this, this dissertation seeks to enable the rapid creation, customization and evolution of human-centric database-driven web applications. Two complementary systems are presented: **FORWARD**, a *rapid application development framework* and **app2you**, a *Do-It-Yourself (DIY) platform* that is built on top of FORWARD. The relationship between app2you and FORWARD is analogous to that between Query by Example (QBE) and SQL in the database field: QBE query tools are visual frontends built on top of general-purpose SQL engines, and they generate a subset of SQL corresponding to what can be easily understood in the visual frontend by non-programmers.

1.1 Rapid Application Development Framework

A key obstacle in the development cycle of web applications is the set of challenges that developers routinely encounter when implementing the pages of web applications with conventional Ajax frameworks[1]. These include:

[1]Similar challenges are presented by other popular frameworks, such as Adobe Flash and Microsoft Silverlight.

- **Distributed programming** An Ajax developer has to coordinate browser-side JavaScript code with server-side Java[2] and SQL code. That is, developing Ajax applications requires distributed programming between the browser and server, and involves multiple languages and data models.

- **Custom logic for partial update** To deliver a polished and performant web application, an Ajax developer needs to implement custom logic for each action that partially updates a web page. In particular, each action requires custom server-side code to retrieve a subset of the data needed for refreshing the page, and JavaScript code to refresh a sub-region of the page. Such event-driven programming is known to be laborious and error-prone, as the developer needs to manually coordinate the different states a page goes through, and ensure that the code implemented correctly transitions from one state to another [Mir03].

- **Disparate component interfaces** 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. However, since there is no standardization of component interfaces, an Ajax developer that integrates components from different libraries need to mitigate differences across interfaces, and write code that refreshes the components' state based on the nature of each update.

To alleviate the complexities and challenges introduced by Ajax programming, we present FORWARD, a rapid application development framework that 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)

[2]Equivalently, C#, Perl, PHP or any other server-side programming language typically used for developing web applications.

templates that put together the page units and the SQL views that feed them with the data. 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.

1.2 Do-It-Yourself (DIY) Platform

Core to the time and monetary expenses of developing a custom database-driven web application, is that building, installing and evolving it is still the exclusive domain of IT professionals, requiring specialized knowledge of database design, database programming and web application programming. An organization needs to engage with IT professionals in a lengthy software development cycle that involves gathering requirements, reviewing specifications, implementation, testing and evaluating deliverables. While large and long-standing organizations can afford such time and money expenses, small and fluid organizations cannot. Furthermore it is very expensive and time-consuming to make even the smallest changes in such applications since the business logic of the application is buried within thousands of lines of code and the original developer may not be around. Even small business process changes lead to practically new projects and expenses. These expenses prevent small fast-paced organizations from commissioning custom applications, delay situational applications that have to be rapidly deployed, and inhibit the timely adaptation of custom applications to changes in business requirements.

To address this, DIY platforms seek to enable *non-programmer business process owners* (also referred to as *business architects* by Forrester Research [TV07]), to build, customize and evolve hosted web applications without programming or database design in a conventional sense. DIY platforms have the potential to dramatically change the economics of building, deploying and maintaining simple web

applications. The intended outcome is paralleled to the emergence of spreadsheets in the 80s and of graphical presentation tools (notably PowerPoint) in the early 90s. In the case of spreadsheets, the desktop revolution had made financial and accounting applications available to users, but many users required a tool that would enable quick customization to their own accounting, record-keeping, reporting and calculation tasks, as various business planning needs emerge. VisiCalc, Lotus 1-2-3, and Excel emerged and radically changed how we treat custom business planning and book-keeping tasks. Similarly, prior to the arrival of Do-It-Yourself presentation tools, polished presentations had to be prepared by specialized graphics professionals. PowerPoint enabled us to easily create and evolve fairly polished presentations by ourselves within hours.

In this dissertation, we present a DIY platform, app2you, where building the application relies exclusively on templates, visual manipulations and specific questions about the pages' behavior: absolutely no coding or database design occurs. app2you provides an application design facility, which provides a WYSIWYG experience for the application owner. Similar to typical WYSIWYG tools, the owner can choose to either customize an existing template or start the application from scratch. He then specifies the application by manipulating visible aspects of it, responding to questions posed by wizards, and setting configuration options. In response to the application owner's choices the design facility infers the necessary FORWARD application (including database schema, data structures and code), and immediately produces a revision of the application for the owner's evaluation. Furthermore the design facility demonstrates to the owner the experience that his users will have when visiting the page and guides him into testing it. In that sense, the experience goes beyond that of Microsoft Excel and PowerPoint, and dramatically shortens the software development cycle into multiple rounds of a few seconds.

The relationship between FORWARD and app2you parallels that between SQL and Query by Example (QBE) in the database field. In the 80s, SQL greatly simplified data access by replacing imperative programming with declarative specification. Nevertheless, writing SQL queries still require programmer sophistica-

tion, therefore QBE tools (such as Microsoft Access) allow non-programmers to formulate queries visually by entering example elements and conditions. A QBE tool translates the visual query into a SQL query (which is more expressive), and a skilled user can customize the SQL query string before it is executed by the database engine. Analogously, programmers will use FORWARD's declarative syntax for rapid development of database-driven web applications, whereas non-programmer business process owners will utilize app2you's design facility. app2you translates the owner's visual manipulations and answers of design prompts into a FORWARD application, which can then be further customized and extended using FORWARD's declarative syntax.

1.3 Roadmap

The roadmap for the dissertation is as follows. In Chapter 2, we first present the FORWARD system, highlighting its language contributions for rapid development and how it automatically provides system optimizations for problems that the developer would otherwise resolve manually. FORWARD's technical specification and implementation details are elaborated in later chapters, where Chapter 3 presents the unified data model of a FORWARD web application, Chapter 4 presents the nested query language, and Chapter 5 presents the incremental view maintenance system optimization. Finally, we present the app2you system, its design facility, and its DIY techniques in Chapter 6.

Chapter 2

Rapid Application Development Framework

2.1 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 2.1.1 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.

FORWARD'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 2.1.3. 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.

2.1.1 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

[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.

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 [O’H06]) the partial update strategy can dramatically decrease the response time.

   For example, consider a proposal reviewing application. On the page shown in Figure 2.1, 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 sub-regions 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 to the next proposal, even before the response handling function has updated the page. Such behavior is a major HCI improvement [Gar05] 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) [Sim09]. 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 2.1, if the developer had issued a query for `Average Grade`, but not `Reviews`, the page 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 [Joh09]) 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.

2.1.2 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 dissertation, 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 `*` for schema-less data and path access of arbitrary depth, which are not necessary for modelling pages but have created challenges for efficient XML/XQuery query processing and view maintenance, and (c) existing database research and technology provide a great leverage for implementation and optimization (see Section 2.1.3), 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 2.5) 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 2.6) 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 and the FORWARD scope in Section 2.2 with a running example. Section 2.2.3 presents the data aspects of the framework. Section 2.2.4 presents the visual layer. Further details on the specification / implementation of the data model and query language of the framework are deferred until Chapters 3 and 4.

2.1.3 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 hand-coded 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 [The09]) 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.

Roadmap Section 2.3 presents optimizations for incrementally maintaining the page, with Section 2.3.1 highlighting the incremental view maintenance of the page

data tree, and Section 2.3.2 presenting the architecture for incrementally refreshing the visual layer. Section 2.4 presents the system implementation and experimental results. Further details on the specification / implementation of the incremental view maintenance on the page data tree is deferred until Chapter 5.

2.2 The FORWARD framework and scope

2.2.1 Running example

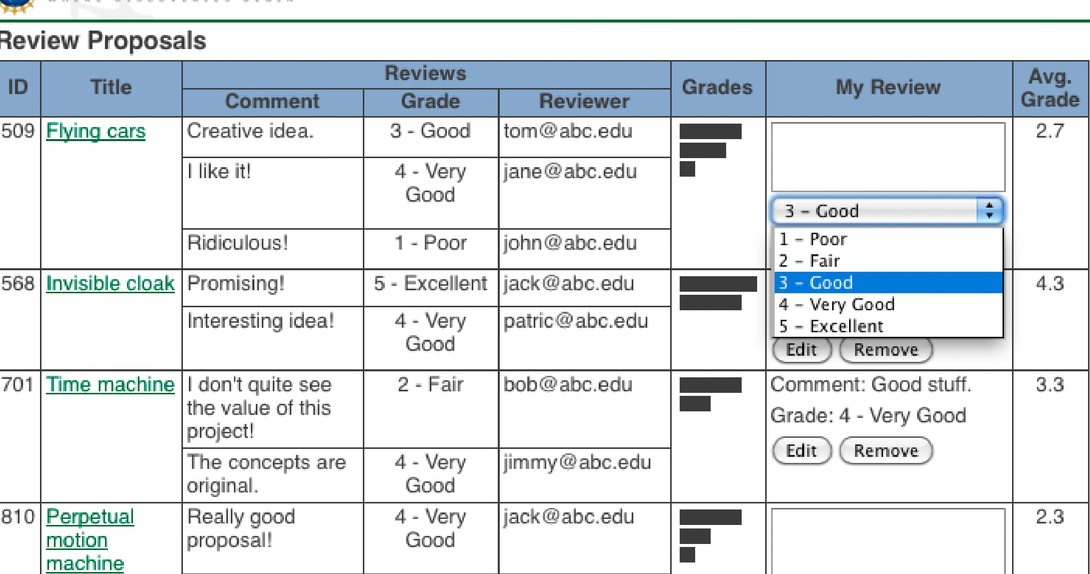


Figure 2.1: `Review Proposals` page

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 2.1). Each Reviewer sees the

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

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.

2.2.2 Architecture

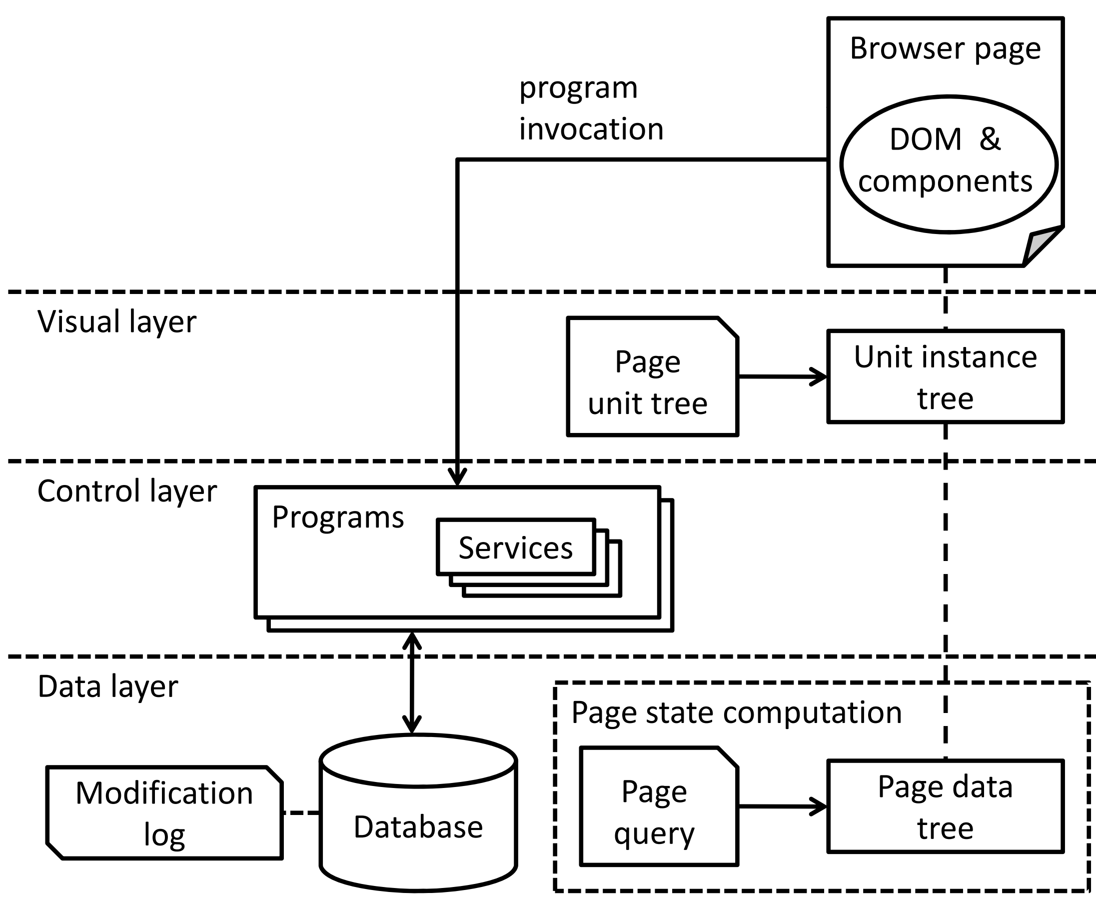


Figure 2.2: FORWARD architecture

Figure 2.2 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 con-

forms 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 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. Technical issues about incremental page update are discussed in detail in Section 2.3, and further implementation details are presented in Chapter 5.

[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 dissertation and are briefly discussed in Section 2.6.

2.2.3 Data layer

```
1  list(
2   tuple(
3    proposal_id  : 509 ,
4    title        : Flying Cars,
5    average_grade: 4.5,
6    reviews      :set(
7     tuple(
8      review_id :  1,
9      comment   :  Creative...,
10     grade     :  3 - Good,
11     reviewer  :  tom@abc.edu)
12     ...),
13   grades       : list(
14    tuple(
15     bar_id : 1,
16     value  : 3)),
17    my_review   : switch(
18     input_tuple: tuple(
19      comment : null,
20      grade   : tuple(
21       grade_ref    : null,
22       grade_options:list(
23        tuple(
24         grade_id   : 1,
25         grade_label: 1 - Poor)
26         ...)))
27
28
29
30      ))
31  ...)
```

Figure 2.3: 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 2.4: Page schema

This section presents an overview of the data model and query language, and the detailed specifications can be found in Chapters 3 and 4.

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 2.1 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*. Each data node (also called *value*) maps to a schema node, and the data tree is homomorphic to the schema tree.

For example, Figure 2.3 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 2.4 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 [BCD89]) 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.

2.2.4 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 [The09] and YUI [YUI09]) 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 2.5 shows the unit tree for `Review Proposals`, with mappings to the page schema tree of Figure 2.4. 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 2.4, and its `ref` and `options` attributes map respectively to the `grade_ref` and `grade_options` attributes.

```
<html:html>
<html:h1>List of all proposals.</html:h1>
<unit:table bind="page_state">

    <column header="Title"> <unit:print bind="title /> </column>
    <column header="Other reviews">
        <unit:table bind="other_reviews">
            <column header="Comment"> <unit:print bind="comment" /> </column>
            <column header="Grade"> <unit:print bind="grade" /> </column>
            <column header="Reviewer"> <unit:print bind="reviewer_id" /> </column>
        </unit:table>
    </column>
    <column header="grades"> <unit:barchart bind="grades" /> </column>
    <column header="My Review">
        <unit:switch bind="my_review">
            <case bind="input_tuple">
                <html:div>
                    <unit:dropdown bind="grade">
                        <ref bind="grade_ref" />
                        <options bind="grade_options">
                            <option bind="grade_option" />
                        </options>
                    </unit:dropdown>
                    <unit:textbox bind="comment" />
                    <unit:button text="Submit" on_click="save_review"/>
                </html:div>
            </case>

            <case bind="display_tuple">
                <html:div>
                    Comment: <unit:print bind="comment" />
                    Grade: <unit:print bind="grade" />
                    <unit:button text="Edit" on_click="edit_review" />
                    <unit:button text="Remove" on_click="remove_review" />
                </html:div>
            </case>
        </unit:switch>
    </column>
    <column header="Avg. Grade"> <unit:print bind="average_grade" /> </column>

</unit:table>
</html:html>
```

Figure 2.5: Unit tree configuration

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 2.5 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 2.3.

2.3 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 2.1.1), 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 refresh. 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 2.3.1) and the visual layer (Section 2.3.2).

2.3.1 Leveraging and extending incremental view maintenance

This section presents an overview of the incremental view maintenance techniques in FORWARD, whereas details can be found in Chapter 5.

The Page State Computation Module (PSC) of FORWARD (see Figure 2.2) 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.

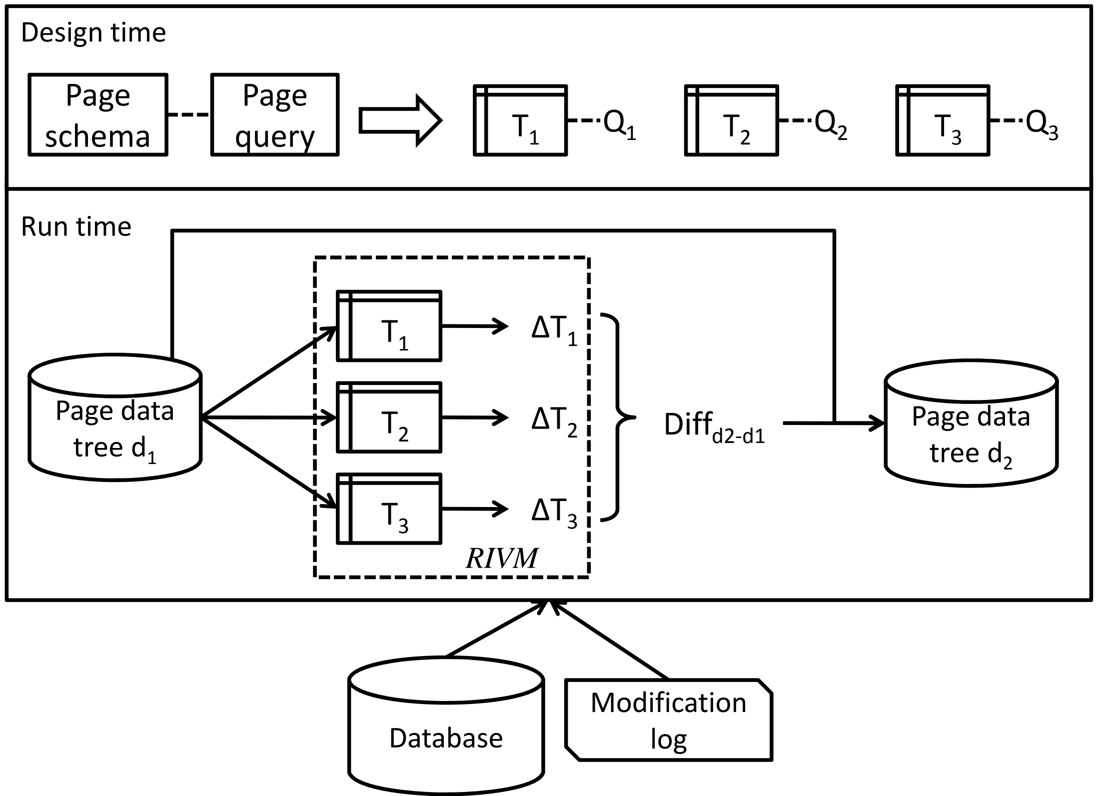


Figure 2.6: Page state computation

Figure 2.6 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 $\mathit{Diff}_{d_2-d_1}$ 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 2.3.1.1 lists the benefits of PSC for page refresh using an example. Section 2.3.1.2 provides background on relational techniques that are leveraged. Section 2.3.1.3 describes how a page schema can be decomposed into flat views, and how to obtain the SQL queries that define each flat view. Section 2.3.1.4 discusses the handling of order through the view maintenance process. Section 2.3.1.5 describes how maintenance results on flat views can be translated and applied to the nested data tree. Further implementation details for PSC, including the incremental rewrite rules supported by the system, are presented in Chapter 5.

2.3.1.1 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 $\mathit{Diff}_{d_2-d_1}$ and consequently d_2. However, computing $\mathit{Diff}_{d_2-d_1}$ 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 2.1. 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.

2.3.1.2 Leveraging relational research

The relational model literature [BLT86, GMS93, GL95, GM95, RSS96] 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 $\Delta^+ R_i$ containing tuples inserted into R_i, and $\Delta^- R_i$ containing tuples deleted from R_i. $\Delta^+ R_i$ and $\Delta^- R_i$ are captured in the modification log. In the same way $\Delta^+ V$ and $\Delta^- V$ 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 $\Delta^+ V$ and $\Delta^- V$ respectively.

PSC focuses on the deferred view maintenance [CGL+96, SBCL00] 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 views. In general, other implementations of RIVM can be used in PSC as well.

2.3.1.3 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 2.3.1.1. 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.

2.3.1.4 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 [DESR03] 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 [YYY+03]. How the embedded order information is restored when the modification is applied to the nested view is discussed next.

2.3.1.5 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 $\mathit{Diff}_{d_2-d_1}$ 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 elaborated in Chapter 5.

2.3.2 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).

2.4 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

Table 2.1: 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		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 2.2: FORWARD 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**	

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 [HA00], and the average US upload and download bandwidths of 1 Mbps and 5 Mbps respectively [oA09].

To demonstrate the end-to-end performance of a server roundtrip, Table 2.1 presents itemized activities and their latencies in the strawman implementation described in the beginning of Section 2.3, starting from the time the submit button is clicked till the browser fully refreshes. Table 2.2 presents the same activities and their latencies in FORWARD, which employs the incremental maintenance techniques of Section 2.3.

In Table 2.1, (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 2.2. (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.

In Table 2.2, note that (4) remains the same as in Table 2.1, 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 2.3.1. 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 2.3.2) 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.

2.5 Related work

The data management research community has created database-driven frameworks for web site [FFLS00] and pure server side web application [CFB00, YSRG06] development. In WebML [CFB00] 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 [FPF+09] that describes how to run XQuery on the browser).

Echo2 [Ech09], ZK [ZK 09], Backbase [Bac09] and ICEfaces [ICE09] 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 [Wik09a] 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 2.1 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 [Goo09] and Cornell's Hilda [YGG+07] 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, [CAL+02] 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 [ZGMHW95, AASY97, MQM97]). More recently, [AMR+98, ZGM98, DESR03, AFP03, STP+05, FKSV08] 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.

2.6 Summary 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.

Acknowledgements

This chapter contains material from “Ajax-based Report Pages as Incrementally Rendered Views”, by Yupeng Fu, Keith Kowalczykowski, Kian Win Ong, Kevin Keliang Zhao and Yannis Papakonstantinou, which appears in Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010. The dissertation author was the primary investigator and author of this paper. The material in this chapter is copyright ©2010 by the Association for Computing Machinery, Inc. (ACM). Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page in print or the first screen in digital media. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Publications Dept., ACM, Inc., fax +1 (212) 869-0481, or email permissions@acm.org.

Chapter 3

Data Model

The data model is designed to enable the developer to avoid the impedance mismatch of programming in multiple languages (JavaScript, SQL, Java) in order to implement Ajax pages. It follows the relational model, minimally extended with nesting, variability and ordering in order to conveniently represent data typically found on web pages. Similarly, the query language presented in Chapter 4 is a minimal extension of SQL. We chose this approach 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 `*` for schema-less data and path access of arbitrary depth, which are not necessary for modelling pages but have created challenges for efficient XML/XQuery query processing and view maintenance, and (c) existing database research and technology provide a great leverage for implementation and optimization, 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.

To represent data on pages, as well as to communicate data across subsystems in a standard fashion (independent of whether they are stored in main memory, relational databases or other data sources), the following is required of the data model:

- Nesting. In an application, data collected by forms and reported pages are often nested. Nesting is not natively handled by the (flat) relational model,

which is one of the sources of "impedance mismatch" between SQL databases and application frameworks.

- Strong typing. A strict type system generally provides more opportunities for analysis and optimization. For example, type constraints such as uniqueness, functional dependencies and cardinality provides richer type information that can benefit query optimization and program verification.

- Ad-hoc typing.[1] To maximize re-usability, a service needs to declare its input to be of any type that possesses specific attributes. For example, an `EmailService` requires attributes `name` and `email`, therefore it should work with types `Person` and `Company` as long as attributes `name` and `email` are defined on them. Ad-hoc typing allows the creation of an ad-hoc type `EmailRecipient` that qualifies as the **supertype** of both `Person` and `Company`, (or indeed any type that has the two necessary attributes).

- Sum types.[2] A data value may take on one of several different (but fixed) types. For example, to model conditional forms, a switch unit conditionally displays one of several different groups of child units. As another example, programs may contain conditional branches. At the reconvergence point, the respective outputs of the if-branch and the else-branch need to be represented as a choice between two types.

[1] Java uses a *nominative type system* that does not facilitate ad-hoc typing, but there are other type systems that do. Dynamic languages such as Python use *duck typing*, where type checking is deferred until runtime. OCaml uses *structural subtyping*, where a subtype is simply one that structurally contains all attributes (or more) of the supertype. A recent addition to C# (due to LINQ) is *anonymous types*, which allow ad-hoc creation of types that encapsulate a set of read-only attributes. Note that all such systems offer the convenience of not requiring explicit type declarations: the statically-typed languages (i.e. OCaml and C#) accomplish this by using type inference. See respective Wikipedia articles for more information.

