DEVELOPING, OPTIMIZING AND HOSTING DATA-DRIVEN WEB APPLICATIONS

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Fan Yang

August 2008

© 2008 Fan Yang

ALL RIGHTS RESERVED

DEVELOPING, OPTIMIZING AND HOSTING DATA-DRIVEN WEB APPLICATIONS

Fan Yang, Ph.D.

Cornell University 2008

Building web applications using current systems is not an easy task and we face the following challenges: (1) It is difficult to program web applications on top of the standard three-tier architecture. (2) Performance optimizations and tunings are mostly done manually, which is tedious, error-prone and suboptimal. (3) It is hard for non-technical users to construct web applications for their own needs. (4) Current platforms do not scale to host a large number of applications in a cost-effective, manageable and/or flexible manner. In this thesis, we propose technologies to address those challenges in developing, optimizing and hosting data-driven web applications.

Data-Driven web applications are usually structured following the standard three-tier architecture with different programming models used at different tiers. This division not only creates an impedance mismatch problem for developers but also forces them to manually partition application logic across tiers, which results in complex logic, suboptimal system design, and expensive re-partitioning of applications as systems evolve. We propose a unified development platform based on HILDA, a high-level language for developing data-driven web applications. The primary benefits of HILDA over existing development platforms are: (a) it uses a unified data and programming model for all layers of the application, (b) it is declarative, (c) it enables conflict detection for concurrent updates, (d) it supports structured programming for web

sites, (e) it separates application logic from presentation. Instead of using different languages for different layers, developers build the whole application in HILDA. HILDA code is translated into executables that run on top of the three-tier architecture. The runtime system automatically partitions the application logic between tiers based on runtime properties of the application, to optimize the system performance while obeying memory constraints at the clients. We evaluate our methodology with traces from a real Course Management System used at Cornell University as well as an online bookstore from the TPC-W benchmark. The results show that automatic partitioning outperforms manual partitioning without the associated development overhead.

There are many cases where non-technical users want to build data-driven web applications to fit their own needs. An emerging trend in Social Networking sites and Web portals is the opening up of APIs to external application developers. For example, the Facebook Platform, Google Gadgets and Yahoo! Widgets allow users to design their own applications, which can then can be integrated with the platform and shared with others. However, current APIs are targeted towards developers with programming expertise and database knowledge; they are not accessible to a large class of users who do not have a programming/database background but would nevertheless like to create new applications. To address this need, we have developed the AppForge system, which provides a WYSIWYG application development platform. Users can graphically specify the components of webpages inside a Web browser, and the corresponding database schema and application logic are automatically generated on the fly by the system. The WYSIWYG interface gives instantaneous feedback on what users just created and allows them to run, test and continuously refine their applications and greatly lower the bar for building such applications.

While each user-generated application by itself is quite small (in terms of size and throughput requirements), there are many such applications and existing data management solutions are not designed to handle this form of scalability in a cost-effective, manageable and/or flexible manner. For instance, large installations of commercial database systems such as Oracle, DB2 and SQL Server are usually very expensive and difficult to manage. At the other extreme, low-cost data hosting solutions such as Amazon's SimpleDB do not support sophisticated data manipulation primitives such as joins that are necessary for developing most Web applications. To address this issue, we explore a new point in the design space whereby we use commodity hardware and free software (MySQL) to scale to a large number of applications while still supporting full SQL functionality, transactional guarantees, high availability and Service Level Agreements (SLAs). We do so by exploiting the key property that each application is "small" and can fit in a single machine (which can possibly be shared with other applications). Using this property, we design replication strategies, data migration techniques and load balancing operations that automate the tasks that would otherwise contribute to the operational and management complexity of dealing with a large number of applications. We have conducted extensive experiments, based on the TPC-W benchmark data sets and workloads, to study the performance aspects of our system. Our experiments demonstrate that our system can host a very large number of Web applications and provide them rich functionality, strong consistency, high performance, high availability and data protection in an inexpensive manner by using commodity hardware and software components.