[2] Also known as *tagged unions*. Examples of language support include C's `UNION` keyword, and Haskell's / ML's algebraic data types. Note that tagged unions are not related whatsoever to the union operator in relational algebra. See respective Wikipedia articles for more information.

3.1 Syntax

The grammar for the syntactic representation of a data tree and a schema tree are given in Figure 3.1 and Figure 3.2 respectively. Subsequently, Figure 3.3 and Figure 3.4 show examples of syntactically representating a data tree and a schema tree.

value△	→	**collection_value**
	|	**tuple_value**
	|	**switch_value**
	|	**scalar_value**
collection_value	→	[*name* :] (`set` | `list`) (**tuple_value** [, **tuple_value**]*)
tuple_value	→	[*name* :] `tuple(` **value** [, **value**]*)
switch_value	→	[*name* :] `switch(` **tuple_value**)
scalar_value	→	[*name* :] (`string` | `int` | `float` | `date`)

Figure 3.1: BNF for data tree

type△	→	**collection_type**
	|	**tuple_type**
	|	**switch_type**
	|	**scalar_type**
collection_type	→	[*name* :] (`set` | `list`) (**tuple_type**, `PRIMARY KEY(` *name* [, *name*]*))
tuple_type	→	[*name* :] `tuple(` **type** [, **type**]*)
switch_type	→	[*name* :] `switch(` **tuple_type** [, **tuple_type**]*)
scalar_type	→	[*name* :] (`string` | `int` | `float` | `date`)

Figure 3.2: BNF for 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 3.3: Example data tree

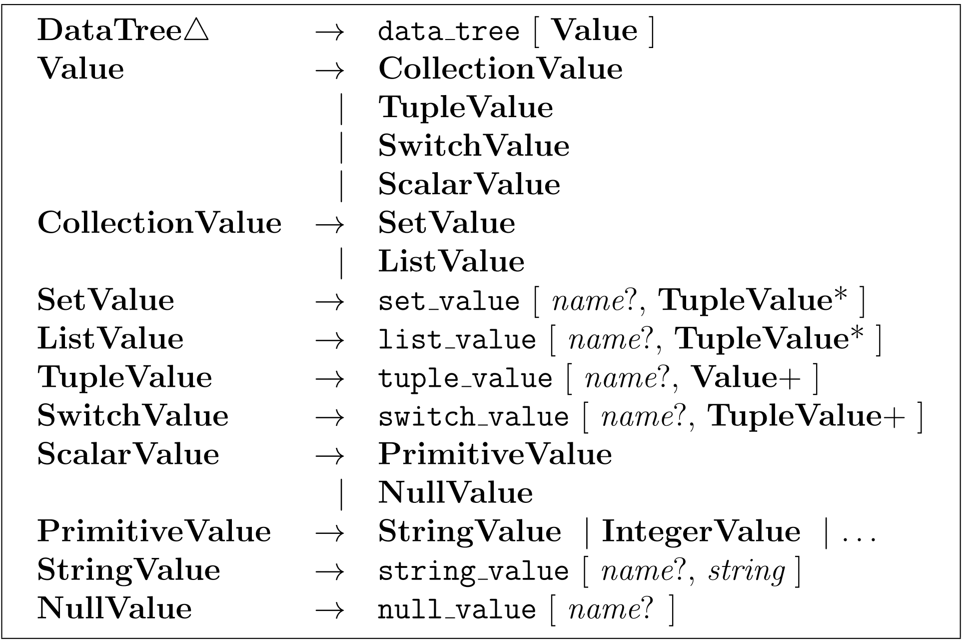
```
list(
  tuple(
    proposal_id     : int,
    title           : string,
    average_grade   : float,
    reviews         : set(
      tuple(
        review_id   : int,
        comment     : string,
        grade       : string,
        reviewer    : string
      )

    ),
    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 3.4: Example schema tree

3.2 Type System

3.2.1 Data Trees

Figures 3.5 and 3.6 provide the tree grammars of the data model concepts that are to be defined shortly. Each concept has both a data equivalent and a type equivalent, where the latter serves to represent characteristics that apply across multiple data instances. We first define the *data tree* and *schema tree* as independent structures: the consistency between the two trees (such as whether a data tree corresponds 1-to-1 to a schema tree) is defined later in Section 3.2.4.



DataTree△	→	`data_tree` [**Value**]
Value	→	**CollectionValue**
	|	**TupleValue**
	|	**SwitchValue**
	|	**ScalarValue**
CollectionValue	→	**SetValue**
	|	**ListValue**
SetValue	→	`set_value` [*name*?, **TupleValue***]
ListValue	→	`list_value` [*name*?, **TupleValue***]
TupleValue	→	`tuple_value` [*name*?, **Value**+]
SwitchValue	→	`switch_value` [*name*?, **TupleValue**+]
ScalarValue	→	**PrimitiveValue**
	|	**NullValue**
PrimitiveValue	→	**StringValue** | **IntegerValue** | ...
StringValue	→	`string_value` [*name*?, *string*]
NullValue	→	`null_value` [*name*?]

Figure 3.5: Grammar for Data Tree

SchemaTree△	→	`schema_tree` [**Type**, **Constraint**+]
Type	→	**CollectionType**
	|	**TupleType**
	|	**SwitchType**
	|	**ScalarType**
CollectionType	→	**SetType**
	|	**ListType**
SetType	→	`set_type` [*name*?, **TupleType**]
ListType	→	`list_type` [*name*?, **TupleType**]
TupleType	→	`tuple_type` [*name*?, **Type**+]
SwitchType	→	`switch_type` [*name*?, **TupleType**+]
ScalarType	→	**PrimitiveType**
	|	**NullType**
PrimitiveType	→	**StringType** | **IntegerType** | ...
StringType	→	`string_type` [*name*?]
NullType	→	`null_type` [*name*?]

Figure 3.6: Grammar for Schema Tree

The basic building block is the *tuple*, which is a sequence of attributes, that is, named *value*s. The tuple's type indicate the fixed size of the sequence, the ordering of the attributes, the unique attributes, and the respective types of attributes. A variety of attribute values can be stored in a tuple.

A *scalar value* is a value that is handled natively by a SQL database:

1. *primitive value*, such as string, binary data (byte array), boolean, big integer, integer, double, float and timestamp.

2. *null value*, which necessitates ternary logic during expression evaluation in the spirit of SQL. It conforms to all possible types.

A *collection value* is a set of homogeneous tuples (i.e. all the tuples have the same type). Unlike the relational model, the collection is itself a value that can be stored in a tuple, thereby allowing collections to be nested.

There are multiple extensions to the nested relational model:

1. Any value can be the root of the data tree.

2. A collection type must be associated with a primary key constraint (Section 3.2.2), such that each tuple in the collection is uniquely identifiable. An unordered collection is a *set value*, whereas an ordered collection is a *list value* (more precisely, a unique list). The ordering of tuples within a list value are determined by the values of *ordering attributes* (Section 3.2.2).

3. A *switch value*[3] allows choosing among different tuple types. Its type is a mapping of distinct cases to tuple types; the value comprises of exactly one case, and a tuple of the corresponding type.

4. A tuple is itself a value, therefore it can be contained directly within another tuple. This is a convenience feature that facilitates grouping of values: such a grouping can always be implicitly captured with hierarchical naming.

   For example, a `person` tuple can contain a `physical` tuple with attributes `height` and `weight`, and a `contact` tuple with attributes `phone` and `address`. This showcases grouping of values.

As a short-hand, *direct attributes* of a tuple type are attributes immediately specified by a tuple type, *indirect attributes* are those reachable by traversing only tuple types, and *conditional attributes* are those reachable by traversing only tuple and switch types. The respective definitions can be extended in a straightforward manner for the direct, indirect and conditional attributes of a collection type, a tuple, or anything that corresponds to exactly one tuple type.

There is a tree homomorphism between a data tree and its corresponding schema tree. One or more data nodes (i.e. values) can map to a single schema node (i.e. type).

Implementation notes:

1. Data nodes do not always have to be recursively contained within a data tree. This is necessary because (1) data nodes need to be constructed before they can be attached to a data tree (2) it facilitates detaching a sub-tree, and re-attaching it to another data tree. Likewise for schema nodes.

[3]Analogous to XML Schema's `choice` element.

2. A data tree d_1 can be compared to another data tree d_2 by considering whether d_1 is isomorphic or homomorphic to d_2. Likewise for schema trees.

3. Cloning an entire data tree, or sub-tree thereof, is supported.

3.2.2 Constraints

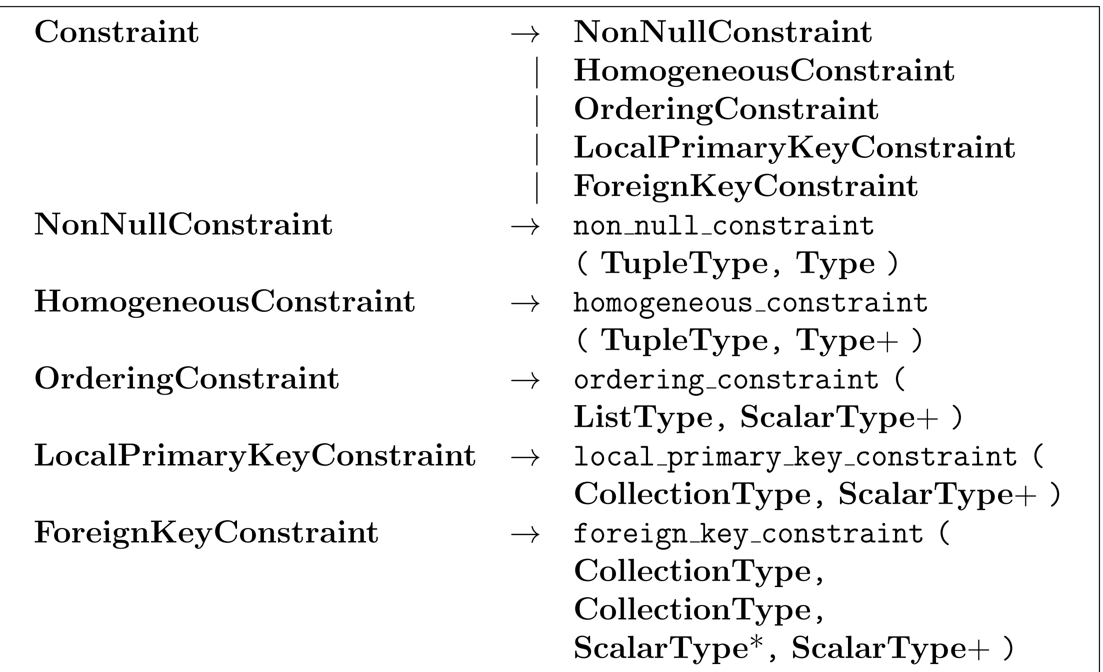


Figure 3.7: Grammar for Constraints

A schema tree contains the *data constraints* for all types recursively within.

A data constraint is a declaration that data satisfies certain criteria. Each data constraint has a *constraint checker* that verifies that the containing data tree is consistent with respect to the constraint. This decoupling of declaration and enforcement provides more flexibility for data tree writers: enforcement can be deferred for efficiency, and data trees can be left as inconsistent if so desired. Note that a constraint can only operate within a single data tree, i.e. it cannot cross data tree boundaries.

We will first present the data constraints that can be declared, and defer the discussion on their enforcement until Section 3.2.4.

The data constraints available are:

Non-null constraint(T, A) Declares on tuple type T that all corresponding tuples cannot map attribute A to the null value. A must be a direct attribute of T.

Homogeneous constraint$(T, \overline{A})$ Declares on tuple type T that each corresponding tuple t maps attributes $\overline{A}$ to either all null values or all non-null values. $\overline{A}$ must contain only direct attributes of T.

Ordering constraint$(L, \overline{A})$ Declares on list type L that for each corresponding list value l, l has tuples ordered by the concatenation of values $\overline{v}$ for attributes $\overline{A}$. $\overline{A}$ must contain only direct or indirect scalar attributes of L.

Local uniqueness constraint$(C, \overline{A})$ Declares on collection type C that for each corresponding collection c, c has tuples with unique[4] values $\overline{v}$ for attributes $\overline{A}$. $\overline{A}$ must contain only direct or indirect scalar attributes of C.

Global uniqueness constraint$(C, \overline{A})$ Declares on collection type C that $c^{\cup}$, the union of all corresponding collections, has tuples with unique values $\overline{v}$ for attributes $\overline{A}$. $\overline{A}$ must contain only direct or indirect scalar attributes of C.

Local primary key constraint$(C, \overline{A})$ Declares on collection type C that in each corresponding collection c, attributes $\overline{A}$ are the primary key. Restrictions are as follows:

1. $\overline{A}$ must contain only direct or indirect scalar attributes of C.
2. The values corresponding to each attribute A of $\overline{A}$ must be immutable.
3. There must be a global uniqueness or local uniqueness (or both) constraint on $(C, \overline{A})$.
4. For each attribute A in $\overline{A}$ and its parent tuple type T, there must be a non-null constraint on (T, A).

[4]Null values are not considered equal under uniqueness, as per the SQL standard.

5. There must be exactly one local primary key constraint on C[5].

Global primary key constraint$(C, \overline{A})$ Declares on collection type C that in $c^{\cup}$, the union of all corresponding collections, attributes $\overline{A}$ are the primary key. Restrictions are as follows:

1. $\overline{A}$ must contain only direct or indirect scalar attributes of C.
2. The values corresponding to each attribute A of $\overline{A}$ must be immutable.
3. There must be a global uniqueness constraint on $(C, \overline{A})$.
4. For each attribute A in $\overline{A}$ and its parent tuple type T, there must be a non-null constraint on (T, A).
5. There must be at most one global primary key constraint on C.

Foreign key constraint$(C_f, C_r, \overline{A_f}, \overline{A_r})$ Declares on collection type C_f that each tuple in corresponding collections has values $\overline{v_f}$ for foreign key attributes $\overline{A_f}$, that match values $\overline{v_r}$ for referenced attributes $\overline{A_r}$ of some tuple in some collection of collection type C_r. If there is at least one null value in $\overline{v_f}$, the match trivially succeeds and the constraint is considered satisfied.[6]

For example, consider an employee collection with a foreign key comprising two attributes `department_id` and `manager_id`. When both attributes are non-null, the constraint ensures that the attributes match a manager who is really managing a specific department. When at least one value is null (e.g. the employee is the department head himself), then the match trivially succeeds. In this example, note that there are likely two other foreign keys respectively for `department_id` (ensure that the department is valid) and `manager_id` (ensure that the manager is a valid employee).

Restrictions are as follows:

[5] The mandatory local primary key constraint ensures that a collection is trivially in partitioned normal form (PNF) [RKS88].

[6] The match semantics come from the default `MATCH SIMPLE` option in SQL. `MATCH FULL` semantics can be obtained by adding a separate homongeneous constraint.

1. $\overline{A_f}$ must contain only direct or indirect scalar attributes of C_f.
2. $\overline{A_r}$ must contain only direct or indirect scalar attributes of C_r.
3. There must be a global uniqueness constraint on $(C_r, \overline{A_r})$.

3.2.3 Primary keys & Context

Given the mandatory local primary key constraint on collection type C and attributes $\overline{L}$, we define the *local primary key* attributes of C to be the attributes $\overline{L}$. Furthermore, we define the *context attributes* of C to be:

- $\overline{G}$ if there is a global primary key constraint on collection type C and attributes $\overline{G}$, otherwise
- $\overline{L}$ if C is a top-level collection, otherwise
- the concatenation of the *parent key* and $\overline{L}$, where the parent key is the context of the closest ancestor collection type of C.

Using Figure 3.8 as an example, and assuming the top-level collection has context attribute `(student_id)`, the nested collection's context attributes can be specified to be any one of the following:

- `(enroll_id)` if there is a global key constraint on `(enroll_id)`.
- `(student_id, class_id)` if there is no global key constraint, and there is a local key constraint on `class_id`.
- `(student_id, enroll_id)` if there is no global key constraint, and there is a local key constraint on `enroll_id`.

student_id	**name**	**enrollment**			
		enroll_id	**class_id**	**class**	**grade**
1	John	1	1	Math	B
		2	2	Physics	C
		3	3	Art	B
2	Mary	4	1	Math	A
		5	4	History	A

Figure 3.8: Example top-level collection

The context attributes as declared in the schema determines the *context*, which serves as an identifier for values that is (1) globally unique within the data tree (2) stable due to the immutability of local / global primary key attributes. The context of a data node d (i.e. value) is a tree comprising:

1. a root-to-node path p_d to the node d.
2. a root-to-node path p_{g_i} for each closest value g_i of context attributes $\overline{G}$ of closest ancestor collection type C, where closeness is determined by the number of nodes in common with p_d.
3. no other paths.

Using the data tree in Figure 3.9a as an example, and assuming that the context attributes are respectively `(plan_id)` and `(plan_id, member_id)`, the context of the string value `John` is illustrated in Figure 3.9b, whereas the context of the top-level collection is illustrated in Figure 3.9c. The nodes identified by the context are highlighted with rectangles.

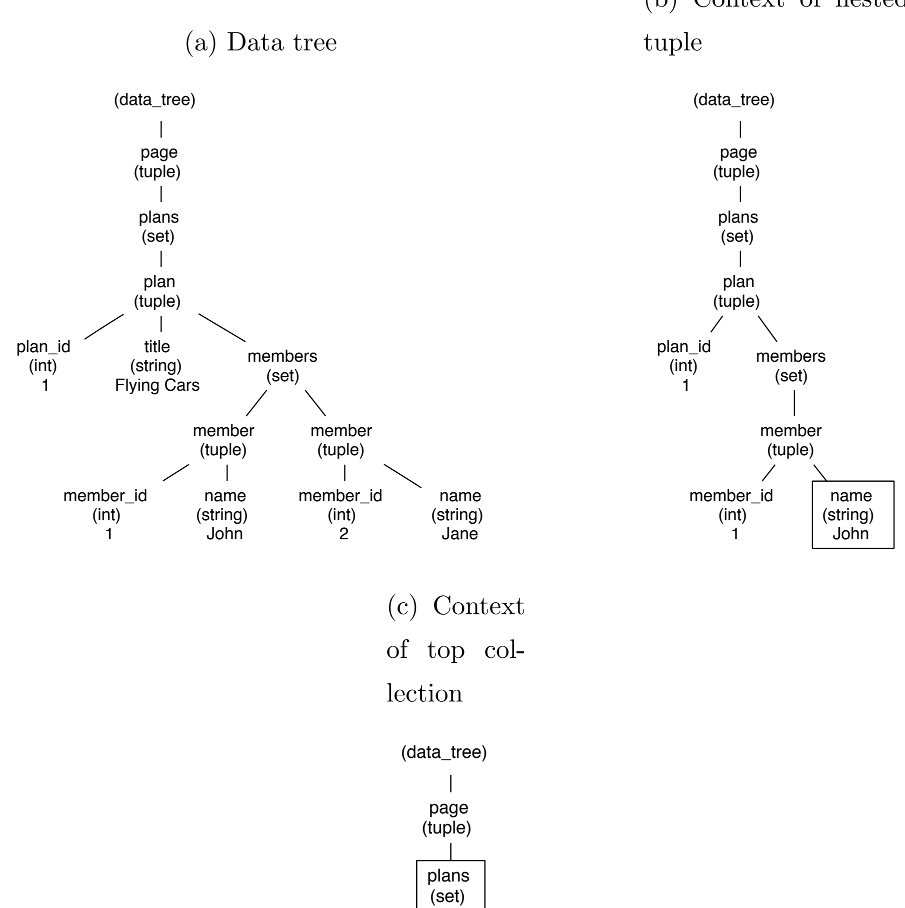


Figure 3.9: Context

To define a textual path syntax that uniquely identifies a data node, we define the *identifier suffix* of the tuple t of a collection c to be:

1. a comma-separated string of the values of its context attributes, if there is a global primary key constraint on collection type C of c, otherwise
2. a comma-separated string of the values of its local primary key attributes.

The *qualified name* of a data node is:

1. its name concatenated with `@` and the identifier suffix, if it is a collection tuple, otherwise

2. its name.

A *context path* (or *data path*) for a data node is a textual syntax of its context, represented by a dot-separated string of the qualified names along its root-to-leaf path. For example, the data path for the string `Jane` in Figure 3.9a is: `page.plans.plan@1.members.member@2.name`.

Analogously (but simpler), the *schema path* for a schema node is a dot separated string of the names along its root-to-leaf path. For example, the corresponding schema path for the `name` attribute in Figure 3.9a is: `page.plans.plan.members.member.name`.

3.2.4 Data Consistency

Each data tree must be associated with a schema tree, whereas a schema tree can be associated with zero or one data tree. This association is bidirectional, by having a pair of pointers between a data tree and schema tree.

A data tree is *consistent* when it is both (1) *type consistent* with respect to the schema tree and (2) *constraint consistent* with respect to the constraints on the schema tree.

For a data tree to be type consistent, a generic *type checker* needs to evaluate to true given the data tree and schema tree. Each node in the data tree must satisfy the following criteria:

1. A data node is associated with a node in the schema tree.

2. The data node is *conformant* to the schema node. For example, a string value is always conformant to a string type, whereas a tuple value is only conformant to a tuple type if both have identical attribute names. As a special case, a null value is conformant to all types.

3. The parent of the data node is associated with the parent of the schema node. For tuple values and switch values, all children nodes must be named, and the respective names on the data tree and the schema tree must agree.

For a data tree to be constraint consistent, it must be type consistent and have all the constraint checkers on the schema tree evaluate to true. Typically a constraint checker is implemented as a query.

Note that a data tree can be left as inconsistent: it is the caller's responsibility to invoke the respective type checker and constraint checkers. For the purposes of efficiency though, especially since a data tree is used to pass data from one sub-system to another, it is important that it memoizes whether it is consistent by storing a *type consistency flag* and a *constraint consistency flag*.

1. When the data tree is modified by adding, removing or changing nodes, both the type consistency flag and the constraint consistency flag are reset.

2. When the schema tree is modified by adding, removing or changing nodes, both the type consistency flag and the constraint consistency flag are reset.

3. Adding or changing a constraint resets only the constraint consistency flag.

4. Removing a constraint does not reset any flags.

When passing a data tree from one sub-system to another, sometimes it may be necessary to mark a data tree as *immutable*. This is accomplished with a *immutable flag* on a data tree that can only be set once, and cannot be reset thereafter. When a data tree is immutable:

1. Modifying the data tree, schema tree or constraints will throw an exception[7].

2. The data tree must be type consistent and constraint consistent. Modifying the type consistency and constraint consistency flags will throw an exception.

3. The only way to modify the data tree, schema tree or constraints again is to clone them (and discard the originals). Obviously, the cloned copy should default to being mutable

[7] Whereas Java collections are immutable in a shallow manner, a data tree is immutable in a deep recursive manner.

To implement the type consistency, constraint consistency and immutable flags, each data node, schema node and constraint needs to locate its corresponding data tree (when it is attached to one). This is achieved by navigating (1) the parent pointers on each node (2) the pointers between the data tree and schema tree.

3.3 Data Diffs

A *data diff* compactly represents the differences between two data trees. This representation is useful when encoding changes that have occurred on a data tree (presumably for efficient transmission or storage), or changes to be made to a data tree (presumably through an update language). A data diff comprises:

1. the *operation*, which is either of *insert*, *replace* or *delete*.
2. the *context*, which is a data path that uniquely identifies where the change has occurred in the data tree.
3. the *payload*, which is a data (sub-)tree that represents the change.

Where the payload is a collection tuple, its values for the context attributes in the payload must be identical to those used in the context.

We focus on data diffs where the updated data tree remains conformant to the schema tree. That is, we defer discussing data diffs which also require schema changes, such as changing the number of attributes within a tuple, changing the type of an attribute etc.

As an example, Figure 3.10 shows a original data tree, whereas Figure 3.11 shows the respective kinds of data diffs that can occur on the data tree. In Figure 3.11, the rectangle indicates the data node that has been uniquely identified by the context. Note that the context has been illustrated as a tree structure for ease of exposition, typically the more compact representation for it is a context path.

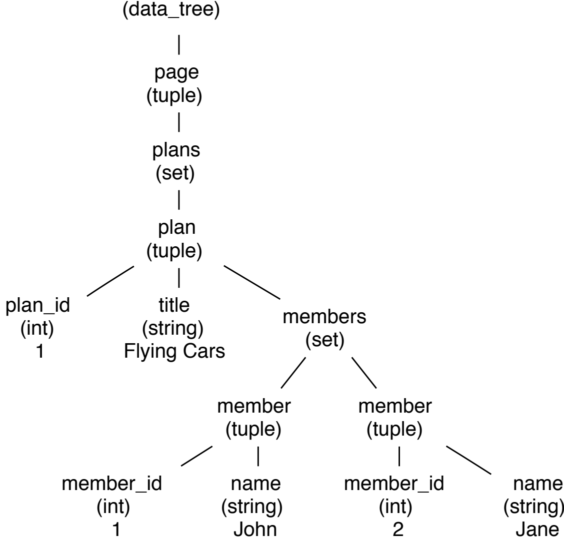


Figure 3.10: Original Data Tree

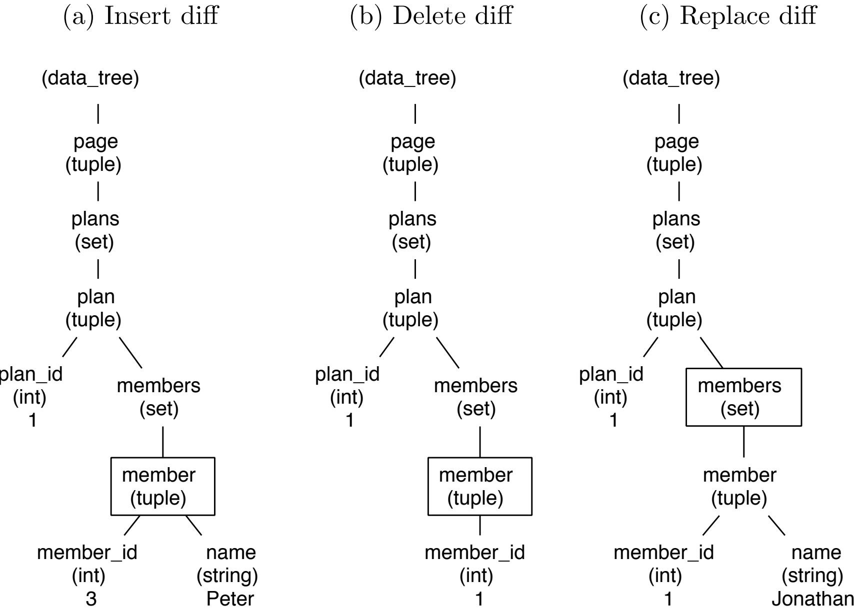


Figure 3.11: Data Diffs

The semantics of the data diffs are as follows:

- **Insert diff**. The context identifies the tuple in the new data tree (note the symmetry to the delete diff), and the payload represents the tuple to be

inserted. Figure 3.11a shows an insert diff, where the context is `page.plans.plan@1.members.member@3`, and the inserted tuple for `Peter` has `member_id` equals to `3`.

When the data diff is used to modify a data tree where default values have been specified on the schema tree, omitting an attribute in the payload will cause default values to be assigned. As a special case, omitting the primary key attributes will cause automatic assignment of unique identifiers, which will necessitate omitting as well the last identifier suffix on the context. In the example, the context will become `page.plans.plan@1.members.member`.

- **Delete diff**. The context identifies the tuple in the data tree to deleted, and there is no payload necessary. All descendants of the tuples would be recursively deleted. Figure 3.11b shows a delete diff, where the context is `page.plans.plan@1.members.member@1`, and the tuple for `John` is deleted.
- **Replace diff**. The context identifies the value in the data tree to be replaced, and the payload represents the replacement value. Figure 3.11c shows a replace diff, where the context is `page.plans.plan@1.members`. Both `John` and `Jane` tuples are removed, and the new `members` collection has a single tuple for `Jonathan`. Note that while the `members` sub-tree has changed, its sibling attribute `title` in the data tree remains unaffected.

Acknowledgements

This chapter contains material from the technical report "Technical Specifications of the FORWARD Framework", by Yupeng Fu, Kian Win Ong, Kevin Keliang Zhao, Yannis Papakonstantinou and Michalis Petropoulos. The dissertation author was the primary investigator and author of the relevant chapters in the technical report.

Chapter 4

Query Language

As described in Chapter 3, the query language is designed as a minimal extension of the SQL syntax (as opposed to XQuery or OQL) because (a) SQL has a much larger programmer audience and installed base (b) SQL has a smaller feature set, omitting operators such as `//` and `*` for schema-less data and path access of arbitrary depth, which are not necessary for modelling pages but have created challenges for efficient XML/XQuery query processing and view maintenance, and (c) existing database research and technology provide a great leverage for implementation and optimization, which enables focus on the truly novel research issues without having to re-express already solved problems in XQuery or having to re-implement database server functionality.

The minimal syntactic extensions to SQL are illustrated below with examples:

1. **Nested output** The `SELECT` clause allows expressions that create nesting, including sub-queries that output nested collections. The following example creates an `employees` collection nested within each department. Note that the sub-query is parameterized by the tuple variable `d` from the top-level query.

```
1  SELECT  *,
2          (
3          SELECT  *
4          FROM    employees AS e
```

```
5          WHERE   e.department_ref = d.department_id
6          ) AS employees
7   FROM    departments AS d
```

2. **Nested input** The `FROM` clause allows navigation into nested collections of tuple variables. The following example creates a flat collection of employees from a nested one.

```
1   SELECT  d.id, e.id, e.name
2   FROM    departments AS d, d.employees AS e
```

3. **Variability** A `SWITCH WHEN ... ELSE ... END` expression allows constructing switch values. Similar to SQL's `CASE WHEN ... ELSE ... END` conditional expression, a `SWITCH` expression evaluates condition expressions on multiple branches. However, each branch evaluates to a (potentially heterogeneous) `case:value` pair, where the case name identifies the currently selected case. The following example constructs a switch value that has either the `undergraduate` or the `graduate` case based on whether the student has an advisor.

```
1   SELECT  s.id,
2           SWITCH
3               WHEN (s.advisor IS NOT NULL) THEN
4                   TUPLE(
5                       s.advisor AS advisor,
6                       s.funding AS funding
7                   ) AS graduate
8               ELSE
9                   TUPLE(
10                      s.tuition AS tuition
11                  ) AS undergraduate
12          END
13  FROM    students AS s
```

4. **Non-collection output** A query can return a value that is not a collection. In particular, an expression (without a `SELECT` clause) is a valid query. The following examples return an integer value and a tuple value respectively.

```
1   1+1
```

```
1 TUPLE(123 AS x, 456 AS y)
```

In order to leverage existing SQL implementations, a query is parsed and translated into one or more SQL queries. The flat SQL tables returned by the SQL queries are then reconstructed into nested data trees. Conceptually, the system resembles XQuery query processors [GST04, PCS+05, BGvK+06] that execute XQuery by translating it into SQL queries that run on top of a relational database. Details of the query language implementation are organized as follows:

- **Syntax Analysis** Section 4.1 presents both the BNF syntax and abstract syntax tree (AST) for the query language. We use JavaCC to generate a parser that parses the textual syntax into an AST.
- **Semantic Analysis** Section 4.2 presents the semantic checks to be performed on the AST. This includes type checking / type inference (Section 4.2.2) and primary key checking / inference (Section 4.2.4).
- **Query Evaluation** Section 4.3 presents how the nested query is evaluated using the SQL engine. In order to produce data structures for a SQL engine, Section 4.3.1 explains how the schema tree is decomposed into relational tables, and Section 4.3.2 explains how the nested query is decomposed into one or more SQL queries with a system of rewrite rules. Section 4.3.3 describes how the SQL queries are evaluated, and finally Section 4.3.4 describes how the results of the SQL queries are used to reconstruct the nested data trees.

We first present the query language assuming that it operates only over data stored in a SQL database. Section 4.4 further discusses how to extend the query language to operate over in-memory data, as well as other data sources.

4.1 Syntax Analysis

The query syntax in Figures 4.1 and 4.2 is designed to closely resemble SQL. For ease of exposition, the BNF presented here does not take into account operator precedence.

```
query△        →  SELECT select_list
                 [ FROM from_item [, from_item]* ]
                 [ WHERE expression ]
                 [ GROUP BY expression [,expression]* ]
                 [ HAVING expression [,expression]* ]
                 [ UNION | INTERSECT | EXCEPT query ]
                 [ ORDER BY expression [ ASC | DESC ]
                 [, expression [ ASC | DESC ]]* ]
                 [ KEY ON expression [,expression]* ]
                 [ LIMIT expression [CACHE integer]]
                 [ OFFSET expression]
select_list   →  *
              |  select_item [,select_item]*
select_item   →  tuple_name.*
              |  expression [ AS alias ]
from_item     →  collection_name AS alias
              |  variable.path AS alias
              |  function_call AS alias
              |  ( query ) AS alias
              |  from_item join_type from_item [ ON condition]
```

Figure 4.1: Syntax of query (SELECT clause)

expression △	→	**expression** *arithmetic_op* **expression**
	|	[+ | -] **expression**
	|	*scalar_literal*
	|	variable.path
	|	**function_call**
	|	(**query**)
	|	parameter
	|	(**expression**)
	|	**condition**
	|	CASE [WHEN **expression** THEN **expression**]+
		[ELSE **expression**] END
	|	SWITCH
		[WHEN **expression** THEN **expression** AS *alias*]+
		[ELSE **expression** AS *alias*] END
	|	TUPLE(**expression** AS *alias* [, **expression** AS *alias*])
condition	→	**condition** *logical_op* **condition**
	|	NOT **condition**
	|	**expression** *comparison_op* **expression**
	|	**expression** [NOT] LIKE **expression**
	|	**expression** IS [NOT] NULL
	|	EXISTS (**query**)
function_call	→	*function_name* ([**expression** [, **expression**]*])

Figure 4.2: Syntax of query (expressions)

Parsing the query produces an AST. The tree grammar for the class structure of the AST nodes are shown in Figures 4.3, 4.4 and 4.5.

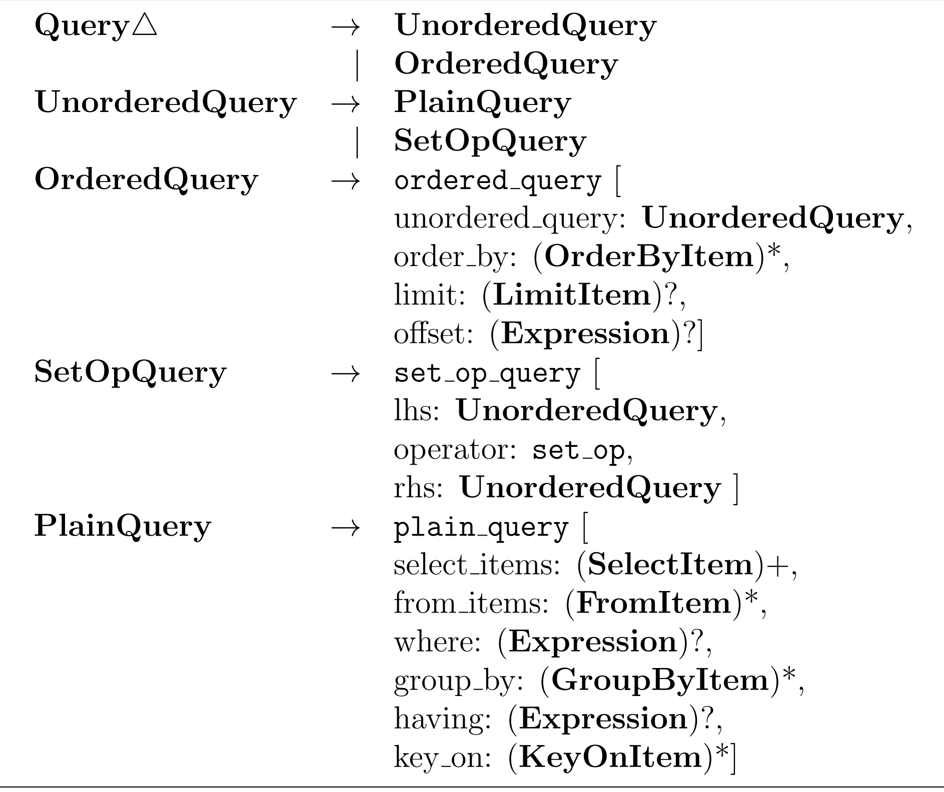


Figure 4.3: AST (queries)

SelectItem	→	**WildcardRef**
	|	**CollectionWildcardRef**
	|	**SelectExpressionItem**
CollectionWildcardRef	→	`collection_wildcard_ref`
		[variable: `string`]
WildcardRef	→	`wildcard_ref`
SelectExpressionItem	→	`select_expr_item` [
		select_expr: **Expression**,
		alias: `string`]
FromItem	→	**FromExpressionItem**
	|	**JoinItem**
FromExpressionItem	→	`from_expr_item` [
		from_expr: **FromExpression**,
		alias: `string`]
FromExpression	→	**AttributeReference**
	|	**CollectionReference**
	|	**Query**
	|	**FunctionCall**
JoinItem	→	`join_item` [
		lhs: **FromItem**,
		`join_type`,
		rhs: **FromItem**]
GroupByItem	→	`group_by_item` [expr: **Expression**]
KeyOnItem	→	`key_on_item` [expr: **Expression**]
OrderByItem	→	`order_by_item` [
		expr: **Expression**,
		asc: `boolean`]
LimitItem	→	`limit_item` [
		limit: **Expression**,
		cache: `integer`]
AttributeReference	→	`attribute_ref`
		[variable: `string`,
		path: `string`]
CollectionReference	→	`collection_ref` [path: `string`]

Figure 4.4: AST (`SELECT` clause)

Scalar	→	**ScalarLiteral**
	|	`parameter`
ScalarLiteral	→	**IntegerLiteral**
	|	**StringLiteral**
	|	...
FunctionCall	→	`function_call` [(**Expression**)*]
TupleAttribute	→	`tuple_attribute` [
		expression: **Expression**,
		alias: `string`]
Expression	→	**AttributeReference**
	|	**CollectionReference**
	|	**Scalar**
	|	**FunctionCall**
	|	**Query**
	|	**SwitchConstructor**
	|	**CaseExpression**
	|	**TupleConstructor**
TupleConstructor	→	`tuple_constructor` [**TupleAttribute**+]
SwitchConstructor	→	`switch_constructor` [
		when_items: (**WhenItem**)+,
		else_item: (**ElseItem**)?]
CaseExpression	→	`case_expr` [
		when_items: (**WhenItem**)+,
		else_item: (**ElseItem**)?]
WhenItem	→	`when_item` [
		when_expr: **Expression**,
		then_expr: **Expression**,
		(alias:`string`)?]
ElseItem	→	`else_item` [
		else_expr: **Expression**,
		(alias: `string`)?]

Figure 4.5: AST (expressions)

The parser handles operator precedence, but all operators are translated uniformly into function calls in the AST. The list of supported operators / functions include:

1. Logical operators, such as `AND`, `OR` and `NOT`.

2. Comparison operators, such as `<>, <, >, <=, >=, =` and `!=`.

3. Arithmetic operators, such as `+, -, *, /` and `**`.
4. String operators, such as `LIKE`.
5. Aggregate functions, such as `SUM`, `COUNT` and `AVG`.
6. The `EXISTS` function, which takes as input a collection. It returns true if the collection contains any tuples and false otherwise.
7. The `TUPLE_ELEMENT` function, which takes as input a collection. It returns `NULL` if the collection is empty, the tuple if the collection has a single tuple, and throws a runtime error otherwise.
8. The `SCALAR_ELEMENT` function, which takes as input a collection whose tuple has only a single scalar attribute. It returns `NULL` if the collection is empty, the scalar value if the collection has a single tuple, and throws a runtime error otherwise.
9. The `CAST` function, which takes as input (1) a scalar value and (2) a string specifying the target type. It returns the appropriately cast value if the cast succeeds, and throws a runtime error otherwise.
10. The `SELECTED_CASE_NAME` function, which takes as input a switch value. It returns the selected case name as a string value.

4.2 Semantic Analysis

4.2.1 Variable reference checking

The query language emulates SQL's variable scoping rules.

1. Within the same query, variables declared in the `FROM` clause can be referenced in expressions of all other clauses. However, variables declared in the `SELECT` clause can only be referenced in the `ORDER BY` clause. This is because joins occur before projection, and projection occurs before sorting.

2. When a sub-query occurs in the `FROM` clause, all variables declared in ancestor queries cannot be referenced in the sub-query. This is to allow the query optimizer to have the flexibility of re-ordering joins. However, the output attributes in the `SELECT` items of the sub-query can be referenced in all clauses of the parent query, since the purpose of the sub-query is to produce an intermediate collection accessible within the parent query.

3. When a sub-query occurs in the `SELECT` clause or the `WHERE` clause, all variables declared in the `FROM` clauses of ancestor queries can be referenced in the sub-query. However, the sub-query cannot reference variables declared in the parent query's `SELECT` items.

Variable reference checking consists of two steps:

1. **Variable resolution** is a recursive procedure that resolves the variable declaration corresponding to a variable reference. For each reference r_i in the AST that references variable v_i, the procedure visits r_i's nearest ancestor `PlainQuery` node q, to check if v_i is declared as an alias (tuple variable) in a `FROM` item of q. If not, the procedure recursively checks the ancestor queries of q.

2. **Attribute resolution** is a procedure to check if the path navigation from a variable reference is valid. After locating variable declaration v_i, the attribute resolution procedure checks its corresponding collection type c_i, and checks whether the path navigation from r_i is a valid path navigation from the tuple type of c_i. We currently restrict the path such that collection types cannot occur anywhere in the path except as the last component.

Note that a variable declaration must be unique within a query and its ancestor queries. That is, variable masking is disallowed. This is achieved by maintaining a symbol table that keeps track of all declared variables.

All `FROM` items and `SELECT` items are required to have aliases in the AST. If the alias has not been explicitly specified by the developer, the checker will infer an alias from the original expression. The inference uses the following defaults:

1. If the item is an attribute reference, the inferred alias is the attribute name.
2. If the item is a function call, the inferred alias is the function name.

Inferred aliases may need to be appended with numerical suffixes to that aliases are unique within a tuple.

4.2.2 Type checking / inference

Type checking and type inference is performed in a single pass by propagating types in a bottom-up fashion in the AST. Each expression node is annotated with a type after its children have been traversed. At the end of the procedure, the output schema of the query is available.

Type checking is done for the following:

1. `FROM` **item** Must be a collection type.
2. **Function call** A function's parameter types and return type are captured by a *function signature*. For each function call site, the arguments' types are checked according to the function signature.
3. `SWITCH` **constructor** Each case of the switch must be a tuple type.
4. `CASE` **expression** Each `THEN` expression of the `CASE` expression must be the identical scalar type.
5. `GROUP BY` **item** Must be scalar type.
6. `ORDER BY` **item** Must be scalar type.
7. `KEY ON` **item** Must be scalar type.
8. `UNION, INTERSECT, EXCEPT` The operands of set operators are of collection types, and their output schemas must be identical.
9. `LIMIT` **clause** The operands of `LIMIT` and `OFFSET` must be of numeric types.

4.2.3 Group by checking

The group by checking procedure checks if the query contains a valid `GROUP BY` clause, and behaves identically to SQL. When a `GROUP BY` clause is present, it is not valid for the expressions in the `SELECT` items to refer to ungrouped attributes unless the attributes are within aggregate functions.

We currently restrict expressions in the `GROUP BY` clause to only be attribute references (originating from input collections).

4.2.4 Primary key checking / inference

Each collection type must have a local primary key constraint. The constraints should be captured in the type annotations of AST nodes, and can be either declared explicitly by the developer using the `KEY ON` clause, or inferred with the following procedure:

- If a query has a `GROUP BY` clause, the local primary key attributes are the group by items. Otherwise, the local primary key attributes comprise the respective local primary key attributes of collection types in the `FROM` clause. Note that to handle aliasing and self-joins, the local primary key attributes of a collection type are used once for each `FROM` item the collection type is referenced.

- Any expressions that produce scalar constants are *constant expressions*, which include scalar, aggregate functions, top-level scalar reference, or any deterministic functions that have constant expressions as arguments. If all the `SELECT` item expressions produce constants, then all the scalars in the result are local primary key attributes.

- The `SELECT` clause must not exclude any local primary key attribute inferred from above.

- The local primary key is not applicable for a query that produces a singleton result, i.e. a query that is coerced by the `TUPLE_ELEMENT` or `SCALAR_ELEMENT` operator.

- The operands to set operators `UNION`, `INTERSECT` and `EXCEPT` must have identical local primary keys. Furthermore, `UNION` should behave as a discriminated union, i.e. the output of each operand set should contain a *discriminator attribute* that indicates which operand set an output tuple correspond to. A discriminator attribute must be a scalar literal. In the following example, `query_id` serves as the discriminator attribute:

```
1
2  SELECT 1 AS query_id, p.id, p.title
3  FROM   proposals AS p
4  WHERE  ...
5
6  UNION
7
8  SELECT 2 AS query_id, p.id, p.title
9  FROM   proposals AS p
10 WHERE  ...
11
```

4.3 Query Evaluation

In the following architecture, we describe how a *query evaluator* evaluates queries by using an existing SQL engine. This architecture deviates from the conventional approach of creating an algebraic plan for query evaluation, and is more reminiscent of XML databases that are built on top of SQL databases by translating XML data and XQuery queries accordingly.

4.3.1 Decomposition of input data

The query operates over input data that is stored in a SQL database. (For data that is transient in memory, this implies that the query evaluator will need to copy that data into the SQL database before evaluating the SQL statements). Since a SQL database does not handle data model extensions such as nested collection types, nested tuple types and switch types, nested data will undergo *decomposition* to become decomposed data that can be stored in the SQL database. The decomposed data can undergo *reconstruction* to become nested data again.

Although the main purpose of decomposition is to remove nesting (by encoding it otherwise), decomposition can also be configured to perform vertical partitioning of 1-to-1 relationships.

Decomposition can occur on the data tree, schema, or constraints. For conciseness, only decomposition of the schema is described.

Decomposition is separated into two phases: *normalization* and *concatenation*. The purpose of normalization is to eliminate nested collection types from the schema, such that only top-level collection types remain. For those top-level non-collection types in the nested schema will be moved under a special collection type named `root` after normalization. In addition, a non-collection non-key type can also be marked for normalization, so that it is split off into a separate top-level collection. On the other hand, the purpose of concatenation is to eliminate nested tuple types and switch types from the schema. During concatenation, each tuple type in the schema will concatenate its direct, indirect and conditional attributes.

Example 4.3.1 shows how a nested schema is decomposed, where Figure 4.6 shows the original nested schema, Figure 4.7 shows the schema after normalization, and Figure 4.8 shows the schema after concatenation, i.e. the decomposed schema.

To facilitate subsequent reconstruction, a mapping from the nested schema to the decomposed schema is needed. This mapping occurs through a naming convention for attributes in the decomposed schema, as illustrated by example in Figure 4.8:

1. A collection type maps to a top-level collection type named after its fully-qualified path, i.e. the string concatenation of names starting from the root. For example, `proposals.reviews`.

2. A scalar type or switch type maps to a scalar type named after its collection-relative path, i.e. the string concatenation of names starting from its closest ancestor collection type. For example, `comment`, `my_review`, `my_review.input_tuple.comment`.

3. A tuple type is not mapped.

4. Normalization will create new attributes in the decomposed schema that correspond to foreign keys. A foreign key attribute is named after the fully-qualified path of the corresponding primary key attribute. For example, `proposals.id`.

Algorithm 1 describes the recursive procedure to normalize a type X, whereas Algorithm 2 describes the recursive procedure to concatenate a type X.

EXAMPLE 4.3.1. The following example shows a nested schema and its decomposition. In this example, the `average_grade` attribute is a non-collection type that has been marked for normalization.

```
proposals : list(tuple(
    id            : int,
    reviews       : set(
        id          : int,
        comment     : string
    ),
    average_grade : float,
    my_review     : switch(
        input_tuple   : tuple(
            comment     : string
        ),
        display_tuple : tuple(
            comment     : string,
            edited      : date
        )
    )
))
```

Figure 4.6: Nested schema before decomposition

```
proposals : set(tuple(
    id           : int,
    my_review    : switch(
        input_tuple   : tuple(
            comment     : string
        ),
        display_tuple : tuple(
            comment     : string,
            edited      : date
        )
    )
))

proposals.reviews : set(tuple(
    proposals.id    : int,
    id              : int,
    comment         : string
))

proposals.average_grade : set(tuple(
    proposals.id          : int,
    average_grade         : float
))
```

Figure 4.7: Schema after normalization

```
proposals : set(tuple(
    id                                : int,
    my_review                         : string,
    my_review.input_tuple.comment     : string,
    my_review.display_tuple.comment   : string,
    my_review.display_tuple.edited    : date
))

proposals.reviews : set(tuple(
    proposals.proposal_id   : int,
    id                      : int,
    comment                 : string
))

proposals.average_grade : set(tuple(
    proposals.proposal_id : int,
    average_grade         : float
))
```

Figure 4.8: Schema after concatenation (decomposed schema)

```
function normalize(X) begin
    // Normalize in post-order
    foreach child type X_C of X do
        normalize(X_C)

    X_P ← global primary key attributes of X's parent

    if X is a nested collection type then
        Add to X's child tuple type the attributes X_P (named with respective
        fully-qualified paths)
        Detach X to become top-level collection type (named with X's
        fully-qualified path)
    else if X is a non-collection type marked for normalization then
        Y ← new top-level set type (named with X's fully-qualified path)
        Add to Y's child tuple type the attributes X_P (named with respective
        fully-qualified paths)
        Add to Y's child tuple type the attribute X (named with X's original
        name)
        Detach X
end
```

Algorithm 1: Normalizing 1-to-many (and possibly 1-to-1) types

function `concat(`X`)` **begin**
  if *X is a collection type* **then**
    `concat(`*X's child tuple type*`)`
  else if *X is a switch or tuple type* **then**
    $Y \leftarrow$ new tuple type
    foreach *descendant scalar attribute X_D of X* **do**
      Add to Y the attribute X_D (named with X_D's collection-relative path)
    foreach *descendant switch attribute X_D of X* **do**
      Add to Y a new string attribute (named with X_D's collection-relative path)
    if *X is a switch type* **then**
      Add to Y a new string attribute (named with X's collection-relative path)
    Replace X with Y
end

Algorithm 2: Concatenating nested tuple types and switch types

4.3.2 Relational decomposition of query

Query plan A *query plan* is a list of steps used to construct a nested data tree, where each step issues a SQL query on a SQL database. Given a nested query, our goal is to decompose the nested query into the multiple steps of a query plan. The query plan is represented by a *generator tree*, where each node is a generator containing a query AST.

Initially, the original AST is enclosed in the only node of a generator tree. The generator tree is rewritten by a system of *rewrite rules*. Each rewrite rule takes a generator tree as input, and outputs a generator tree where certain query fragments, corresponding to query language features that are not supported by SQL, have been removed. The rewrite rules progressively remove non-SQL features, until each generator node contains a SQL-compliant AST. The ASTs are serialized into SQL query strings, and executed on a SQL database.

Algorithms for generating query plan

1. Invoke each rewrite rule until the generator tree reaches a fixed point.

   Each rewrite rule can be invoked more than once, and the rules can be non-deterministically invoked in any order. The generator is guaranteed to reach a fixed point, since each rewrite rule only removes a non-SQL feature and does not add any non-SQL features.

   The rewrite rules are presented below as rewrite patterns. Note that:

   - The syntax <condition> denotes a list of conditions.
   - The variable `s1` is the alias for `set_1`, and is declared either in <collection> or in outer queries.
   - The *element generator* is a special kind of generator, resulting from rewriting the `TUPLE_ELEMENT` and `SCALAR_ELEMENT` functions. The element generator has an additional constraint that its result collection contains only a single tuple. Due to this property, an element generator is treated differently from a collection generator during primary key generation and reconstruction, which we will elaborate in later sections.

   (a) Switch constructor

```
generator 'g'[
    SELECT <other_attr>,
    SWITCH
        WHEN condition_1 THEN result_1 AS alias_1
        WHEN condition_2 THEN result_2 AS alias_2
        ELSE                  result_3 AS alias_3
    END AS alias
    FROM <collection>
    WHERE <condition>
]
```

   rewriting:

```
generator 'g'[
    SELECT <other_attr>,
    CASE
        WHEN condition_1 THEN 'alias_1'
        WHEN condition_2 THEN 'alias_2'
```

```
6	        ELSE                  'alias_3'
7	    END AS alias,
8	    result_1 AS "alias.alias_1",
9	    result_2 AS "alias.alias_2",
10	    result_3 AS "alias.alias_3"
11	    FROM <collection>
12	    WHERE <condition>
13	]
```

(b) Nested collection in `FROM`

```
1	generator 'g'[
2	    SELECT <attr>
3	    FROM  <collection>, s1.set_2 AS s2
4	    WHERE <condition>
5	]
```

rewriting:

```
1	generator 'g'[
2	    SELECT <attr>
3	    FROM   <collection>, "set_1.set_2" AS s2
4	    WHERE  <condition>
5	    AND    s1.id = s2.ref
6	]
```

(c) Nested select

```
1	generator 'g'[
2	    SELECT <other_attr>,
3	    ( SUB_QUERY
4	    ) AS out
5	    FROM   <collection>
6	    WHERE <condition>
7	]
```

rewriting:

```
1	generator 'g'[
2	    SELECT <other_attr>
3	    FROM  <collection>
4	    WHERE <condition>
5	]
6	
7	generator 'g.out'[
8	    SUB_QUERY
9	]
```

(d) Nested collection reference in `SELECT` clause

```
generator 'g'[
    SELECT <other_attr>, s1.set_2 AS out
    FROM  <collection>
    WHERE <condition>
]
```

rewriting:

```
generator 'g'[
    SELECT <other_attr>
    FROM   <collection>
    WHERE <condition>
]
generator 'g.out'[
    SELECT *
    FROM   s1.set_2 AS s2
]
```

(e) Tuple constructor

```
generator 'g'[
    SELECT Tuple( s1.scalar AS scalar, s1.set AS set) AS out, <other_attr>
    FROM  <collection>
    WHERE <condition>
]
```

rewriting:

```
generator 'g'[
    SELECT  s1.scalar AS "out.scalar", s1.set AS "out.set", <other_attr>
    FROM  <collection>
    WHERE <condition>
]
```

(f) Tuple element function

```
generator 'g'[
    SELECT  TUPLE_ELEMENT(SUB_QUERY) AS out, <other_attr>
    FROM  <collection>
    WHERE <condition>
]
```

rewriting:

```
generator 'g'[
       SELECT <other_attr>
```

```
3        FROM  <collection>
4        WHERE <condition>
5  ]
6  element_generator 'g.out'[
7        SUB_QUERY
8  ]
```

(g) Scalar element function

```
1  generator 'g'[
2      SELECT  SCALAR_ELEMENT(SUB_QUERY) AS out, <other_attr>
3      FROM  <collection>
4      WHERE <condition>
5  ]
```

rewriting:

```
1  generator 'g'[
2        SELECT <other_attr>
3        FROM  <collection>
4        WHERE <condition>
5
6  ]
7  element_generator 'g.out'[
8        SUB_QUERY
9  ]
```

2. Generate output schema with the select items in the AST node `Query`.

3. Annotate the local primary key in each output schema

4. Create the parent key in each generator

   In each generator, the parent key, which is a list of attributes referencing the global primary key of the parent generator, is created and added to both the AST and the output schema .

5. Build parameter mappings

   Each generator node contains a list of *parameter mapping*s. Each parameter mapping maps a parameter at a particular position in the AST to an attribute name in an ancestor's output schema.

In an AST, when a scalar reference `variable.path` is encountered, the name `variable` is looked up in the symbol table to find the generator where it is declared. If the generator is not the one that contains the current AST, then we create a new parameter mapping, and map this scalar reference to the select item in the generator's query that outputs the attribute specified by `path`.

Note that in a parameter mapping, we identify parameters by their positions instead of names, since JDBC does not support named parameters.

6. In each generator, translate the SQL compliant AST into a SQL statement.

   If the collection and attribute names contain the token "`.`", they are quoted in order in order to escape the special character.

4.3.3 Evaluating the generators

During runtime, before evaluating the query plan, each statement in the generator tree is prepared with the associated parameter mappings. The evaluation then is performed using Algorithm 3.

```
function evaluate(Generator gen) begin
    statement ← get the prepared statement in gen
    foreach parameter in gen.parameter_mappings do
        tuple ← get the active tuple from parameter.generator
        value ← get the value from tuple with the attribute name
        parameter.name
        Set the parameter in statement of parameter.position with value
    collection ← construct collection with the evaluation result of statement
    append collection to gen.result_collection
    foreach tuple in collection do
        Set tuple as the active tuple of current generator gen
        foreach child_gen of gen's children do
            evaluate(child_gen)
end
```

Algorithm 3: Evaluating the query plan

4.3.4 Reconstructing the nested output data tree

Algorithms 4 and 5 illustrate how the nested output data tree can be reconstructed, given the nested schema X_S and the decomposed output data tree y_{all}. This is made possible by the naming convention presented in Section 4.3.1 that provides a mapping from the nested schema to the decomposed schema. The high-level idea is that the algorithm first finds the top-level collection (in order to obtain all primary key attributes), thereafter joins this collection (i.e. *denormalizes* it) with the remaining decomposed collections. During denormalizing, the decomposed collection is also *unconcatenated* to restore the original nested tuples and switches.

function `reconstruct(`X_S, y_{all}`)` **begin**
    foreach *decomposed collection* y_c *of* y_{all} *(in increasing order of number of parts in name)* **do**
        $X \leftarrow$ type in X_S with fully-qualified path that matches y_c's name
        if X *is a top-level collection type* **then**
            $x \leftarrow$ `denormalize(`X, y_c`)`
        else
            $X_T \leftarrow$ parent tuple type of X
            foreach *tuple* x_t *that is a descendant in* x *and corresponds to* X_T **do**
                $y_\sigma \leftarrow$ subset of y_c that matches the global primary key of x_t
                add to x_t the child attribute `denormalize(`X, y_σ`)`
    return x
end

Algorithm 4: Reconstructing a nested data tree from a decomposed data tree

```
function denormalize(X, y_σ) begin
    if X is a collection type then
        x ← new collection
        foreach child tuple y_t of y_σ do
            add to x the child tuple unconcat(child tuple type of X, y_t)
    else
        y_t ← the only tuple in y_σ
        x ← unconcat(X, y_t)
    return x
end

function unconcat(X, y_t) begin
    switch X do
        case collection type
            x ← null
        case tuple type
            x ← new tuple
            foreach child attribute X_A of X do
                add to x the child attribute unconcat(X_A, y_t)
        case switch type
            y ← scalar value in y_t (named with X's collection-relative path)
            x ← new switch value using y
            foreach child case X_A of X do
                add to x the child case unconcat(X_A, y_t)
        case scalar type
            y ← scalar value in y_t (named with X's collection-relative path)
            x ← new scalar value using y
    // x may be null if y_t does not have the corresponding
       attribute(s)
    return x
end
```

Algorithm 5: Reconstructing a nested data tree from a decomposed data tree

4.4 Data Sources

In the general case, a query can be issued over different data sources, such as SQL databases, in-memory data, LDAP databases, WSDL / XML-RPC web services etc. A *data source* is a system that:

1. allows reading and writing data.
2. allows data to be transient and/or persistent. Data are physically stored either on volatile storage such as main memory or durable storage such as hard drives.
3. is self-consistent. It should provide an exclusive interface to the data, so that other programs cannot bypass the system to modify the data. Typically, a data source provides synchronization primitives such as locks and transactions, so that writers can maintain the consistency of the data stored.

As an example for self-consistency, multiple text files can be considered a single data source if all of them are exclusively locked before a write to any occurs. Conversely, even if two databases are hosted within the same SQL engine, they are considered separate data sources since SQL transactions provide ACID guarantees within a single database, but not across databases.

For the time being, we focus our attention on SQL databases and in-memory data. We assume for simplicity that a query executes over a single data source, as executing queries over multiple data sources require distributed transactions. One consequence of the single source assumption is that the query engine needs to copy in-memory data into the SQL database before the query is executed.

4.4.1 Local versus Remote Data Sources

We distinguish between *local data sources* and *remote data sources.* A local data source is one where data can be directly accessed, whereas a remote data source is one where data can only be indirectly accessed through a query language. Note that a local data source does not have to be transient data in memory: it may

provide an abstraction over persistent files by reading from disk and writing to disk on-demand. For the time being however, we focus on in-memory data structures (within the JVM) as the only local data source, and a JDBC/SQL database as the only remote data source.

We restrict the data model to allow only a collection type to have a *data source mapping*, which associates the collection type with a data source, and maps the collection type to a specific data structure in the data source. For example, a collection type can be mapped to a SQL table by specifying the (namespace, table name) pair. We further restrict that collection types nested within each other must be associated with the same data source, otherwise the corresponding data tree is not type consistent.

When a collection type is associated with a local data source, the corresponding collection recursively contains data within the same data source. For a remote data source though, there are no corresponding data nodes to the schema nodes, since the data can only be indirectly accessed. For example, Figure 4.9b shows a schema tree containing collection types associated either with a local data source or a remote data source. Figure 4.9a shows that the data tree has a collection that corresponds to the top-level collection type of a local data source, and only an empty set value that corresponds to the top-level collection type of a remote data source.

In the general case where the root data node is not a collection value, the values need to be copied into a special `root` table.

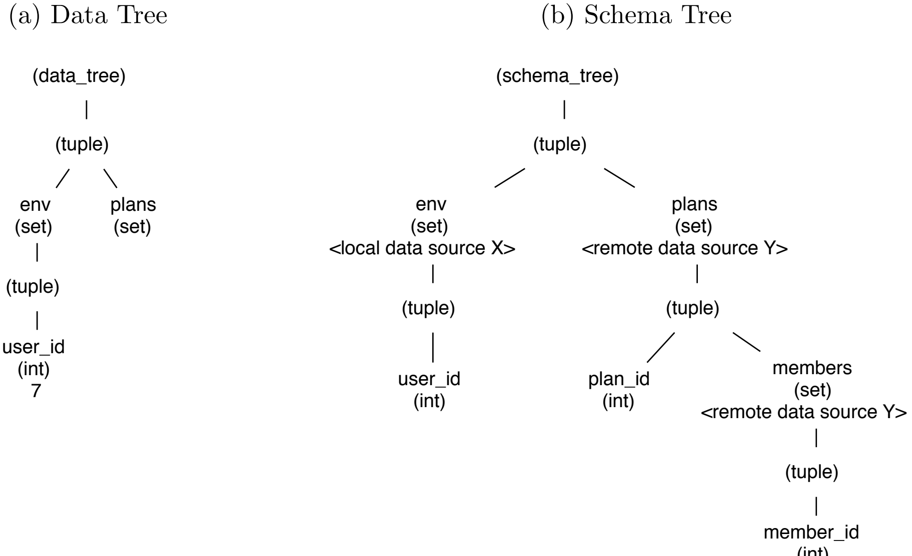


Figure 4.9: Data and Schema Tree for Local / Remote Data Sources

4.4.2 Data Source Catalog

A data source has an *initialization configuration.* For example, a JDBC/SQL remote data source's initialization configuration can comprise the driver name, the connection URL and authentication credentials.

A data source also has a *catalog* that stores and provides metadata about the data source. These metadata include:

1. A list of *supported scalar types.* A data source may choose to support only certain scalar types (e.g. no binary data type), or require certain scalar types to be explicitly declared.

2. A *persistent schema tree* that contains the corresponding types of all persistent data, i.e. the schema of the data source. Any schema tree that needs to refer to the data tree persisted in a particular data store will do so by cloning the data store's persistent schema tree.

Acknowledgements

This chapter contains material from the technical report "Technical Specifications of the FORWARD Framework", by Yupeng Fu, Kian Win Ong, Kevin Keliang Zhao, Yannis Papakonstantinou and Michalis Petropoulos. The dissertation author was the primary investigator and author of the relevant chapters in the technical report.

Chapter 5

Incremental View Maintenance

The incremental view maintenance module uses the log of modifications that happened to the database data, and possibly the database data itself, to incrementally maintain an old view instance to a new view instance.

5.1 Delta Tuples

A *database instance* comprises a set of named *base data trees* $\{D_1, D_2, \ldots, D_m\}$. We write $D_i(s)$ to denote the data tree D_i when the database instance is in state s. Each base data tree D_i can be decomposed into a set of flat *base relations* $\{R_{i1}, R_{i2}, \ldots, R_{in}\}$ by the decomposition algorithm in Section 4.3.1. The base relations correspond to the SQL tables stored in the underlying SQL database.

A *modification log* L is a collection of records shared in an application that captures all modifications, including inserts, updates and deletes, to each base data tree D_i up to the current state s_c. A subset of the modification log $L_{D_i}(s_p, s_c)$ records all modifications to the data tree D_i between a past state s_p and current state s_c. Whenever an insert, replace, or delete command in the update language is issued, a corresponding modification record will be added to the log. A modification log record has:

1. The name of the target data tree.

2. A data diff, as defined in Section 3.3.

3. A set of *flat deltas*, which is a collection of flat Δ^+ and Δ^- tuples, resulting from the decomposition of the data diff. Deltas record the actual modifications to the base relations. The decomposition of data diffs is similar to the decomposition of data trees, with the additional requirement that replace and delete diffs will produce flat deltas corresponding to the old values before the modification. It is the responsibility of the modification commands to provide these old flat deltas. This requirement allows the incremental view maintenance to handle replace commands as a pair of delete / insert commands.

4. The *version number* of the database instance. The modification log itself maintains a version counter that is incremented for each modification log record.

Thus, a *base data tree difference*, which is the diff between base data trees $D(s_p)$ and $D(s_c)$, can be equivalently represented by a set of *base relation delta tuples*, which are the delta tuples in $L_D(s_p, s_c)$ for each base relation R_i, $i = 1, \ldots, n$ of D. Since

$$R_i(s_c) = R_i(s_p) + \Delta^+ R_i - \Delta^- R_i$$

Rearranging, we have:

$$R_i(s_p) = R_i(s_c) - \Delta^+ R_i + \Delta^- R_i$$

The above allows computing the past state from the current state, which will be utilized in later sections for deferred view maintenance.

Assumptions Two flat tuples are equal if and only if all attribute values are equal to each other. We assume that there are no redundancies in Δ^+ and Δ^-, namely:

1. $R_i(s_p) \bigcap \Delta^+ R_i = \emptyset$: An inserted tuple is not in the past state of R_i.

2. $R_i(s_c) \bigcap \Delta^- R_i = \emptyset$: A deleted tuple is not in the current state of R_i.

3. $\Delta^+ R_i \bigcap \Delta^- R_i = \emptyset$: No tuple is deleted and again inserted, or inserted and again deleted.

To ensure the above, retrieving delta tuples between states s_p and s_c from the modification log requires the following post-processing: if a tuple occurs in both $\Delta^+ R_i$ and $\Delta^- R_i$, it would be removed from both in the returned delta tuples.

5.2 Leveraging Relational Incremental View Maintenance

A *view* is a query over the database, and a *view instance* is the output data tree of the query. Given view V with query Q, and data trees D_i, $i = 1, \ldots, m$ in state s, the view instance $V(s)$ is:

$$V(s) = Q(D_1(s), D_2(s), \ldots, D_m(s))$$

Recall in Chapter 4, in order to evaluate a nested query Q, we decompose Q into a sequence of parameterized SQL-compliant queries q_i, $i = 1, \ldots, l$, and each decomposed query q_i is over the decomposed base relations R_j, $j = 1, \ldots, n$ and results in a *decomposed view instance* $v_i = q_i(R_1, \ldots, R_n)$.

For each such decomposed view instance v, its flat tuples with respect to states s_p and s_c are defined as:

$$\Delta^+ v = v(s_c) - v(s_p)$$

$$\Delta^- v = v(s_p) - v(s_c)$$

An efficient incremental view maintenance algorithm on the relational model will compute $\Delta^+ v$ and $\Delta^- v$ without having to explicitly compute $v(s_c)$ and compare it with $v(s_p)$. For example, given a relational algebra expression $v = R_1 \bowtie R_2$, if insertions are the only modifications (i.e. only $\Delta^+ R_1$ and $\Delta^+ R_2$ occur), then $\Delta^+ v$ can be efficiently computed using $(\Delta^+ R_1 \bowtie R_2(s_c)) \cup (R_1(s_c) \bowtie \Delta^+ R_2)$.

Since the query language implementation in Chapter 4 utilizes rewrite rules to rewrite to valid SQL (as opposed to generating an query plan of algebraic operators), we also implement a relational incremental view maintenance algorithm based on *rewrite rules*. For a SQL query $q(R_1, \ldots, R_n)$, the algorithm computes the flat deltas by evaluating two *incremental queries* $\Delta^+ q$ and $\Delta^- q$, which are created by modifying q using rewrite rules. That is,

$$\Delta^+ v = \Delta^+ q(R_1(s_c), \ldots, R_n(s_c), \Delta^+ R_1, \ldots, \Delta^+ R_n, \Delta^- R_1, \ldots, \Delta^- R_n)$$

$$\Delta^- v = \Delta^- q(R_1(s_c), \ldots, R_n(s_c), \Delta^+ R_1, \ldots, \Delta^+ R_n, \Delta^- R_1, \ldots, \Delta^- R_n)$$

The incremental view maintenance of decomposed queries q_i in a generator tree is performed similar to the (full) query evaluation algorithm, in that it traverses the queries in the generator tree top down to incrementally maintain each decomposed view v_i as follows:

1. For the top level query q_1, compute the flat deltas $\Delta^+ v_1$ and $\Delta^- v_1$ by evaluating the incremental queries $\Delta^+ q_1$ and $\Delta^- q_1$
2. Using $\Delta^+ v_1$, $\Delta^- v_1$ and the materialized past view $v_1(s_p)$ as the context, populate the parameters in the descendant generators' queries
3. For each descendant query q_i, recursively compute the flat deltas $\Delta^+ v_i$, $\Delta^- v_i$ as in 1. The evaluation algorithm will be elaborated more in section 5.4.

Finally, a *reconstruction* algorithm takes all the decomposed views' deltas, and builds a collection of diff data trees representing the nested view's difference $\Delta^+ V$ and $\Delta^+ V$.

5.3 Rewrite Rules to Create Incremental Queries

As will be seen in the query patterns / rewrite rules below, based on the structure of the query pattern, a base relation can be classified as *safe* or *unsafe*. A

safe query is one where all its base relations in the query are safe. The significance of a safe query is that it can be incrementally maintained efficiently through the evaluation of its corresponding incremental queries.

For a safe SQL query q, there are two incremental queries $\Delta^{+}q$, $\Delta^{-}q$ that serve to compute the flat deltas directly. The system has a set of *rewrite rules*, each rewrite rule has a pattern which it matches q against, and if the match succeeds, creates the appropriate incremental queries. We assume that q can be matched by at most one pattern. Note that the incremental queries are issued over the base relations at the current state s_c, thus each rewrite rule needs to perform deferred view maintenance.

The following rewrite rules have assumed the knowledge of primary key constraints in order to identify rewriting opportunities. As illustrated by previous work on incremental view maintenance [QGMW96, Vis98, LZ07], further optimization opportunities can be identified by exploiting foreign key constraints as well.

1. Base relations participate in joins at top-level `FROM` clause

   Suppose q is

   ```
   1   SELECT  <attr>
   2   FROM    S1 AS s_1, S2 AS s_2 , ... ,Sn AS s_n
   3   WHERE   <condition>
   ```

   where `S1,...,S2` are base relations. Note the absence of nested sub-queries in the `condition`, and the absence of a `GROUP BY` clause in the query.

   A base relation `Si` is safe in the query if one of the following holds:

   (a) the key of `Si` is in the `SELECT` clause of q.

   (b) `condition` includes an equality comparison between the key of `Si` and a constant.

   (c) `condition` includes an equality comparison between the key of `Si` and the key of another safe relation `Sj`.

Note that (c) usually occurs when `Sj` is a weak entity of `Si`. For example, if a Countries collection has a nested Cities collection in the nested schema, in the flat relational schema the `cities` relation will have among its primary key attributes a foreign key to the `countries` relation. Joining on the foreign key in the query will satisfy (c).

Recall also in Chapter 4 that we require the `SELECT` clause of a query q to always include the primary key attributes of all base relations in the FROM clause (provided there is no `GROUP BY` clause). Therefore, (a) is trivially satisfied, each base relation is always safe in query q, and q is always a safe query.

After rewriting, the incremental query $\Delta^{+}q$ is:

```
1   SELECT  <attr>
2   FROM    delta_plus_S1 AS s1, S2 AS s_2, ..., Sn AS s_n
3   WHERE   <condition>
4
5   UNION
6
7   SELECT  <attr>
8   FROM    S1 AS s_1, delta_plus_S2 AS s_2, ..., Sn AS s_n
9   WHERE   <condition>
10
11  UNION
12  ...
13  UNION
14
15  SELECT  <attr>
16  FROM    S1 AS s_1, S2 AS s_2, ..., delta_plus_Sn AS s_n
17  WHERE   <condition>
```

We can see the incremental query is a union expression where each sub-query contains the delta plus relations `delta_plus_Si` for each base relation `Si`. Such a sub-query is named the *incremental query component* for base relation `Si`.

For the following rewrite rules to be more concise, we will use `PAST_Si` as a syntactic shortcut for the value of relation `Si` at the past state s_p. `PAST_Si` is translated to the sub-query:

```
SELECT * FROM Si
EXCEPT
SELECT * FROM delta_plus_Si
UNION
SELECT * FROM delta_minus_Si
```

Thus, the incremental query $\Delta^{-}q$ is:

```
    SELECT  <attr>
    FROM    delta_minus_S1 AS s1, PAST_S2 AS s_2, ..., PAST_Sn AS s_n
    WHERE   <condition>

    UNION

    SELECT  <attr>
    FROM    PAST_S1 AS s_1, delta_minus_S2 AS s_2, ..., PAST_Sn AS s_n
    WHERE   <condition>

    UNION
    ...
    UNION

    SELECT  <attr>
    FROM    PAST_S1 AS s_1, PAST_S2 AS s_2, ..., delta_minus_Sn AS s_n
    WHERE   <condition>
```

Note that the rule is based on each separate `FROM` item. Even if a relation appears more than once in the top-level `FROM` clause, the aliases will be considered separately.

2. Positively nested sub-queries

   A *positively nested sub-query* is a nested `SELECT` expression that is immediately enclosed by an `EXISTS`.

   Consider the query q with one positively nested level of sub-queries as follows:

```
    SELECT  <attr>
    FROM    S1 AS s_1, S2 AS s_2 , ... ,Sn AS s_n
    WHERE   <condition1> AND
            EXISTS (
                SELECT  <attr>
                FROM    N1 AS n_1, N2 AS n_2 , ... ,Nl AS n_l
```

```
7                 WHERE   <condition2>
8             )
```

The relations `N1,...,Nl` are safe, if they are *correlated* to the top-level relations `S1,...,Sn`. `Ni` is correlated if one of the following holds:

(a) `Ni` is one of the top-level relations `S1,...,Sn`.

(b) `condition2` includes an equality comparison between the key of `Ni` and the key of one of the top-level relations `S1,...,Sn`.

(c) `condition2` includes an equality comparison between the key of `Ni` and a constant.

(d) `condition2` includes an equality comparison between the key of `Ni` and the key of a correlated relation `Nj`.

If the query has more than one positively nested level, the definition of correlation can be similarly extended.

Suppose all the relations `N1,...,Nl` in q are correlated, then incremental query $\Delta^+ q$ is:

```
1     SELECT  <attr>
2     FROM    delta_plus_S1 AS s_1, S2 AS s_2 , ... , Sn AS s_n
3     WHERE   <condition1> AND
4             EXISTS (
5                 SELECT  <attr>
6                 FROM    N1 AS n_1, N2 AS n_2 , ... , Nl AS n_l
7                 WHERE   <condition2>
8             )
9
10    UNION
11    ...
12    UNION
13
14    SELECT  <attr>
15    FROM    S1 AS s_1, S2 AS s_2 , ... , delta_plus_Sn AS s_n
16    WHERE   <condition1> AND
17            EXISTS (
18                SELECT  <attr>
19                FROM    N1 AS n_1, N2 AS n_2 , ... , Nl AS n_l
20                WHERE   <condition2>
21            )
```

```
    UNION

    SELECT  <attr>
    FROM    S1 AS s_1, S2 AS s_2 , ... , Sn AS s_n
    WHERE   <condition1> AND
            EXISTS (
                SELECT  <attr>
                FROM    delta_plus_N1 AS n_1, N2 AS n_2 , ... , Nl AS n_l
                WHERE   <condition2>
            )

    UNION
    ...
    UNION

    SELECT  <attr>
    FROM    S1 AS s_1, S2 AS s_2 , ... , Sn AS s_n
    WHERE   <condition1> AND
            EXISTS (
                SELECT  <attr>
                FROM    N1 AS n_1, N2 AS n_2 , ... , delta_plus_Nl AS n_l
                WHERE   <condition2>
            )
```

The incremental queries for a query with multiple levels of positively nested sub-queries can be derived in a similar way.

The incremental query $\Delta^{-}q$ is similar to $\Delta^{+}q$, but with the following changes: (1) replace `delta_plus_relation` with `delta_minus_relation` (2) replace the current state of each relation with its past state.

If there are unsafe relations in the sub-query, it is still possible to perform the incremental maintenance when there are only insertions to the base relations. The incremental view $\Delta^{+}v$ will contain tuples from the result of $\Delta^{+}q$, but excluding those from the old view $v(s_p)$. Suppose the old view is stored in the relation `V`, then each incremental query component for the unsafe base relation `Ni` will cause an additional predicate to be added to the condition of the outer query, as such:

```
    SELECT  <attr>
    FROM    S1 AS s_1, S2 AS s_2 , ... , Sn AS s_n
```

```
3        WHERE   <condition1> AND
4                EXISTS (
5                    SELECT  <attr>
6                    FROM    N1 AS n_1, N2 AS n_2 , ...,
7                            delta_plus_Ni AS n_i ,... , Nl AS n_l
8                    WHERE   <condition2>
9                )
10               AND NOT IN (SELECT * FROM V)
```

However, delete and update modifications may require recomputation of the view.

3. Negatively nested sub-queries

A *negatively nested sub-query* is a nested `SELECT` expression that is immediately enclosed by an `EXISTS`. Consider the query q with one negatively nested level of sub-query of the form:

```
1        SELECT  <attr>
2        FROM    S1 AS s_1, S2 AS s_2 , ... , Sn AS s_n
3        WHERE   <condition1> AND
4                NOT EXISTS (
5                    SELECT  <attr>
6                    FROM    N1 AS n_1, N2 AS n_2 , ... , Nl AS n_l
7                    WHERE   <condition2>
8                )
```

For negatively nested sub-queries, the high-level idea is that insert modifications on base relations in the sub-query result in delete modifications on the view, whereas delete modifications on base relations in the sub-query result in insert modifications on the view.

Similarly, the availability of incremental queries depends on whether base relations in the sub-query are correlated to base relations in the top-level query. The incremental query $\Delta^{+}q$ is:

```
1        SELECT  <attr>
2        FROM    delta_plus_S1 AS s_1, S2 AS s_2 , ... , Sn AS s_n
3        WHERE   <condition1> AND
4                NOT EXISTS (
5                    SELECT <attr>
6                    FROM  N1 AS n_1, N2 AS n_2 , ... , Nl AS n_l
```

```
7                 WHERE <condition2>
8             )
9
10    UNION
11    ...
12    UNION
13
14    SELECT  <attr>
15    FROM    S1 AS s_1, S2 AS s_2 , ... , delta_plus_Sn AS s_n
16    WHERE   <condition1> AND
17            NOT EXISTS (
18                SELECT  <attr>
19                FROM    N1 AS n_1, N2 AS n_2 , ... , Nl AS n_l
20                WHERE   <condition2>
21            )
22
23    UNION
24
25    SELECT  <attr>
26    FROM    PAST_S1 AS s_1, PAST_S2 AS s_2 , ... , PAST_Sn AS s_n
27    WHERE   <condition1> AND
28            EXISTS (
29                SELECT  <attr>
30                FROM    delta_minus_N1 AS n_1,PAST_N2 AS n_2 , ... , PAST_Nl AS n_l
31                WHERE   <condition2>
32            )
33
34    UNION
35    ...
36    UNION
37
38    SELECT  <attr>
39    FROM    PAST_S1 AS s_1, PAST_S2 AS s_2 , ... , PAST_Sn AS s_n
40    WHERE   <condition1> AND
41            EXISTS (
42                SELECT  <attr>
43                FROM    PAST_N1 AS n_1, PAST_N2 AS n_2 , ... , delta_minus_Nl AS n_l
44                WHERE   <condition2>
45            )
```

and $\Delta^{-}q$ is:

```
1     SELECT  <attr>
2     FROM    delta_minus_S1 AS s_1, PAST_S2 AS s_2 , ... , PAST_Sn AS s_n
3     WHERE   <condition1> AND
4             NOT EXISTS (
5                 SELECT  <attr>
```

```
6                 FROM    PAST_N1 AS n_1,PAST_N2 AS n_2 , ... , PAST_Nl AS n_l
7                 WHERE   <condition2>
8             )
9
10    UNION
11    ...
12    UNION
13
14    SELECT  <attr>
15    FROM    PAST_S1 AS s_1, PAST_S2 AS s_2 , ... , delta_minus_Sn AS s_n
16    WHERE   <condition1> AND
17            NOT EXISTS (
18                SELECT  <attr>
19                FROM    PAST_N1 AS n_1, PAST_N2 AS n_2 , ... , PAST_Nl AS n_l
20                WHERE   <condition2>
21            )
22
23    UNION
24
25    SELECT  <attr>
26    FROM    S1 AS s_1, S2 AS s_2 , ... , Sn AS s_n
27    WHERE   <condition1> AND
28            EXISTS (
29                SELECT  <attr>
30                FROM    delta_plus_N1 AS n_1,N2 AS n_2 , ... , Nl AS n_l
31                WHERE   <condition2>
32            )
33
34    UNION
35    ...
36    UNION
37
38    SELECT  <attr>
39    FROM    S1 AS s_1, S2 AS s_2 , ... , Sn AS s_n
40    WHERE   <condition1> AND
41            EXISTS (
42                SELECT  <attr>
43                FROM    N1 AS n_1, N2 AS n_2 , ... , delta_plus_Nl AS n_l
44                WHERE   <condition2>
45            )
```

Notice that in the incremental queries, the sub-query's `NOT EXISTS` has been converted to `EXISTS`.

If the nested relations are not safe, incremental maintenance can still be performed provided only delete modifications have occurred on base relations.

As previously, exclude tuples from the old view $v(s_p)$ from the result of Δ^+q, by adding the predicate `NOT IN V` to the incremental query components.

4. Aggregation

We only consider queries with aggregate functions that are used directly as `SELECT` items, i.e. the aggregate functions are not used as arguments of other functions. Without loss of generality, consider a query with only one aggregate function, which takes as argument the attribute of the first base relation in the `FROM` clause:

```
SELECT   <group_by_attr>, aggregate(s_1.aggr_attr) AS aggr_attr
FROM     S1 AS s_1, S2 AS s_2, ..., Sn AS s_n
WHERE    <condition>
GROUP BY <group_by_attr>

```

Unlike preceding rules, insert / delete modifications on the base relations whose attributes are used in aggregate functions may result in update modifications on the view. This occurs when the old view $v(s_p)$ already has tuples whose grouped by attributes have the same value as the delta plus tuples, which indicates that these tuples should be removed by the incremental query Δ^-q.

Each aggregate function has its own rewrite rule:

(a) `SUM` function

First we consider insert modifications. Where relation `V` stores the materialized old view, the incremental query Δ^+q is:

```
SELECT   <group_by_attr>,
         SUM(s_1.aggr_attr) + CASE
             WHEN V.aggr_attr IS NULL THEN 0
             ELSE V.aggr_attr
         END AS aggr_attr
FROM     delta_plus_S1 AS s_1 LEFT OUTER JOIN V, S2 AS s_2, ..., Sn AS s_n
WHERE    <condition> AND <group_by_attr> = <V.group_by_attr>
GROUP BY <group_by_attr>

UNION

```

```
12  SELECT   <group_by_attr>, SUM(s_1.aggr_attr) AS aggr_attr
13  FROM     S1 AS s_1, delta_plus_S2 AS s_2, ..., Sn AS s_n
14  WHERE    <condition>
15  GROUP BY <group_by_attr>
16
17  UNION
18
19  SELECT   <group_by_attr>, SUM(s_1.aggr_attr) AS aggr_attr
20  FROM     S1 AS s_1, S2 AS s_2, ..., delta_plus_Sn AS s_n
21  WHERE    <condition>
22  GROUP BY <group_by_attr>
```

Since the insertion on `s_1` may cause an update on the view, the incremental query $\Delta^{-}q$ is required to remove tuples from the old view that are affected by this modification:

```
1  SELECT   <group_by_attr>, V.aggr_attr AS aggr_attr
2  FROM     delta_plus_S1 AS s_1 JOIN V, PAST_S2 AS s_2, ..., PAST_Sn AS s_n
3  WHERE    <condition> AND <group_by_attr> = <V.group_by_attr>
4  GROUP BY <group_by_attr>
```

For the delete modification, currently we always recompute the view.

5. Set operators

For a query that has set operators, the rewrite rule invokes respective rewrite rules applicable for each `SELECT` expression separately, thereafter apply the set operators to the flat delta tuples. Consider the query q of the form:

```
1  SELECT <attr1> FROM <relations1> WHERE <condition1>
2  UNION
3  SELECT <attr2> FROM <relations2> WHERE <condition2>
4  ...
5  UNION
6  SELECT <attrK> FROM <relationsK> WHERE <conditionK>
```

We first consider the incremental query $\Delta^{+}q$. For each `SELECT` expression, an initial set of incremental query components is generated by using the corresponding rewrite rules, thus these components are assembled using `UNION`. To ensure that duplicates are not added to the incremental view, each tuple created by the incremental query is checked to not be in the old view.

Generating incremental query $\Delta^{-}q$ is more complicated. Even if a tuple is no longer produced by one of the `SELECT` expressions, it can only be deleted from the view if it is also not produced by any of the other `SELECT` expressions. Without loss of generality, consider the incremental query component for the first `SELECT` expression in q. The following conjunct must be added to the WHERE clause of the component:

```
AND <attr1> NOT IN (SELECT <attr2> FROM <relations2> WHERE <condition2>)
AND ...
AND <attr1> NOT IN (SELECT <attrK> FROM <relationsK> WHERE <conditionK>)
```

Such conjuncts may result in incremental view maintenance being comparable to, or more expensive than, reevaluating the original query.

6. `CASE` expression

   Consider a query q with CASE expression of the form:

```
SELECT  <attr>,
        CASE
            WHEN condition1 THEN expression1
            WHEN condition2 THEN expression2
            ...
            WHEN conditionK THEN expressionK
            ELSE expression
        END
FROM    <relations>
WHERE   <condition>
```

The conditions `condition1, ..., conditionK` are mutually exclusive, and the query can be rewritten to an equivalent query with the `UNION` operator as such:

```

SELECT  <attr>, expression1
FROM    <relations>
WHERE   <condition> AND condition1

UNION

SELECT  <attr>, expression2
FROM    <relations>
```

```
10  WHERE   <condition> AND condition2
11
12  UNION
13  ...
14  UNION
15
16  SELECT  <attr>, expressionK
17  FROM    <relations>
18  WHERE   <condition> AND conditionK
19
20  UNION
21
22  SELECT  <attr>, expression1
23  FROM    <relations>
24  WHERE   <condition> AND NOT condition1 AND NOT condition2 AND ... AND NOT conditionK
```

Thus this rewrite rule can be reduced to the `UNION` operator rewrite rule.

7. `ORDER BY` clause

The incremental view maintenance strategy for ordering is to represent the flat deltas $\Delta^{+}v$ or $\Delta^{-}v$ in arbitrary order, apply them to the old view, thereafter sort the new view according to the specified order.

Consider query q with an `ORDER BY` clause in the format:

```
1  uq(R1, R2, ..., Rn)
2  ORDER BY <order_by_attr>
```

where `uq` is an unordered query with base relations `R1,...,Rn`, and `<order_by_attr>` is a list of attributes declared in the `ORDER BY` clause of `uq` to specify the order of the returned tuples. Furthermore, the query language requires that the `SELECT` clause includes the attributes in the `ORDER BY` clause, i.e. ordering attributes cannot be omitted from the results. Note that we require that `uq` only contains the features specified by the preceding rewrite rules.

The incremental query $\Delta^{+}q$ is identical to $\Delta^{+}uq$, similarly the incremental query $\Delta^{-}q$ is identical to $\Delta^{-}uq$. The new view $v(s_c)$ is obtained by issuing the following query:

```
1  SELECT * FROM V
2  EXCEPT
3  SELECT * FROM delta_minus_V
4  UNION
5  SELECT * FROM delta_plus_V
6  ORDER BY <order_by_attr>
```

where relation `V` captures the old view of $v(s_p)$, and `delta_plus_V`, `delta_minus_V` correspond to $\Delta^+ v$ and $\Delta^- v$ respectively.

8. Top-K tuples

Retrieving the top-K tuples in SQL is achieved with the `LIMIT` clause. Note that when using the `LIMIT` clause, the `ORDER BY` clause must also be present so that the results are deterministic. Consider the query q with a `LIMIT` clause in the format:

```
1  uq(R1, R2, ..., Rn)
2  ORDER BY <order_by_attr>
3  LIMIT    K
```

where `uq` is an unordered query over base relations `R1,...,Rn`, `<order_by_attr>` is the list of ordered by attributes, and `K` is the maximum number of tuples to return.

The high-level idea is as follows. For tuples that are in $\Delta^+ v$ but not in the old view, we insert them into the old view at the proper ordinal positions (since the view is ordered), and only return the specified maximum number of tuples for the new view. For the tuples in $\Delta^- v$, it is possible that after excluding these tuples from the old view, the number of tuples left is smaller than the K. The remaining tuples will need to be retrieved by re-evaluating the query. Instead of repeatedly re-evaluating the query (for example, when pagination occurs on the web application), the system caches the next C tuples, where C is a configurable number.

First, we modify the original query q into q' as follows:

```
uq(R1, R2, ..., Rn)
ORDER BY <order_by_attr>
LIMIT    K+C
```

where we change `K` to `K+C`, such that when we initialize the view or recompute the view, we retrieve at most `C` more tuples.

The incremental query $\Delta^{+}q$ is:

```
delta_plus_uq(R1, R2, ..., Rn)
ORDER BY <order_by_attr>
LIMIT    K+C
```

where `delta_plus_uq` is the incremental query $\Delta^{+}uq$, which can be created using the corresponding rewrite rules.

The incremental query $\Delta^{-}q$ is:

```
delta_minus_uq(R1, R2, ..., Rn)
ORDER BY <order_by_attr>
LIMIT    K+C
```

where similarly `delta_minus_uq` is the incremental query $\Delta^{-}uq$.

Finally, we obtain a view $v'(s_c)$ by issuing:

```
SELECT * FROM V
EXCEPT
SELECT * FROM delta_minus_V
UNION
SELECT * FROM delta_plus_V
ORDER BY <order_by_attr>
LIMIT    K+C
```

where relation `V` captures the old view of $v(s_p)$, and `delta_plus_V`, `delta_minus_V` correspond to $\Delta^{+}v$ and $\Delta^{-}v$ respectively. The view $v'(s_c)$ has at most $K + C$ tuples. If it has more than K tuples, we leave it as is, otherwise we re-evaluate query q' to obtain the new view.

The reconstruction of the new nested view $V(s_c)$ will use only the top K tuples in the new view $v(s_c)$ to obtain the correct data tree.

Creating incremental queries on demand Depending on the original query and the number of base relations in the `FROM` clause, the generated incremental queries can become very complex.

From the rewrite rules, we observe that the incremental query is the union of query components, each of which is associated with a single flat delta relation. Therefore, instead of statically determining incremental queries that contains all query components, the system creates incremental queries dynamically at run time so that only query components corresponding to non-empty delta relations need to be included in the incremental queries.

5.4 Evaluation of incremental query

After the incremental queries have been created, for each node i in the generator tree, the *incremental query evaluator* computes the flat deltas $\Delta^+ v_i$ and $\Delta^- v_i$ of query q_i.

Recall from Chapter 4 that a query q_k at level k of the generator tree has respective join relationships with each of its $k-1$ ancestor queries. That is, q_k can be expressed as $q_k(R_1, \ldots, R_n, v_1, \ldots, v_{k-1})$, where $R_1, \ldots, R_n$ are the base relations used in q_k, and $v_1, \ldots, v_{k-1}$ are the respective views of the ancestor queries $q_1, \ldots, q_{k-1}$. If all base relations $R_1, \ldots, R_n$ are safe, then the rewrite rules will produce incremental queries for q_k. Notice that all views $v_1, \ldots, v_{k-1}$ are safe, since their primary keys are used in join conditions with the parent key of q_k. Thus, we can derive the incremental queries for q_k using the rewrite rules in Section 5.3. For example,

$$
\begin{aligned}
\Delta^+ q_k = & q_k(\Delta^+ R_1, \ldots, R_n(s_c), v_1(s_c), \ldots, v_{k-1}(s_c)) \cup \cdots \cup \\
& q_k(R_1(s_c), \ldots, \Delta^+ R_n, v_1(s_c), \ldots, v_{k-1}(s_c)) \cup \\
& q_k(R_1(s_c), \ldots, R_n(s_c), \Delta^+ v_1, \ldots, v_{k-1}(s_c)) \cup \cdots \cup \\
& q_k(R_1(s_c), \ldots, R_n(s_c), v_1(s_c), \ldots, \Delta^+ v_{k-1})
\end{aligned}
$$

if q does not have a negatively nested sub-query. We can obtain $\Delta^- q_k$ in a similar fashion.

Thus, the evaluation algorithm operates as follows. At each generator node i, if the original query q_i is safe, the algorithm obtains the old views and flat delta tuples from each ancestor generator node, and evaluates the incremental queries Δ^+q_i and Δ^-q_i to produce the flat deltas for q_i. If however q_i is unsafe, the algorithm re-evaluates q_i to obtain the new view, and manually compares it to the old view to obtain the flat deltas. The children generators are similarly evaluated, in a pre-order traversal of the generator tree.

There are two alternative ways of incremental query evaluation.

1. Parameterized evaluation

   The parameterized evaluation algorithm (Algorithm 6) extends the general query evaluation algorithm in Chapter 4 to simulate the join with materialized views of ancestor generators. The algorithm takes as input a generator *gen*, a context tuple and a query type *type*. The query type identifies where the context tuple comes from and determines which statement in *gen* is to be evaluated. After incremental query creation, up to four queries may be present in each generator, which are:

   (a) `CurrentStateQuery`, the original query that computes the view at the current state $q(s_c)$.

   (b) `PastStateQuery`, the query rewritten from the original query by replacing all the base relations to their past state, to compute the view at the past state $q(s_p)$.

   (c) `PositiveIncQuery`, the incremental query computing the positive deltas of the view Δ^+q.

   (d) `NegativeIncQuery`, the incremental query computing the negative deltas of the view Δ^-q. Note that the negative incremental query may not exist if the original query is not safe.

   Note that we assume in the algorithm that the old view is always accessible.

   EXAMPLE 5.4.1. We use an example to illustrate how the algorithm works in practice.

```
function evaluate(gen, context, query_type)
begin
    old_view ← get the old view using gen.name and context
    switch query_type do
        case CurrentStateQuery
            delta_plus_view ← CurrentStateQuery (gen, context)
            append delta_plus_view to gen.delta_plus
        case PastStateQuery
            delta_minus_view ← PastStateQuery (gen, context)
            append delta_minus_view to gen.delta_minus
        case PositiveIncQuery
            delta_plus_view ← PositiveIncQuery (gen, context)
            append delta_plus_view to gen.delta_plus
        case NegativeIncQuery
            if gen has NegativeIncQuery statement then
                delta_minus_view ← NegativeIncQuery (gen, context)
                append delta_minus_view to gen.delta_minus
            else
                new_view ← CurrentStateQuery (gen, context)
                append old_view − new_view to gen.delta_plus
        new_view ← old_view − delta_minus_view + delta_plus_view
        evaluateChildren(new_view, gen,PositiveIncQuery)
        evaluateChildren(delta_plus_view, gen,CurrentStateQuery)
        evaluateChildren(old_view, gen,NegativeIncQuery)
        evaluateChildren(delta_minus_view, gen,PastStateQuery)
end
function evaluateChildren(collection, gen, query_type)
begin
    foreach tuple in collection do
        foreach child_gen of gen's children do
            evaluate(child_gen, tuple, query_type)
end
```

Algorithm 6: Parameterized evaluation of the incremental query

plan_id	title	organization
1	Flying Cars	UCSD

(a) Relation `plan` at s_p

staff_id	name	organization
1	John	UCSD
2	Jane	Stanford

(b) Relation `staff` at s_p

Figure 5.1: Example base relations

Figure 5.1 shows the base relations that the example query operates over: 5.1a represents the base relation `plan` at the past state s_p, and similarly 5.1b represents `staff` at the past state s_p. The example query q produces a nested data tree where each report tuple nests all staff members of the same organization, as follows:

```
1  SELECT  P.plan_id, P.title, P.organization
2          (
3              SELECT  S.staff_id, S.name
4              FROM    staff AS S
5              WHERE   S.organization = P.organization
6          ) AS members
7  FROM    plan AS P
```

At state s_p, query q outputs a view $V(s_p)$. Figures 5.2a and 5.2b illustrate the decomposed views of $V(s_p)$:

plan_id	title	organization
1	Flying Cars	UCSD

(a) Relation `report` at s_p

staff_id	plan_id	name
1	1	John

(b) Relation `report.members` at s_p

Figure 5.2: Example decomposed old view

Suppose during the past state s_p and the current state s_c, several modifications are made to the base relations, as the flat deltas Δ^+`plan` and Δ^+`staff` show in Figures 5.3a and 5.3b.

plan_id	title	organization
2	Invisible cloak	Stanford

(a) Relation `delta_plus_plan`

staff_id	name	organization
3	Mary	UCSD
4	Mario	Stanford

(b) Relation `delta_plus_staff`

Figure 5.3: Example flat deltas of base relations

In order for the query to incrementally maintain the view, each generator tree node i is attached with incremental queries $\Delta\overline{q_i}$ that are the result of rewriting the decomposed query q_i, as shown in Figure 5.4.

Generator for `report`

```
1  CurrentStateQuery {
2  SELECT  P.plan_id, P.title, P.organization
3  FROM    plan AS P
4  }
5
6  PositiveIncQuery {
7  SELECT  P.plan_id, P.title, P.organization
8  FROM    delta_plus_plan AS P
9  }
```

Generator for `report.members`

```
1  CurrentStateQuery {
2  SELECT  S.staff_id, S.name, :report/plan_id
3  FROM    staff AS S
4  WHERE   S.organization = :report/organization
5  }
6
7  PositiveIncQuery {
8  SELECT  S.staff_id, S.name, :report/plan_id
9  FROM    delta_plus_staff AS S
10 WHERE   S.organization = :report/organization
11 }
```

Figure 5.4: Example generator tree with incremental queries

plan_id	title	organization
2	Invisible cloak	Stanford

(a) Relation `delta_plus_report`

staff_id	plan_id	name
2	2	Jane
4	2	Mario
3	1	Mary

(b) Relation `delta_plus_members`

Figure 5.5: Example incremental view deltas

The evaluation algorithm starts at the generator `report`. First, it evaluates the positively incremental query statement and results in a collection containing the report tuple with title `Invisible cloak`. Then the algorithm uses this tuple as the context tuple to evaluate the `CurrentStateQuery` statement in the child generator `report.members`, and produces two member tuples with names `Mario` and `Jane`.

Notice that if the algorithm had incorrectly used the `PositiveIncQuery` statement for `report.members`, the member tuple with the name `Jane` will not have been produced. This is an example of where incremental queries cannot be used. Specifically, the original query that produces member tuples is not safe, as the join condition does not involve the primary key of `report`.

Thereafter, in order to produce joins with tuples in Δ^{+}`staff`, the algorithm iterates over each tuple in the old decomposed view of `report` as the context tuple. The member tuple with name `Mary` is constructed by evaluating the positively incremental query statement in `report.members`. The flat deltas after evaluation are illustrated in figures 5.5a and 5.5b.

2. Deparameterized evaluation

   The parameterized evaluation may not be efficient, because a query has to be separately evaluated for each context tuple with corresponding arguments for the parameters. A more efficient way is to remove the parameters from a query so that it is a standalone query that is evaluated only once.

   The deparameterization is applied to the full query with the following procedure:

(a) Suppose the full query q_k is at level k of the generator tree. Deparameterization adds all views $v_1, \ldots, v_{k-1}$ of its $k-1$ ancestor generators to the `FROM` clause of q_k as base relations.

(b) All parameters in q_k of the form `:generator/attribute` are replaced with `view.attribute`, i.e. a join to the view's attribute.

(c) If q_k has a `GROUP BY` clause and/or aggregate functions, all the primary keys of the ancestor views are added to the `GROUP BY` clause.

After the full query is deparameterized, the incremental queries are generated as per normal, with the rewrite rules treating the materialized views as regular base relations.

Algorithm 7 illustrates how incremental query evaluation is performed with deparameterized queries.

function `evaluate`(*gen, old_data_tree*) **begin**
    old_view ← get the decomposed old view from *old_data_tree* using *gen.name*
    Materialize *old_view* into database with the table name as *gen.name*
    delta_plus_view ← evaluate the deparameterized PositiveIncQuery statement in *gen*
    delta_minus_view ← evaluate the deparameterized NegativeIncQuery statement in *gen*
    new_view ← *old_view* − *delta_minus_view* + *delta_plus_view*
    Materialize *new_view* into database with the table name as *gen.name*
    foreach *child_gen of gen's children* **do**
        `evaluate`(*child_gen, old_data tree*)
end

Algorithm 7: Deparameterized evaluation of the incremental query

5.5 Reconstruction of the new view

After the incremental query evaluation, each generator i contains the new decomposed view $v_i(s_c)$. Using the reconstruction algorithm (Algorithms 4 and 5)

obtains $V(s_c)$ the view at current state.

Acknowledgements

This chapter contains material from the technical report “Technical Specifications of the FORWARD Framework”, by Yupeng Fu, Kian Win Ong, Kevin Keliang Zhao, Yannis Papakonstantinou and Michalis Petropoulos. The dissertation author was the primary investigator and author of the relevant chapters in the technical report.

Chapter 6

Do-It-Yourself Platform

6.1 Introduction

app2you belongs to the emerging space of *Do-It-Yourself (DIY), custom, hosted, database-driven web application platforms* that empower non-programmer business process owners to rapidly create and evolve applications customized to their organizations' data and process needs. The hoped-for outcome of DIY platforms is paralleled to the emergence of spreadsheets in the 80s and of graphical presentation tools in the 90s [Bor07]. Before the arrival of tools such as PowerPoint, polished presentations had to be prepared by graphics professionals. PowerPoint enabled us to do them ourselves.

Generally DIY application platforms provide an *application design facility* (also called *application specification mechanism*) where the *application owner* (also called *process owner*) specifies the application by manipulating visible aspects of it or by setting configuration options. A simple early example of DIY creation was form builders, where the owner introduces form elements in a form page and the platform, in response, creates a corresponding database schema.

A DIY platform must maximize the following two metrics:

- **Application scope**, which is characterized by the computation, collaboration on a process and pages (presentation) that can be achieved by applications specified using the platform.

- **Ease of specification** of applications. When ease increases, a larger number of less technically sophisticated creators are empowered to create applications.

The two metrics present an inherent tradeoff. At the one extreme, building applications using Java, Ajax and SQL provides unlimited scope, but does not provide ease of specification. Platforms such as Ruby on Rails [Rub] and WebML [CFB00] make specification easier and faster, but still not easy enough to enable non-programmer owners. At the other extreme, creating an application by copying an application template, as done in Ning [Nin] or Salesforce [Sal], is easy but the scope is limited[1] to their finite number of templates. DIY platforms are between these two extremes of the scope/ease trade-off (see Section 6.4 for a discussion of particular platforms).

To systematically explore the scope versus ease tradeoff between these two extremes, we characterize FORWARD and app2you as follows:

- the *FORWARD scope* captures the full generality of human-centric database-driven workflow applications, but achieving its full scope requires knowledge of SQL. As seen in Chapter 2, a programmer uses FORWARD's declarative syntax to specify an application by providing (1) extended SQL queries to retrieve the data displayed on pages, and (2) imperative server-side programs that modify the database state using SQL `INSERT / UPDATE / DELETE` commands.

- Of the general FORWARD scope, app2you's DIY design facility captures a strict subset called the *app2you scope*, which is the focus of this chapter. The app2you design facility generates FORWARD applications, but also imposes limitations on the FORWARD features it uses. Its essential limitation is that the core queries are SPJ queries, including the possibility of `EXISTS` conditions in the `WHERE` clause. An overlay of aggregation and calculation,

[1] WYSIWYG tools occasionally provide programmatic escapes for sophisticated users to write custom code. For example, Microsoft Excel can be further programmed using the embedded VBA scripting language. The scope and ease of two such inter-operable tools should be considered separately, as will be the case for FORWARD and app2you.

which captures Excel functionality, is applied on the core. These limitations are common in real-world applications and have marginal negative repercussions on application scope practically, but enable automatic inference of schemas, queries and programs by inspection of the structure of the application pages. The declarative nature of FORWARD is also critical in enabling these inferences.

The limited app2you scope presents an excellent scope / ease-of-specification tradeoff point: app2you's DIY design facility enables the easy specification of applications by "business architects" [TV07], that is, application owners that are not programmers but have the sophistication to reduce their business process into web pages by specifying, in a WYSIWYG fashion and in response to easy-to-understand prompts, properties of the pages such as who can access each page, what is the page's main function, what happens in response to an action.

Formulating the FORWARD scope as an extension of the app2you scope also enables a principled exploration of design approaches for app2you. In optimizing for ease of specification, many DIY platforms suffer from arbitrary, and sometimes unnecessary, limitations on application scope. Not only do the arbitrary limitations restrict customization of key functionalities of the resulting applications, they also disable in-depth comparisons between the various approaches and impedes the development of platforms that can seamlessly combine the best aspects of each approach.

Last but not least, building a DIY platform on top of a rapid development framework introduces novel opportunities for efficient collaboration between business process owners and IT specialists. Where an entire web application cannot be fully created or customized in app2you, the non-programmer owner can still create by himself the bulk of the application's pages and workflow, while the programmers provide assistance in elaborate graphics, integration with outside services, code for complex functions, complex SQL and other such aspects in FORWARD. This assistance can occur at different stages of the software development cycle. Programmers can choose to participate at the beginning by providing generic application templates that can be readily customized, at the end where app2you

would have served as a rapid prototyping tool to clarify the requirements, or indeed continuously throughout the cycle as point interventions on abstract and complex aspects of the application.

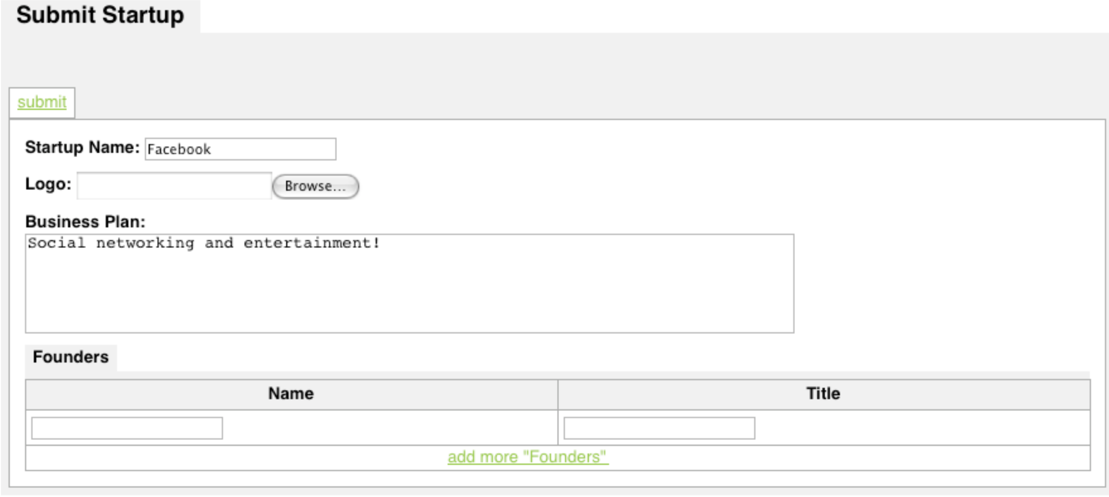


Figure 6.1: `Submit Startup` Page

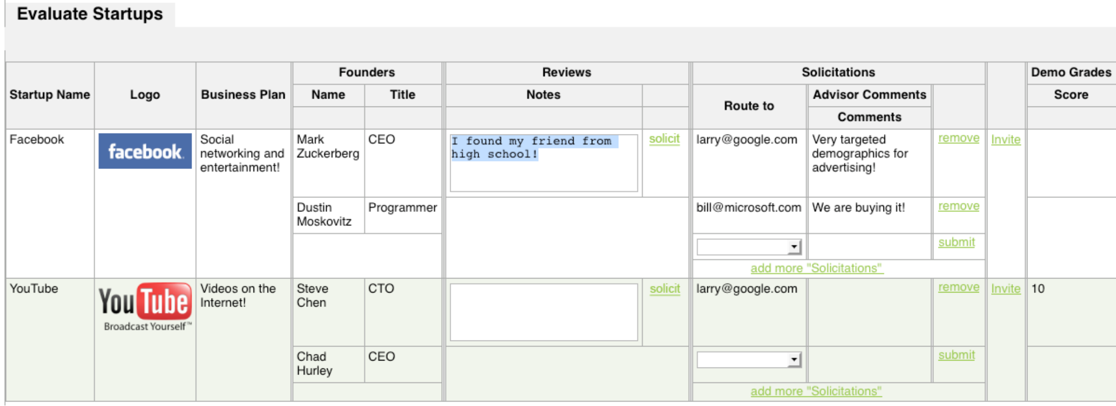


Figure 6.2: `Evaluate Startups` Page

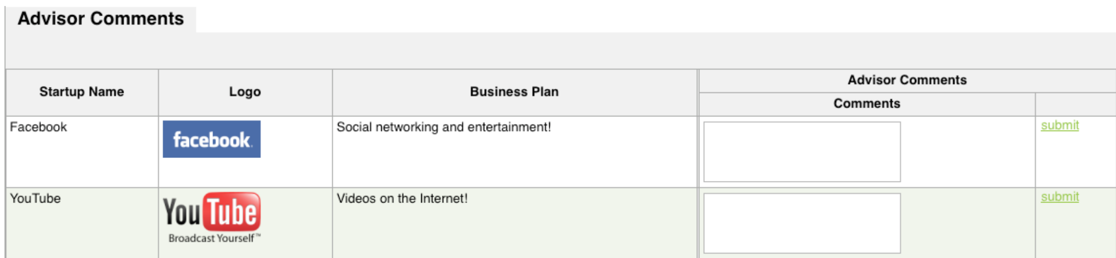


Figure 6.3: `Advisor Comments` Page

In an app2you application users with potentially different roles and rights interact on a web-based process. Depending on the state of the process/application, each user has rights to access certain pages, read certain records in them and execute actions, which often pertain to reported records.

Let us first convey informally the scope of app2you applications through a redacted version of a real-world app2you application. More real world examples are found in the Appendix. We use a simplified and modified version of the app2you application for TechCrunch50 (TC50) 2008 [Tec], a conference where over 1000 startups submitted requests, along with information packages, in order to present themselves and their products. The app2you application was used to collect the submissions, review them, schedule multiple rounds of appointments where the candidates met the reviewers online to demo their products, and select the top 50 startups. At page `Submit Startup` (Figure 6.1) any user with a registered account can prepare and submit information regarding her startup, which includes the name, logo, and list of founders.[2] Every user is constrained to at most one submission. The submitted startups are displayed on the `Evaluate Startups` page (Figure 6.2), which is accessible by all reviewers, each of whom can execute three *actions* on each one of them: submit a review consisting of `Notes` for each startup; solicit comments from one or more advisors (essentially external reviewers), in which case the startup submission will be displayed on the `Advisor Comments` page (Figure 6.3) to the particular advisors;[3] invite the startup submitter to schedule an interview. The invitation results to the candidate receiving an email notifying

[2] Users must first pass from a typical login and signup page before they reach the page of Figure 6.1.

[3] In the actual application there were solicitations to other reviewers.

him to visit the `Schedule Appointment` page, which reports available interview slots, submitted by the reviewers on the `Post Interview Slots` page, and lets the invited startups choose one of them (see Figure 6.4). The submitted choices are reported on the `Grade Demo` page, where the reviewers post their grade for the demo given at the agreed interview slot as part of the second review round. Finally, the submitted demo reviews are reported on the `Evaluate Startups` page, where reviewers can now make an informed decision of which 50 startups are the most promising ones. The actual application evolved during a period of two months, indicating the great value of the Do-It-Yourself approach in allowing applications to evolve as the business process evolves.

The general goals of app2you's Do-It-Yourself design facility are typical in easy-to-use systems: (i) WYSIWYG design, where the owner immediately experiences the result of each specification action. (ii) Wizards that suggest to the owner common and semantically meaningful specification options and automate their implementation. (iii) Wizards that explain the specification at a high level where the user does not have to engage in schema design or database queries. Satisfying such ease-of-use specification goals has required the introduction of multiple novel DIY specification techniques, which are described in this chapter, and the construction of algorithms that automatically infer the schemas and queries that fuel the pages. The reported techniques have been designed during a period of one year, as we have been observing stumbling blocks faced by owners in their efforts to build applications.

Due to the highly interactive, WYSIWYG nature of the DIY design facility we suggest that the reader watches the 10-minute high resolution video at `http://www.vimeo.com/2075363`*, password* `app2you`*, which is also a key part of the chapter's presentation.*

The first technique is *page-driven design* (Section 6.3.2), which provides to the owner a WYSIWYG model of the pages. The owner specifies properties of the pages that have immediately visible effects on the page. For this contribution we borrowed techniques from the WYSIWYG/automatic design of database schemas by the creation of respective input forms as done in form-builders even before the

web. We expanded with the WYSIWYG specification of forms and actions that operate within the *context of reported records* (for example, every input form, such as `solicit` action and `invite` action on `Evaluate Startups`, operates within the context of a submitted startup). We also launched wizards for specifying properties by answering questions, expressed in easy-to-understand language referring to the pages and the actions that happen on them. app2you in response creates automatically the pages' structure (called *page sketches*, Section 6.2) and the underlying schemas and queries, therefore relieving the business process owner from designing the database layer, which is one level away from the layers that she understands, namely, the application page layer and the overall workflow. Hiding database schemas, queries, constraints and other low-level details is facilitated by an architecture where high level, easy-to-explain derived properties of the page sketch hide hard-to-understand complex primitive properties (Section 6.3.1).

We found out that the inherent difficulty (in comparison to, say, a spreadsheet) in producing a WYSIWYG model for a collaborative application is that the pages typically behave differently depending on which user accesses them and the application state. The resulting enhancement to page-driven design (Section 6.3.2) is prompting the owner to experience the page's function as if she were a suggested sample user of it. Such prompts are issued when assuming the role of such sample user and performing particular suggested actions reveals properties of the pages' operation that would otherwise not exhibit themselves. Forcing the use of the application also leads to collecting sample data, which become useful in exhibiting the operation of other pages also.

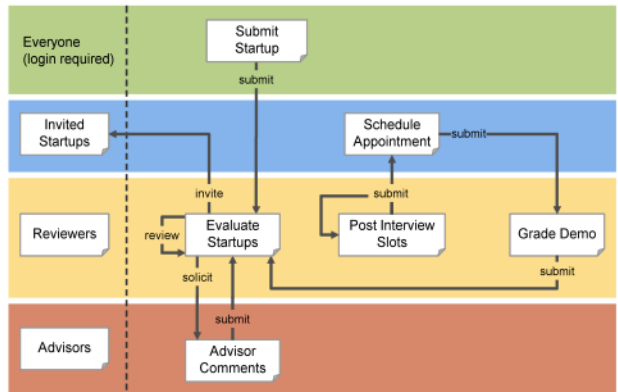


Figure 6.4: TechCrunch50 Workflow Visualization
(in ppt; see video demo for the actual one)

Page-driven design by itself still turns out to be insufficient for allowing the owner to reduce a non-trivial multistep process she has in mind into a working application. In order to appreciate the difficulty that non-programmer owners face, visualize a database-driven application as a workflow. Figure 6.4 shows the workflow visualization of the functionality of the TC50 application. *User groups* are on the left. The rows in Figure 6.4 visualize *access rights*, that is, which pages are accessible by which user group. Intuitively, it is easy for the owner to specify in a single specification action what appears as a single transition in the workflow graph, such as the `solicit` edge. Unfortunately, a major shortcoming of DIY online databases, which is not resolved by page-driven design alone, is that they require the owner to decompose a single user action in the process into coordinated activity in two pages. For example, in page `Evaluate Startups` the user submits the solicited advisor's name. Page `Advisor Comments` has a too-complex-for-non-programmers query that filters startup submissions according to whether the currently logged-in user appears in a solicitation related to this startup. The technique that will resolve this problem is the *workflow-driven design* extension (Section 6.3.3) to the page-driven paradigm.

Finally, in Section 6.3.4, we discuss work on semiautomatic creation of reports. The goal is a report building interface that

- suggests semantically meaningful joins of various data sets; or joins of the

currently reported data with other collected data sets of the application

- does not suggest joins that would lead to provably redundant information on the report
- explains to the owner the (potentially nested) involved data sets and joins by referring to names that appear in the application; avoids causing confusion with details of how the pages are normalized in tables
- requests minimal information in the form of plain multiple choice menus
- discovers the best placement of information on the report in order to illustrate associations and constraints between the reported data sets.

In effect the interface must compensate for the minimality of the owner-provided information with algorithms that detect and perform complex nested report creation operations.

Roadmap Section 6.2 presents the app2you scope as a subset of the FORWARD scope, and argues why the app2you scope can still capture many practical applications. Section 6.3 briefly describes an array of design techniques. Section 6.4 discusses related work.

6.2 Framework and Scope

An app2you application is described by an *application sketch*, which is modified by the owner when the application is in design mode. The sketch consists of *primitive properties* (collectively called *primitive sketch*) and *derived properties*, where the former are low level data structures (e.g., queries, constraints) as defined in the FORWARD scope (Chapter 2), and their settings cannot be derived from the settings of other properties. For ease of specification the non-programmer owner typically does not access the primitive sketch aspects directly, since deconstructing a process into primitive aspects tends to require CS sophistication. Rather the non-programmer owner indirectly accesses them via the *derived properties*, which

explain at a high level common questions and options, using wizards and other components of the DIY design facility (Section 6.3).

6.2.1 Reports

In the FORWARD framework, the unit tree is homomorphic to the page schema tree, which is in turn homomorphic to the page query's abstract syntax tree. A *field* is a triplet comprising (1) a unit, (2) its corresponding type node in the schema tree, and (3) the corresponding fragment in the original page query that outputs the attribute. The query fragment can be represented as a standalone query q, which is parameterized by the context c and retrieves from the database tuples $t_1, \ldots, t_n$ that have schema t, and correspond to the values rendered by the field's unit. In particular, a field whose unit has child units is an *iterator*, otherwise it is an *atomic field*. Consequently, a hierarchical structure is induced between iterators and fields, and an iterator field recursively contain its own *nested iterators*. A child unit operates within context $c + t$, which is the concatenation of c (i.e., the context within which the iterator operates) and t (i.e., the context that the iterator generates).

For example, the `Evaluate Startups` page (Figure 6.2) has a top-level iterator, which contains the atomic fields `Startup Name`, `Logo`, and `Business Plan`. This iterator runs a query `SELECT * FROM Submit_Startup`, where `Submit_Startup` is the automatically inferred table that collects the non-nested fields of the startup submission form (see Figure 6.1). It also contains the (nested) iterator `Founders`, which, in turn, contains the atomic fields `Name` and `Title`. The query of the iterator `Founders` is `SELECT * FROM Founders WHERE Founders.Parent=?` and the parameter `(?)` is instantiated by the `Submit_Startup.ID` of the query result of the containing iterator.

The FORWARD framework allows iterator queries to be arbitrary SQL queries over the schema, typically parameterized by values of the context. In this way SQL experts can utilize SQL's full power. The app2you scope queries (*lqueries*) are SPJ queries with `EXISTS` predicates, that is, queries of the form `SELECT * FROM` ***`OuterJoinExpression`*** `WHERE` ***`BooleanCondition`***, where the condition

may be parameterized with values from the context and may also involve `EXISTS(`*`SubQuery`*`)` predicates where the parameterized *`SubQuery`* is recursively an lquery. Applications of the app2you scope use only lqueries and corresponding constraints, which are generated by the DIY facility. A DIY-built application however may go outside the app2you scope and into the FORWARD scope by selectively utilizing "manually" written queries and constraints for a few complex functionalities.

App2you also allows *calculated fields* to be associated with queries. In the app2you scope such queries will capture the typical functionality of Excel spreadsheets. In particular, a calculated field may:

1. Compute a new scalar value from values of the context. For example, if the context has attributes `First Name` and `Last Name` then the calculated field `Name` may be calculated as `concat(Last Name, ",", First Name)`.

2. Compute an aggregate value by applying an aggregate function over a (potentially hidden) nested iterator of the page. For example, the non-programmer owner may include in `Evaluate Startups` a calculated field `Number of Founders` that performs the count function over the `Founders` iterator fields.

3. Combinations of the two above.

Lqueries, scalar calculations and aggregates capture the needs of most typical reporting applications and even the needs of relatively unusual action-controlling constraints, such as "each startup may receive at most 5 advisor reviews". Therefore the limitation leads to small scope loss. At the same time, this limitation enables ease of specification benefits: First, the design facility automates the creation of reports (see Section 6.3.4) for lqueries. Second, filtering and aggregation uses DIY interfaces that have proven themselves in other settings (e.g, spreadsheets). Third and most important, lqueries enable the easy and efficient computation of the context created by each report tuple, therefore enabling the automatic inference of the database commands associated with contextual actions, such as the `submit`, `invite` and `solicit`, i.e., actions that appear in the context of reported data, as explained in Section 6.2.2.

Note that for DIY simplicity the design facility focuses on pages with a single iterator at the top level of the page. Such pages are called report pages.

6.2.2 Contextual Actions

Recall in the FORWARD framework that a unit can be associated with one or more server-side programs. An *action* is a triplet comprising (1) a program (2) the unit from which the program can be invoked (3) a *constraint*, represented by a yes/no query (see discussion below on representation of queries), whose semantics is that the action is applicable only when the constraint is satisfied. Invoking an action produces side-effects (or *effects*).

The most common effect of executing an action is an update on the database; this will be the only effect discussed in detail next. In the FORWARD framework such effect is captured by an SQL statement, which is possibly parameterized by the context. In the app2you scope the database effect is automatically inferred by the DIY design facility: It is an insertion in the database of the values collected by the input fields. It is described in the sketch by (i) naming the database table that takes the insertion and (ii) mapping the input fields to type-compatible attributes of the table. If the form contains repeated nested forms, such as the `Founders` in the `Submit Startup` form that contains `Name` and `Title` pairs, then each nested form is mapped to a corresponding database table. Note that the inserted record also includes *system attributes* such as the auto-generated `ID`, the `submitter` and `creation timestamp` of the record.

Other effects of an action may be (i) sending an email, described by a template (in the style of MS Word mail merge) whose placeholders can refer to both the input fields of the form and the system attributes and (ii) causing a navigation to another page, which can be used to produce confirmation pages and forms submission processes that span multiple pages.

For example, the data submission form of Figure 6.1 has a `submit` action. Its effect is inserting the collected data in tables `Submit_Startup` and `Founders` and sending a confirmation email. It has the constraint that the currently logged-in user has not submitted a startup already. The `solicit` and `invite` of Figure

6.2 are the buttons of respective actions.

A feature that sets the scope of app2you applications apart from the scope of online databases (see Section 6.4) is the ability of reports to have nested actions, which operate in the context of the reports. For example, the `solicit` action in Figure 6.2 operates within the context created by the containing report iterator. Such a nested action is said to be an *annotation* of its report iterator. A nested action differs from a top level action as follows: First, when it inserts in the database it may map values from its context into attributes of the insertion table. For example, when the `solicit` action is executed it stores a tuple in a table `Solicitation` and this tuple has a foreign key attribute that stores the `ID` of the startup submission within whose context the particular nested action operates. Second, its constraint and its side effects may also utilize the context. For example, the `invite` action is associated with a constraint that there may be at most one invitation for each startup.

Note the following important interplay between lqueries and the automatic inference of the insertions happening when an action is issued: lqueries enable automatic inference of a compact context for the nested actions that appear within reports fueled by such lqueries. In particular, each record produced by a lquery creates a context consisting of the `ID`s of the few database tuples that joined together to result in it. This, in turn, enables fully automatic inference of an efficient database insertion performed when the nested action is activated. In particular, the insertion stores the IDs of the compact context along with the input fields of the action and the system attributes. This, in turn, leads to ease of specification since the non-programmer owner does not have to specify what part of the context of a nested action will be stored with the insertion.

Note that the DIY design facility is facilitated by *iterator+action combos* where the iterator part of the combo ranges over actions created in response to the action part of the combo. For example, the iterator+action `Advisor Comments` in Figure 6.3 combines the `submit` action with an iterator showing the comments collected by the `submit` action.

6.2.3 User Group Definitions

In the app2you scope *user groups* (such as `Invited Applicants`, `Advisors` and `Reviewers`) are identified as a pair consisting of a report page and a field (of such report) whose values are user identities. The `submitter` is typically such a field.

6.3 Do-It-Yourself Design Facility

app2you's design facility allows its non-programmer business process owner to easily and rapidly create a working web application that can be immediately tested and experienced. If the result is not what the owner had in mind a new round of specification-testing can be played out within seconds.

We focus on three key DIY-enabling techniques of the design facility and the architecture that enables them: page-driven design (Section 6.3.2), workflow-driven design (in progress, Section 6.3.3) and automatic creation of complex reports (in progress, Section 6.3.4). We use the following principles as a scorecard for the DIY design facility.

- Prefer to provide concrete explanations of sketch properties using WYSIWYG feedback and verbalization of prompts and options that refers to pages, actions and other highly visible properties of the page; rather than being abstract and making references to database terms.

- Prefer to provide a high-level specification from which primitive properties can be generated, rather than a low-level specification of primitive properties that requires the owner to deconstruct high level concepts into low level concepts.

- Prefer to summarize and enumerate design options to focus on common cases, rather than provide an unstructured, high degree of freedom. "Advanced user", less prominent interfaces should cater to the less common cases.

6.3.1 Derived Properties

Often an important combination of primitive properties must be explained to a non-programmer owner at a high level, which is close to the non-programmer's understanding of the workflow and the function of the pages. Therefore the *derived properties interface* reads the primitive sketch and exports *derived properties* and corresponding common options (called *derived* options) for their settings. When the owner chooses an option the derived properties interface translates it back to the primitive sketch. We describe next a simple example of a derived property, exemplifying the concept. Derived properties become paramount in the following sections.

For example, recall that a user of the `Submit Startup` page may submit only one startup. Once she makes her submission, the form of Figure 6.1 disappears. At the primitive sketch level, this behavior is achieved by a non-obvious primitive property: The constraint associated with the form checks that the set of startup submissions of the currently logged-in user is empty. Understanding the behavior of the `Submit` form at this level is fairly complex. Therefore the page wizard offers a derived property asking the much more obvious question of Figure 6.5.

The combination of a primitive sketch with a derived properties interface produces many benefits on scope and ease of specification:

- It enables the incremental addition of derived properties in the platform, as common cases that lend themselves to higher level explanations emerge, without disrupting existing applications. Indeed, applications created before the introduction of a new derived aspect in the platform can benefit from its introduction: The derived properties interface reads their primitive sketch and exposes a high level derived property.

- It enables a 90/10 rule where the design facility first poses common questions, often relying on derived properties and derived options in order to express them. At the same time, the wide scope enabled by the primitive sketch is available.

6.3.2 Page-Driven Design

The first step towards providing a high-level specification is to allow the process owner to design her application through the WYSIWYG model of pages, as opposed to engaging in low-level web and database programming. Various properties of pages are either specified by direct visualization on the pages, or via answering simple questions about the page. The design facility in response automatically creates the page's form/action and iterator structure, underlying schemas and queries.

Through the high-level specification, page-driven design relieves the owner from specifying data structures in the abstract while en route to construct pages. Moreover, explaining the design options available at the page level promotes easy comprehension, especially if they are explained directly in terms of the application layer that are easily perceived by the owner such as what is the report/form structure of the pages. Lastly, page-driven design facilitates immediate feedback on whether a design satisfies the owner's requirements, since the owner can both inspect and experience the page directly.

6.3.2.1 Page Wizard

(a) Access Rights

(b) Page Templates

(c) Record Names

Figure 6.5: Page Wizard for `Submit Startup` Page

The page wizard is the starting point of page-driven design. It prompts with simple questions about page-specific information, such as the page name,

URL, and the groups that are authorized to access the page. For example, access to `Submit Startup` is granted to system-defined group `Everyone (no login required)` (Figure 6.5a), whereas access to `Evaluate Startups` is granted to custom group `Reviewers`. Allowing a page to be accessed by a group is also visualized on the workflow diagram by placing the page in the appropriate swim-lane (row).

The page wizard prompts for the main function of the page by enumerating a list of templates, where each template bundles a commonly occurring combination of page properties including presentation format and action rights. Templates are provided to speed up the design of common cases. Such common cases may include forms that allow each user to submit at most one record, and tabular reports where each user sees all records but can only edit/remove the records she submitted, etc. Where the common case is not fully applicable to the scenario at hand, the owner can always customize the page by overriding individual properties independently.

Figure 6.5b shows the `Private Form` template used in creating the `Submit Startup` page. The template provides the following defaults for the following derived properties:

- The `submit` property of the page's form is set to `on, but max one per user`. (Each applicant can only submit one startup.)
- The `display` property of the page's iterator is set to `on if user has submitted the record, off otherwise`. (Each applicant can only see the startup info she has submitted.)
- The `edit` and `remove` properties of the page's iterator are also set to `on if user has submitted the record, off otherwise`. (Each applicant can only edit or remove the startup info she has submitted)

Whenever the `submit` aspect is `on`, the page wizard also prompts the owner to optionally assign a name to the records collected. The record name helps the system phrase questions and options more specifically. Figure 6.5c shows the wizard for `Submit Startup`. It starts with a system-proposed default of `Record submitted at Submit Startup`, which is later set by the owner to `Startup`.

6.3.2.2 WYSIWYG Design

After the basic properties of the page have been specified through the page wizard, the owner can customize the form of the page in a WYSIWYG fashion. To create new input fields, the owner drags-and-drop input components such as text boxes, image upload prompts, dropdown boxes, check boxes etc. into the action form of the page (Figure 6.6).

For each input component dragged into the form, a corresponding field is added to the action, and a corresponding attribute is added to the schema of the database table where the records corresponding to the action are inserted. The input component determines the data type of the field. For example, the `Logo` field is created through an image upload component, therefore storage is allocated for binary data, data can be submitted through an HTML file input form element, and submitted data are displayed as images.

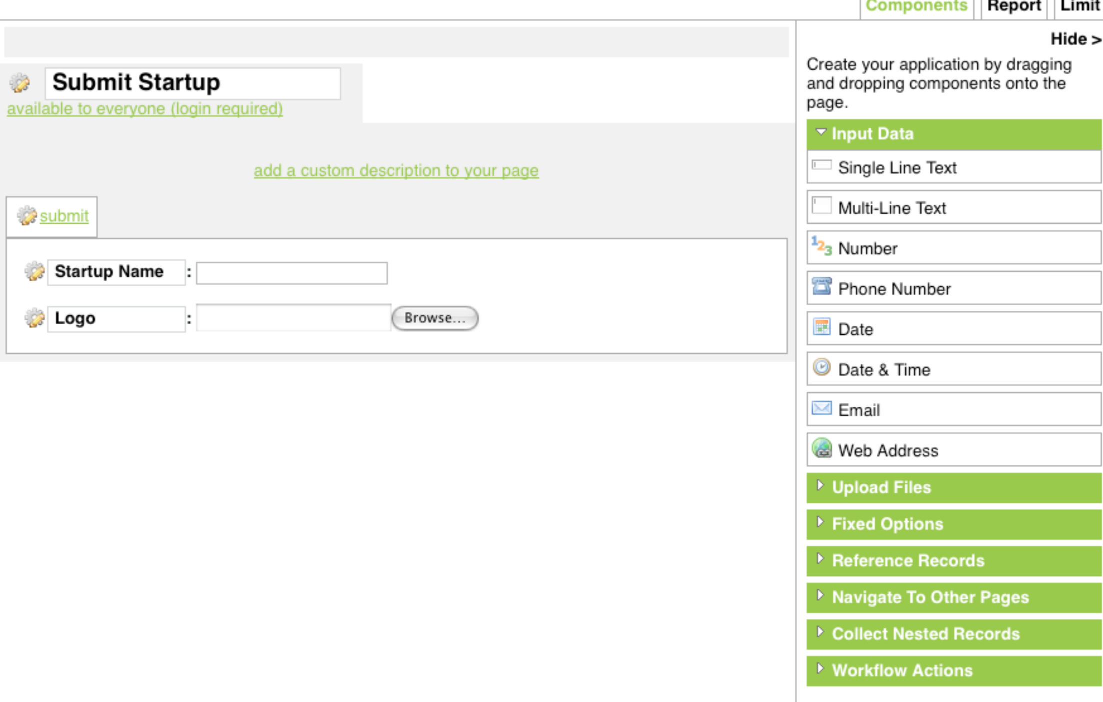


Figure 6.6: WYSIWYG Page Design

The owner may also introduce repeating nested data by creating nested tables, such as the `Founders` on the `Submit Startup` page.

6.3.2.3 DIY creation of nested actions

While existing form builders and online databases employ WYSIWYG design for (pure) input forms, app2you advances page-driven design to also encompass the WYSIWYG specification of nested actions that operate within the context of reported records. On the `Evaluate Startups` page for example, the `Reviews`, the `solicit` and `invite` nested actions all operate within the context of a startup. A nested action is also called *annotation* by the DIY design facility if it is a `submit` actions, which is associated with input forms, since the data it collects intuitively annotate the data of the report.

An annotation is created by the drag-and-drop of an input component into a report.[4] For example, the `Reviews` annotation is created when a multi-line textbox for `Notes` is dropped into an area corresponding to a startup, excluding the area corresponding to `Founders`. In this way the owner visually specifies the context of the `Reviews` annotation to be a startup. Had she accidentally dropped it into the area for `Founders`, she would have seen a multi-line text box for each founder, which creates an immediate visual indication of the mistake. Recall that the design facility automatically infers the database insertions that will be issued when a review is submitted. For example, the insertion of a review will lead to inserting in the underlying table `Reviews` a record that contains the values collected by the input fields, system attributes and a foreign key that refers to the startup that provides the context for the particular review. The updates issued when a review is edited are computed similarly.

6.3.2.4 Experiencing the page

WYSIWYG design is not sufficient since there are properties that are not immediately evident from the page's visual appearance. For example, how many submissions can a user make? Can a user see which other users have submitted?

The inherent difficulty faced by an owner of a collaborative application

[4] As we will see in Section 6.3.3.1, more generally, nested actions are created by the introduction of a workflow action (such as the `solicit`, `invite`) into a report.

(as compared to an owner of a spreadsheet) in comprehending the behavior of an application and verifying it against her requirements, is that pages typically behave differently depending on what data has been submitted and who accesses the data. The design facility takes a number of steps towards resolving this problem. First, it makes every feature that is available in use mode also available during design mode. The fact that the page sketches are interpreted, instead of requiring a design-compile cycle, facilitates this. Second, it always prompts the owner to submit sample data and make actions so that corresponding records can be shown on report pages. The third step is to prompt and help the owner assume the role of particular sample users in order to visualize the behavior of properties that would otherwise be hidden.

The system suggests to the owner to experience a page as a sample user if it recognizes that certain properties of the page cannot be explained by the owner's current WYSIWYG experience. For example in `Submit Startup`, the system suggests the experience `submit as a sample user` in order to explain to the owner the following properties:

- The `display` property of the page is set to `on`.[5] The owner understands this when she sees that the startup info record submitted by the sample user is displayed on the page.

- The `submit` property of the page's action form is set `on, but max one per user`. The owner understands this when she sees that the action form and button disappears once she submits a startup info record.

- The `edit` and `remove` properties of the page's iterator are set to `on`.[6]

Note however, that the experience of the first sample user does not fully explain whether the `display`, `edit` and `remove` properties are unconditionally or conditionally on. For example, does the iterator display all records submitted, or only

[5] The display aspect of a page is a derived aspect that asks whether a page that has a form also has a report iterators that displays the data submitted at the form.

[6] The edit and remove aspects of a page are derived aspects that ask whether the report iterator of the page provides the built-in actions edit and remove.

records submitted by the current user? Therefore, the design facility subsequently engages the owner to experience as a second sample user. The experience shows that in this the page, each user can only see, edit and remove records she has submitted. If this is contrary to requirements, the owner can then either select another template, or customize the individual properties defaulted by the template.

When the records displayed by iterators and the actions that are available are controlled by complex conditions, it is harder to reason about what sample data and sample users are needed in order to experience a page. For example, obtaining the experience of a solicited advisor at the `Advisor Comments` page requires that (i) at least one (sample) solicitation has been made and (ii) the owner uses the `Advisor Comments` page as if she were the solicited advisor. When the conditions have been introduced in response to workflow-driven design, as described next, it is easier to reason about such sample users and data.

Note that in practice sample data are not needed when the first pages of the application have actually gone in use and have already obtained actual data.

6.3.3 Workflow-Driven Design

In the workflow visualization of an application (see Figure 6.4), edges (also called transitions) capture actions that happen on the page at the source of the edge and affect the experience and rights of a user on the page at the target of the edge. The starting points of a workflow are data collection pages, such as `Submit Startup` and `Post Appointment Slots` that provide actions collecting new records without implicit or explicit references to other records. The records may be reported on the data collection page itself, or appear on reports that combine data collected from one or more pages. Reports, such as `Evaluate Startups`, may allow their user to act on individual reported records (`review`, `solicit` or `invite`). Formally, there is an edge from page P_1 to page P_2 labeled with action a_1 if executing a_1 on P_1 may change

1. the *read rights* of a user u on P_2, that is, u can read on P_2 a record r as a result of a_1. For example, the `submit` edge from page `Submit Startup`, accessible to `Everyone (login required)`, to page `Evaluate Startups`, accessible to

reviewers, denotes that reviewers gain read rights to a startup once the action is submitted.

2. the *action rights* of a user u on P_2, that is, u can perform an action a_2 on P_2 as a result of a_1. For example, the `solicit` edge indicates that upon executing the `solicit` on `Evaluate Startups` a user (in this case the solicited advisor) can read and comment on a startup submission at the `Advisor Comments` page.

3. the *access rights* of a user u on P_2, that is, u gains access on page P_2. For example, the `invite` edge of Figure 6.4 indicates that upon executing the `invite` action on `Evaluate Startups` a user (in this case the startup submitter) gains access to the `Schedule Appointment` page.

An implementation that visualizes the workflow also allows drilling down into the nature of the edges so that the owner can tell which type of right is affected by the edge, why it is affected, etc.

Some workflow transitions correspond to application functionality that is easily built using page-driven design. For example, the `submit` edge from `Submit Startup` happens because the owner ordered at the page wizard that the `Evaluate Startups` reports the data collected on `Submit Startup`.

However, process owners often want to capture more elaborate workflow logic, which leads to application functionality that cannot be easily-built in page-driven design. Consider in Figure 6.4 the `solicit` edge from page `Evaluate Startups` to page `Advisor Comments`, accessible toadvisors. Here, the reviewers may solicit reviews for each startup from a subset of the advisors. Using page-driven design, the owner has to add an annotation (action) to `Evaluate Startups` so that reviewers can choose the advisors to solicit reviews from. Then she needs to create the `Advisor Comments` page, for the advisors to submit their comments, by initially report all the startupsfrom the `Evaluate Startups` page, and then keep only those where the currently logged-in user is one of the advisors chosen to solicit a review from; not a simple condition to state regardless of how friendly the query building GUI is. Indeed, the query in SQL is:

```
SELECT *
FROM Submit_Startup
WHERE EXISTS (
    SELECT *
    FROM Solicitations
    WHERE SS_ref = Submit_Startup.ID
    AND route_to=<current user>
)
```

The `Solicitations` table folds the advisors chosen (`route_to` column) by the reviewers for each `Startup` (`SS_ref` column). The `SS_ref` column is a foreign key referring to the `ID` of a `Startup`. The `route_to` column is a foreign key referring to the `ID` of an advisor and the condition makes sure that the solicited advisor is the currently logged-in user. No matter how user friendly the query building of the design facility becomes, the above query is too hard to be conceived by a non-programmer.

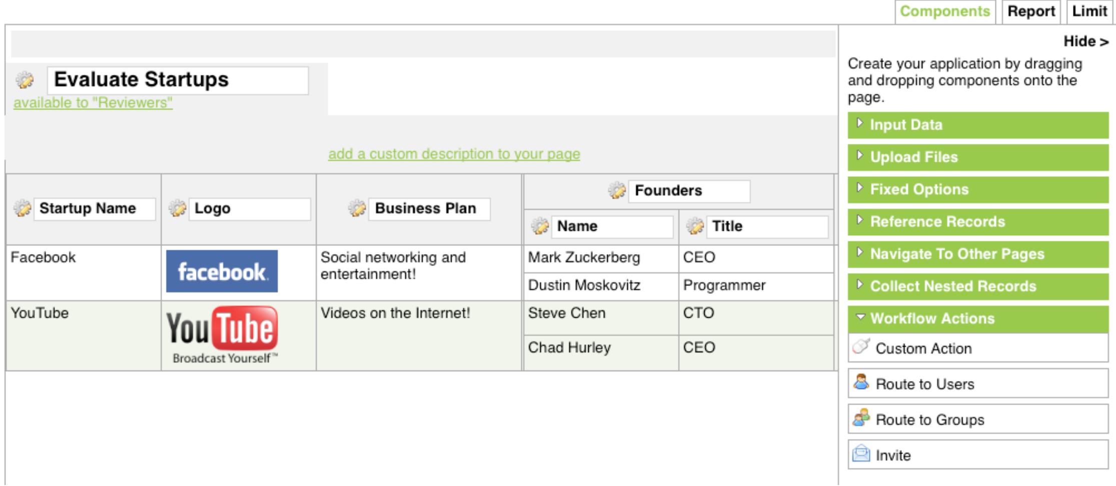


Figure 6.7: Workflow-Driven Design

Deconstructing a single workflow transition, which corresponds to a single user action, into the above design steps is not a trivial task for the process owner. For that reason, we enhance the design facility's page-driven design with *workflow-driven design* where all of the above design steps are integrated into a single DIY task performed on the starting page of a workflow transition.

Workflow-driven design is initially experienced by the owner as a set of `Workflow Actions` components, shown on the right side of Figure 6.7, which can be dragged-and-dropped on a page as any other component. For the `solicit` example, the process owner decides to drop the `Route to Users` component on the `Evaluate Startups` page, which triggers the `Create New Action` wizard. The wizard saves the owner from having to formulate queries like the one above.

6.3.3.1 Workflow Wizard

The requirement for the workflow wizard is to either automatically infer or ask the process owner about one or more of the following properties of the workflow action and correspondingly of the transition that appears in the workflow visualization:

1. The action on the current page that corresponds to the workflow transition.
2. The type of record involved in the workflow action, which can typically be automatically inferred from the context in which the action was introduced
3. The user group that will be affected by the action.
4. How exactly the action will affect the rights of the user group on the corresponding records. That is, will the action change the access rights, the readrights and/or the action rights of the affected user group on the target page. In most cases this is implied by the choice of the action and no additional information is needed.
5. Depending on the answer to the above question, additional questions about the exact implementation of the rights become relevant. For example, if the workflow action makes the record readable by users of the affected group, which is the page where the users will read the record?
6. How the affected user group will be notified of the action?

As an example, let us consider what the workflow wizard for the `Route to Users` action should do, in the spirit of the above properties, while the owner customizes the action to solicit comments from advisors.

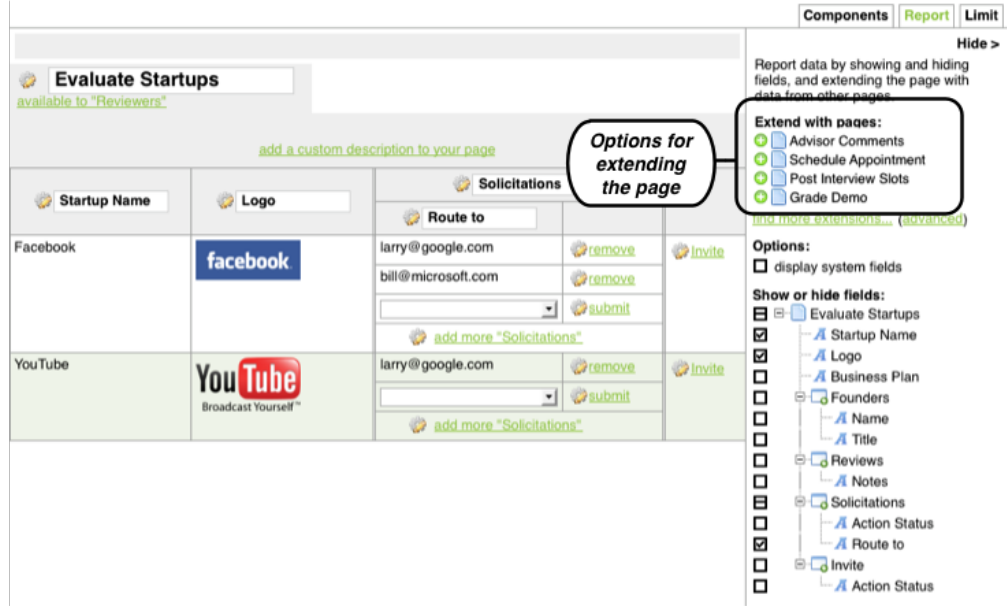
Property 1 is answered purely by the fact that the owner drags-and-drops the `Route to Users` action from the `Components` list (see Figure 6.7) into the page. Property 2 is inferred by the fact that the owner dropped the workflow action in `Evaluate Plans` page; therefore the type of record involved in the workflow action is a `Startup` record. The wizard can proceed in a series of questions. Property 3 comes from asking the owner to decide which user group to route `Startup` records. The answer in the running example is `Advisors`. Property 4 is implied by the choice of the `Route to Users` action, whose effect is that the involved record (`Startup` record) becomes readable by users of the chosen group (`Advisors`). Property 5 comes from asking the owner which is the page where `Advisors` will read `Startup` records; the owner will answer that is a new page, named `Advisor Comments`. Property 6 is addressed by a last question, where the owner chooses to send an email to the relevant users of the `Advisors` group.

Once the owner exits the wizard, the system automatically places a `solicit` action in the context of each `Startup`, along with a drop-down box that references the advisors, as shown in Figure 6.2.

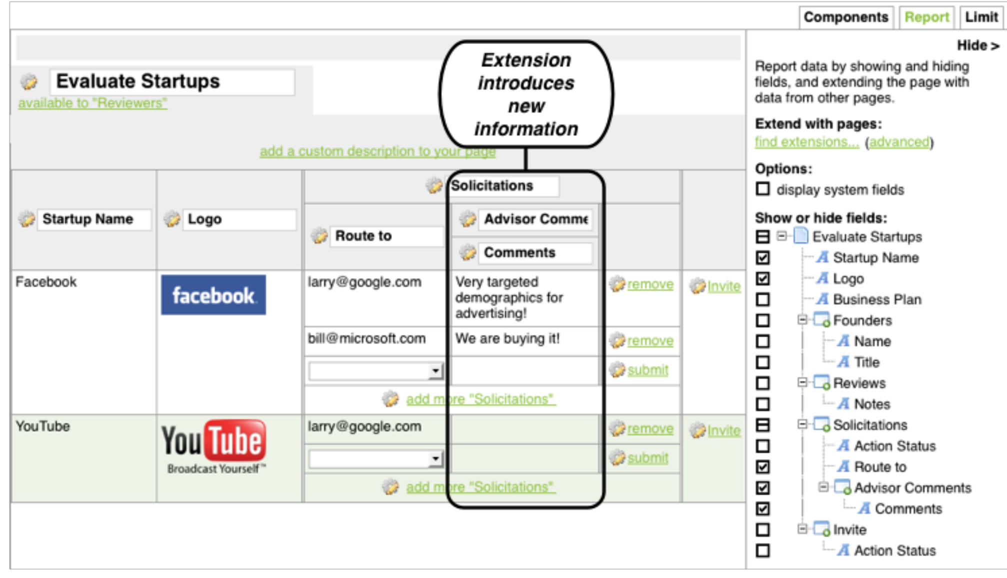
6.3.4 Automated Report Creation

Section 6.3.2 has demonstrated how the high-level specification of pages can generate a database schema, while Section 6.3.3 has shown how raising the specification level to that of application workflows can ease the design of actions. In keeping with this high level of specification, it is desirable for owners to design reports powered by complex queries, without having to specify low-level implementation details of queries such as projections, join conditions and selection conditions.

Since report pages are created after data collection pages the automated report creation can leverage semantic information previously specified by the owner. Consequently, app2you is able to provide the owner a minimal interface for designing complex reports, while compensating for this minimalism with algorithms that offer semantically meaningful options and automate implementation details.



(a) Access Rights



(b) Page Templates

Figure 6.8: Automated Report Extension on `Evaluate Startups` Page

Let us consider how the owner can extend the `Evaluate Startups` page with the comments that advisors have submitted on `Advisor Comments`. Such an augmented `Evaluate Startups` is shown in Figure 6.8b

Figure 6.8a shows the WYSIWYG design of `Evaluate Startups`, during

which the owner selects the `Report` tab and sees options for extending the page. For example, the first option corresponds to extending `Evaluate Startups` with data on `Advisor Comments`. The intuitive understanding is that selecting an option will cause the system to produce a more complex report, which is an amalgamation of both pages.

Figure 6.8b shows that after the above-mentioned option is selected, the system has introduced advisor comments by extending the `Solicitations` iterator's unit with data collected by `Advisor Comments`. Notice that the extension was placed at an optimal point, next to corresponding solicitations.

Through the WYSIWYG interface (and appropriate sample data), the owner receives immediate visual feedback of the extension. She can then perform further customization, such as hiding extraneous fields and iterators, deleting the extension and starting over, or repeat the design activity of extending the page.

This minimal interface is intended to capture the common case of designing reports. Sophisticated owners may choose to obtain explanatory details for an option in order to customize join conditions.

To enable this high degree of ease for the owner, the system has employed various DIY features and heuristics. The technical challenge lies in intelligently restraining the infinite space of all possible joins, to produce a summarized enumeration of options for the common case.

6.3.4.1 Generating Joins

When the owner selects the `Report` tab, the system produces the list of options by first generating a (finite) list of join paths. This is the core mechanism by which ultimately the owner chooses from enumerated options, rather than specify join conditions using arbitrarily complex Boolean conditions.

For each pair comprising an iterator b of the base page, (that is, the report page to be extended - `Evaluate Startups` in the example) and an iterator or an action e of any extension page, app2you attempts to find join paths that connect b and e. A join path is a left-deep relational join of the form:

$$FC(b) \ldots \bowtie_{c_n} FC(i_n) \bowtie_{c_{n+1}} \ldots FC(e)$$

where $FC(i)$ is the *flat context* of an iterator i.

Some example join paths between base iterators on the `Evaluate Startups` page and extension iterators on the `Advisor Comments` page are the following. For the sake of example, assume the `Advisor Comments` page also shows the `Founders`.

1.

$$FC(\texttt{Evaluate_Startups})$$
$$\bowtie_{\texttt{lhs.startup_id = rhs.startup_ref}}$$
$$FC(\texttt{Founders})$$

2.

$$FC(\texttt{Evaluate_Startups})$$
$$\bowtie_{\texttt{lhs.startup_id = rhs.startup_ref}}$$
$$FC(\texttt{Advisor_Comments})$$

3.

$$FC(\texttt{Solicitation})$$
$$\bowtie_{\texttt{lhs.startup_id = rhs.startup_ref AND lhs.route_to = rhs.submitted_by}}$$
$$FC(\texttt{Advisor_Comments})$$

The flat context of an iterator i is its corresponding non-parameterized query. If i is the top-level iterator of the page, then $FC(i)$ is simply the query producing the records displayed in i. If i is nested within iterator h, then $FC(i)$ is $FC(h)$ appropriately joined with the query producing the records displayed in i.

The join conditions c_n are conjunctions of equalities between attributes. Currently, the system considers two types of attribute join-pairs: (1) between id attributes, and corresponding foreign key attributes (2) between email attributes corresponding to user groups, and `Submitted By` / `Edited By` attributes of records

accessible by said user groups. This reflects the common intuition where the majority of join conditions involve unique identifiers, be they surrogate keys generated by the database or natural keys such as email addresses.

Note that the generated join paths do not contain cycles (i.e. an iterator can only occur once in the path), otherwise there can be an infinite number of paths. The exception is that b and e can be the same iterator, so that the owner can make arbitrary self-joins by choosing the same e for subsequent extension rounds.

6.3.4.2 Detecting Redundant Joins

The list of join paths generated is finite, but not all join paths are useful enough to present as options to the owner. For each join path, app2you makes a hypothetical extension of the base iterator, and uses view equivalency to test whether the extension adds only redundant information on the page.

For example, join path 1 is provably redundant and can be discarded, since there is already a `Founders` iterator on the base page `Evaluate Startups`.

A conservative definition of redundancy is the following: A join path is redundant if it leads to a new iterator x, where there is already an iterator y such that for all possible database instances that satisfy the schema and its constraints, each tuple $t_x = (v_1, \ldots, v_n)$ in $FC(x)$ has a corresponding tuple t_y in $FC(y)$ that has $v_1, \ldots, v_n$ and vice versa.

Note that such a definition does not prevent self-joins or, more generally, reports where a database table occurs multiple times as a result of different join conditions. The redundancy test is accomplished by essentially reducing all constraints into embedded dependencies, asserting the existence of t_x in $FC(x)$ and running a chase procedure [PDST00] that deduces tuples that must exist in the flat contexts of other iterators on the page.

6.3.4.3 Optimizing Join Placement

Given two generated join paths where the extension iterators are the same, one join path may be strictly better than the other. For example, contrast Join path 2 with 3. Extending `Evaluate Startups` with 2 will place advisor comments

on each startup, whereas 3 will place advisor comments on each solicitation. Intuitively, 3 is preferable to 2, as only the former visualizes the existing association between a solicitation to a specific advisor, and the corresponding comment.

This intuition can be expressed as functional dependencies between records. A startup can be routed at most once to each advisor, and an advisor can comment at most once on each startup. Therefore, a comment functionally determines a solicitation, which in turn functionally determines a startup. Since app2you relies heavily on WYSIWYG visualization to assist the owner in making design choices, it is important that wherever possible, functional dependencies and other constraints in the schema be visualized with the appropriate placement and nesting of iterators. Extending with 3 will produce a more accurate visualization of the functional dependencies.

Implementation-wise, running the chase procedure in Section 6.3.4.2 has the side benefit of also producing the necessary functional dependencies.

6.3.4.4 Bundling Additional Joins

After discarding pruned join paths, the surviving ones are aggregated by the pages of the extension iterators, and presented as a list of options as in Figure 6.8a. This achieves the minimal interface with a corresponding high-level of specification, as the owner only needs to comprehend pages (and not join paths) to start creating complex reports.

Note that the system uses the page rather than the iterator as the level of summarization. This comes from the observation that due to the parameterization between nested iterators, the standalone functionality of an iterator is harder to perceive than that of a page. Moreover, the existence of a report page is a strong hint that its structural organization is useful. Therefore, bringing in the entirety of the page en masse as part of the extension and allowing the owner to later hide extraneous fields and iterators provides better visual cues, than allowing the owner to extend one iterator at a time.

For an example, consider an alternate scenario where startups can provide rebuttals to advisor comments. There will be a page `Rebut Advisor Comments`,

that reports comments and annotates them with a `Rebuttal` iterator. If `Evaluate Startups` were not extended with `Advisor Comments`, but were instead extended with `Rebut Advisor Comments`, the bundling of additional joins will introduce both comments and rebuttals with a single round of extension.

6.3.4.5 Visualizing Projections

Iterators and fields can be easily shown and hidden with checkboxes (Figure 6.8b). For example, each iterator has a few hidden-by-default system fields, such as `Submit Timestamp`. The owner can easily customize the new `Advisor Comments` iterator to display when each advisor submitted her comment. From the DIY perspective, it is far preferable for the owner to toggle visibility of iterators and fields through an enumerated list, than to manually specify a projection list of attributes (a la query languages such as SQL).

6.4 Related Work

The time is opportune for Do-It-Yourself database-driven applications for two reasons. First, they leverage the emergence of hosted applications (software as a service) and Web 2.0 Ajax-based interfaces that allow application page design from the comfort of one's browser, while providing the richness of desktop interfaces. The two aspects combine to remove the hassles of (i) downloading/installing software in order to create an application and (ii) deploying/exporting an application on the web. But the Do-It-Yourself ability presents a larger, qualitatively-different challenge: How to disrupt conventional database-driven web application programming by providing brand new models of specifying database-driven web applications so that non-programmer business owners can build their own applications.

Multiple systems support the fast creation of custom web applications by removing the need to program in a complex Turing-complete programming language, such as Java. [CFB00] is a prime example of *schema-driven application creation* (also see DeKlarit [DeK] and Oracle Application Express [Ora]). The

creator starts by designing the Entity-Relationship data model for her application. Then it is easy to specify pages by putting together units that accomplish typical functionalities of Web applications. For example, a unit may report the data of an entity and utilize the relationships of the data model to navigate to related entities. It is reported in [ABB+07] that the development and maintenance of WebML applications led to 30% increased productivity with 46 distinct applications maintained by 5 part-time, junior developers.

The emerging Do-It-Yourself custom application platforms primarily target non-programmer process owners. A common theme is that the owner does not need to create a database schema in the abstract. Rather she builds forms, which automatically lead to corresponding tables that are typically reported on the same page. Such systems tend to be *online databases* [Wik10, eUn, Int] for easy information sharing and collaboration, often delivering great advantages over online spreadsheets, which are their main competitor for structured information sharing[7]. However, the resulting applications have a very limited scope (and business logic): Users simply post and read structured data in the shared space.

A next generation of Do-It-Yourself systems promises to go beyond information sharing and to enable users to capture their business processes by web applications. At a high level, these enablers are either "MS Access online" [Wik09b, YGB+08] or customizable vertical templates [Sal].

The "MS Access online" enablers allow users to create multiple Do-It-Yourself online tables (having forms and reports to give access to them). In the same spirit with MS Access, the reports have to be fueled by queries where the user has explicitly specified joins and selections. Finally, business logic and flow of data from table to table is offered in the form of scripting programming languages [Lon] or graphical languages [Wik09b] that allow the user to describe series of insertions, deletions and updates and the conditions under which they should happen. The adherence to tables with separate forms and reports creates problems at both the scope axis and the easy specification: The web applications we are dealing with

[7] Yahoo Pipes [Yah] and IBM Mashup Center [IBM] represent high end versions of the information sharing space, where data from multiple sources and RSS feeds can be automatically integrated and presented online.

day-to-day are not mere collections of tables with a report and a form for each table. A typical case is that the input forms of a page typically operate within the context of reported dynamic data and even within the context that prior pages create, i.e., there is no artificial divide of "input only" and "report only", as is clearly evidenced by pages such as `Evaluate Startups` and `Advisor Comments`. In addition to app2you, AppForge [YGB+08] also solves this problem by allowing input forms in the context of reports.

Another scope problem of "MS Access Online" is the inability to capture that access rights to a page may depend on the business logic itself. For example, in the TC50 application the group `Invited Applicants` is derived automatically and controls access to `Schedule an Appointment`.

The "MS Access online" class is problematic in creating workflow application since the business process owner needs to reduce the collaborative process she has in her mind into normalized tables and into sophisticated queries and updates. For example, we showed in Section 6.3.3 how hard it is to explain using a query that the `Advisor Comments` should only show startup submissions that have been passed to the currently logged-in user. This raises the bar of sophistication needed by the builders towards the level of sophistication that programmers have, therefore seriously limiting who can create and evolve applications. The anecdotal evidence behind this thesis is plenty: Instructors of undergraduate database classes know the difficulty that, even computer science students, have in designing appropriate schemas and writing non-trivial queries. Furthermore, despite the best efforts of tools, such as the tools of Microsoft Office Access and Microsoft InfoPath, to make database schema design and query writing approachable by the masses, the general public has found it hard to engage in those activities. The above evidence is not surprising since database schemas and queries are abstract structures that have no immediately visible connection to the web application and workflow aspects that the non-sophisticated designer can immediately associate with, which are the Web pages with which the users of the application will be interacting.

Applications with fixed workflow and database table structure and customizable input form structure (i.e., one can change the attributes of tables as

long as the tables and their interactions remain fixed) have been a great success [Sal]. We believe that customization does not need to stop at that point since, by doing so, the scope of available applications is limited by the available initial templates.

Appendix

More than twenty forms-driven applications have been built and used in 2008 on app2you.com. For example, a recruiter has collected job openings from its customers. A wide group of users, defined and controlled by the recruiter, sees selected fields of the job openings' records and is invited to recommend individuals, who are notified about the positions, provide their level of interest and proceed to exchange information with the customer and the recruiter if interested.

In another example, the United Cerebral Palsy non-profit organization maintains an online loan library of toys, keeping track of who currently holds a toy and who has requested it.

In multiple variations of classroom management applications students submit their projects, often after a phase where they have teamed up in project teams. The TAs and instructor provide feedback and grade. Variations include setting up appointments for project presentations and rehearsals, voting for the best project etc.

In multiple variations of reviewing applications, candidates submit material that is pushed through a review process with various rules and steps.

Acknowledgements

This chapter contains material from “Do-It-Yourself Database-Driven Web Applications”, by Keith Kowalczykowski, Kian Win Ong, Kevin Keliang Zhao, Alin Deutsch, Yannis Papakonstantinou and Michalis Petropoulos, which appears in Proceedings of the Fourth Biennial Conference on Innovative Data Systems Research, CIDR 2009. The dissertation author was the primary investigator and

author of this paper. The paper is published under a Creative Commons License Agreement (http://creativecommons.org/licenses/by/3.0/). Permission is granted to copy, distribute, display, and perform the work, make derivative works and make commercial use of the work, but the work must be attributed to the authors and CIDR 2009.

Bibliography

[AASY97] Divyakant Agrawal, Amr El Abbadi, Ambuj K. Singh, and Tolga Yurek. Efficient view maintenance at data warehouses. In *SIGMOD*, pages 417–427, 1997.

[ABB+07] Roberto Acerbis, Aldo Bongio, Marco Brambilla, Massimo Tisi, Stefano Ceri, and Emanuele Tosetti. Developing ebusiness solutions with a model driven approach: The case of acer emea. In *ICWE*, pages 539–544, 2007.

[AFP03] M. Akhtar Ali, Alvaro A. A. Fernandes, and Norman W. Paton. Movie: An incremental maintenance system for materialized object views. *Data Knowl. Eng.*, 47(2):131–166, 2003.

[AMR+98] Serge Abiteboul, Jason McHugh, Michael Rys, Vasilis Vassalos, and Janet L. Wiener. Incremental maintenance for materialized views over semistructured data. In *VLDB*, pages 38–49, 1998.

[AVFY98] Serge Abiteboul, Victor Vianu, Brad Fordham, and Yelena Yesha. Relational transducers for electronic commerce. In *PODS '98: Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 179–187, New York, NY, USA, 1998. ACM.

[Bac09] Backbase enterprise ajax framework, 2009. `http://www.backbase.com/products/enterprise-ajax/`.

[BCD89] François Bancilhon, Sophie Cluet, and Claude Delobel. A query language for the o_2 object-oriented database system. In *DBPL*, pages 122–138, 1989.

[BGvK+06] Peter A. Boncz, Torsten Grust, Maurice van Keulen, Stefan Manegold, Jan Rittinger, and Jens Teubner. Monetdb/xquery: a fast xquery processor powered by a relational engine. In *SIGMOD Conference*, pages 479–490, 2006.

[BLT86] José A. Blakeley, Per-Åke Larson, and Frank Wm. Tompa. Efficiently updating materialized views. In *SIGMOD Conference*, pages 61–71, 1986.

[Bor07] Jeanette Borzo. Do-it-yourself software. *Wall Street Journal*, September 2007. `http://online.wsj.com/article/SB119023041951932741.html`.

[CAL+02] K. Selçuk Candan, Divyakant Agrawal, Wen-Syan Li, Oliver Po, and Wang-Pin Hsiung. View invalidation for dynamic content caching in multitiered architectures. In *VLDB*, pages 562–573, 2002.

[CFB00] Stefano Ceri, Piero Fraternali, and Aldo Bongio. Web modeling language (webml): a modeling language for designing web sites. *Computer Networks*, 33(1-6):137–157, 2000.

[CGL+96] Latha S. Colby, Timothy Griffin, Leonid Libkin, Inderpal Singh Mumick, and Howard Trickey. Algorithms for deferred view maintenance. In *SIGMOD*, pages 469–480, 1996.

[DeK] Deklarit. `http://www.deklarit.com`.

[DESR03] Katica Dimitrova, Maged El-Sayed, and Elke A. Rundensteiner. Order-sensitive view maintenance of materialized xquery views. In *ER*, pages 144–157, 2003.

[DMS+05] Alin Deutsch, Monica Marcus, Liying Sui, Victor Vianu, and Dayou Zhou. A verifier for interactive, data-driven web applications. In *SIGMOD Conference*, pages 539–550, 2005.

[Ech09] Echo web framework, 2009. `http://echo.nextapp.com/site/`.

[eUn] eunify. `http://www.eunify.net/`.

[FFLS00] Mary F. Fernández, Daniela Florescu, Alon Y. Levy, and Dan Suciu. Declarative specification of web sites with strudel. *VLDB J.*, 9(1):38–55, 2000.

[FKSV08] J. Nathan Foster, Ravi Konuru, Jérôme Siméon, and Lionel Villard. An algebraic approach to view maintenance for xquery. In *PLAN-X*, 2008.

[FPF+09] Ghislain Fourny, Markus Pilman, Daniela Florescu, Donald Kossmann, Tim Kraska, and Darin McBeath. Xquery in the browser. In *WWW*, pages 1011–1020, 2009.

[Gar05] Jesse James Garrett. Ajax: A new approach to web applications. http://adaptivepath.com/ideas/essays/archives/000385.php, February 2005. [Online; Stand 18.03.2008].

[GL95] Timothy Griffin and Leonid Libkin. Incremental maintenance of views with duplicates. In Michael J. Carey and Donovan A. Schneider, editors, *SIGMOD*, pages 328–339. ACM Press, 1995.

[GM95] Ashish Gupta and Inderpal Singh Mumick. Maintenance of materialized views: Problems, techniques, and applications. *IEEE Data Eng. Bull.*, 18(2):3–18, 1995.

[GMS93] Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In Peter Buneman and Sushil Jajodia, editors, *SIGMOD*, pages 157–166. ACM Press, 1993.

[Goo09] Google widget toolkit, 2009. http://code.google.com/webtoolkit/.

[GST04] Torsten Grust, Sherif Sakr, and Jens Teubner. Xquery on sql hosts. In *VLDB*, pages 252–263, 2004.

[HA00] Md. Ahsan Habib and Marc Abrams. Analysis of sources of latency in downloading web pages. In *WebNet*, pages 227–232, 2000.

[IBM] Ibm mashup center. http://www-01.ibm.com/software/info/mashup-center/.

[ICE09] Icefaces, 2009. http://www.icefaces.org/main/home/.

[Int] Intuit quickbase. http://quickbase.intuit.com/.

[Joh09] Bruce Johnson. Reveling in constraints. *Queue*, 7(6):30–37, 2009.

[Kra05] Joe Kraus. The long tail of software. millions of markets of dozens. http://bnoopy.typepad.com/bnoopy/2005/03/the_long_tail_o.html, March 2005.

[Lon] Longjump. http://longjump.com/.

[LZ07] Per-Åke Larson and Jingren Zhou. Efficient maintenance of materialized outer-join views. In *ICDE*, pages 56–65, 2007.

[Mir03] Samek Miro. Who moved my state? *Dr. Dobb's*, 2003.

[MQM97] Inderpal Singh Mumick, Dallan Quass, and Barinderpal Singh Mumick. Maintenance of data cubes and summary tables in a warehouse. In *SIGMOD*, pages 100–111, 1997.

[Nin] Ning. http://www.ning.com/.

[oA09] Communications Workers 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.

[O'H06] Charlene O'Hanlon. A conversation with werner vogels. *Queue*, 4(4):14–22, 2006.

[Ora] Oracle application express. http://www.oracle.com/technology/products/database/application_express/index.html.

[PCS+05] Shankar Pal, Istvan Cseri, Oliver Seeliger, Michael Rys, Gideon Schaller, Wei Yu, Dragan Tomic, Adrian Baras, Brandon Berg, Denis Churin, and Eugene Kogan. Xquery implementation in a relational database system. In *VLDB*, pages 1175–1186, 2005.

[PDST00] Lucian Popa, Alin Deutsch, Arnaud Sahuguet, and Val Tannen. A chase too far? In *SIGMOD Conference*, pages 273–284, 2000.

[QGMW96] Dallan Quass, Ashish Gupta, Inderpal Singh Mumick, and Jennifer Widom. Making views self-maintainable for data warehousing. In *Proceedings of the Fourth International Conference on Parallel and Distributed Information Systems, December 18-20, 1996, Miami Beach, Florida, USA*, pages 158–169. IEEE Computer Society, 1996.

[RKS88] Mark A. Roth, Henry F. Korth, and Abraham Silberschatz. Extended algebra and calculus for nested relational databases. *ACM Trans. Database Syst.*, 13(4):389–417, 1988.

[RSS96] Kenneth A. Ross, Divesh Srivastava, and S. Sudarshan. Materialized view maintenance and integrity constraint checking: Trading space for time. In *SIGMOD*, pages 447–458, 1996.

[Rub] Ruby on rails. http://rubyonrails.org/.

[Sal] Salesforce.com. http://www.salesforce.com/.

[SBCL00] Kenneth Salem, Kevin S. Beyer, Roberta Cochrane, and Bruce G. Lindsay. How to roll a join: Asynchronous incremental view maintenance. In *SIGMOD*, pages 129–140, 2000.

[Sim09] Lindsey Simon. Minimizing browser reflow, 2009. http://code.google.com/speed/articles/reflow.html.

[STP+05] Arsany Sawires, Jun'ichi Tatemura, Oliver Po, Divyakant Agrawal, and K. Selçuk Candan. Incremental maintenance of path expression views. In *SIGMOD*, pages 443–454, 2005.

[Tec] Techcrunch50 2008. `http://www.techcrunch50.com/2008/`.

[The09] The dojo toolkit, 2009. `http://www.dojotoolkit.org/`.

[TV07] Colin Teubner and Ken Vollmer. Bpms revenue to reach $6.3 billion by 2011. Technical report, Forrester Research, 2007. Archived at `http://db.ucsd.edu/app2you/2009-www/2007-forrester-bpms.pdf`.

[Vis98] Dimitra Vista. Integration of incremental view maintenance into query optimizers. In *EDBT*, pages 374–388, 1998.

[Wik09a] Wikipedia. Asp.net, 2009. Accessed Nov 04 2009. `http://en.wikipedia.org/w/index.php?title=ASP.NET&oldid=323456166`.

[Wik09b] Wikipedia. Coghead, 2009. Accessed Jul 26 2010. `http://en.wikipedia.org/wiki/Coghead`.

[Wik10] Wikipedia. Dabble, 2010. Accessed Jul 26 2010. `http://en.wikipedia.org/wiki/Coghead`.

[Yah] Yahoo! pipes. `http://pipes.yahoo.com/pipes/`.

[YGB+08] Fan Yang, Nitin Gupta, Chavdar Botev, Elizabeth F. Churchill, George Levchenko, and Jayavel Shanmugasundaram. Wysiwyg development of data driven web applications. *PVLDB*, 1(1):163–175, 2008.

[YGG+07] Fan Yang, Nitin Gupta, Nicholas Gerner, Xin Qi, Alan J. Demers, Johannes Gehrke, and Jayavel Shanmugasundaram. A unified platform for data driven web applications with automatic client-server partitioning. In *WWW*, pages 341–350, 2007.

[YSRG06] Fan Yang, Jayavel Shanmugasundaram, Mirek Riedewald, and Johannes Gehrke. Hilda: A high-level language for data-drivenweb applications. In *ICDE*, page 32, 2006.

[YUI09] Yui library, 2009. `http://developer.yahoo.com/yui/`.

[YYY+03] Ke Yi, Hai Yu, Jun Yang, Gangqiang Xia, and Yuguo Chen. Efficient maintenance of materialized top-k views. In *ICDE*, pages 189–200, 2003.

[ZGM98] Yue Zhuge and Hector Garcia-Molina. Graph structured views and their incremental maintenance. In *ICDE*, pages 116–125, 1998.

[ZGMHW95] Yue Zhuge, Hector Garcia-Molina, Joachim Hammer, and Jennifer Widom. View maintenance in a warehousing environment. In *SIGMOD*, pages 316–327, 1995.

[ZK 09] Zk direct ria, 2009. `http://www.zkoss.org/`.