BIOGRAPHICAL SKETCH

Fan Yang is a PhD Candidate at Computer Science Department in Cornell University, advised by Prof. Johannes Gehrke. His research interests lie in the platforms for developing data driven web applications with focus on the declarative programming model for developing such applications and automatic system optimizations based on the declarative model. His more recent work includes improving usability of DB systems for non-technical users and scalability for web-scale data platforms. Before studying in Cornell University, he received B.S. degree from Peking University, China.

This document is dedicated to my family.

ACKNOWLEDGEMENTS

First of all, I would like to express my deepest sense of gratitude to my advisors Prof. Johannes Gehrke and Dr. Jayavel Shanmugasundaram for their patient guidance, encouragement, and excellent advice throughout my PHD study.

I am thankful to Prof. Alan Demers, Dr. Mirek Riedewald and my colleagues Mr. Nicholas Gerner, Mr. Nitin Gupta, Mr. Xin Qi, Mr. Feng Shao, Ms. Lin Guo, Mr. Anand Bhaskar, Mr. Chavdar Botev, and Mr. Muthiah M Muthaia for being helpful and inspiring during my research studies.

Finally, I take this opportunity to express my profound gratitude to my beloved family, for their support and patience during my studies at Cornell University.

TABLE OF CONTENTS

Biographical Sketch iii
Dedication iv
Acknowledgements v
Table of Contents vi
List of Tables viii
List of Figures ix

1 Introduction **1**
1.1 Declarative Language For Developing Data-Driven Web Applications 1
1.2 Across Tiers Optimization for Data-Driven Web Applications . . 5
1.3 Building Data-Driven Web Applications by Non-Technical Users 7
1.4 Hosting A Large Number of User-Created "Small" Applications 12
1.5 Organization of the Dissertation 15

2 HILDA: High Level Declarative Language for Data-Driven Web Applications **17**
2.1 Case Study: A Course Management System 17
2.1.1 Assignment Creation 18
2.1.2 Viewing Student Grades 19
2.1.3 Student Group Management 20
2.1.4 Web Site Structure 21
2.2 The HILDA Language 21
2.2.1 Design Decisions 22
2.2.2 AUnits Overview 24
2.2.3 User-Defined AUnits 26
2.2.4 AUnits: Inheritance 38
2.2.5 PUnits 39
2.3 Discussion 42

3 Automatic Client-Server Partitioning for Data-Driven Web Applications **43**
3.1 Partitioning Example 43
3.2 The HILDA Runtime System 45
3.3 Model of Client-Server Partitioning 47
3.3.1 Partitioning Philosophy 47
3.3.2 Terminology 48
3.3.3 Cost Model 50
3.3.4 Solution For Partitioning 53
3.4 Experimental Evaluation 60
3.4.1 Experimental Setup 60
3.4.2 Experimental Results 63

4 **WYSIWYG Development for Data-Driven Web Applications** **70**
4.1 AppForge System Overview 70
4.1.1 System Architecture 70
4.1.2 AppForge GUI 72
4.1.3 Running Example 73
4.2 AppForge Application Model 80
4.2.1 Background 81
4.2.2 Application Model 82
4.3 Constructing Views 84
4.3.1 Schema Navigation Menu 85
4.3.2 Graphical Primitives for Editing Views 87
4.3.3 Expressiveness Theorem 89
4.4 Automatic Schema Generation 93
4.4.1 Editing entities 93
4.4.2 Editing Relationships 95
4.4.3 Expressiveness Theorem 98
4.5 Preliminary user study of AppForge interface 99

5 **Hosting A Large Number of ”Small” Database Applications** **104**
5.1 System Overview 104
5.2 Fault Tolerance 108
5.2.1 Implementing Synchronous Replication 110
5.2.2 Sufficient Condition for Global Serializability 115
5.2.3 Database Migration 116
5.3 Enforcing Service Level Agreement Guarantees 120
5.3.1 Problem Definition 120
5.3.2 Measuring SLA and Resource Requirement 122
5.3.3 SLA Based Database Placement 124
5.3.4 Enforcing SLA During Database Migration 125
5.4 Experiments 126
5.4.1 Experiment Setup 126
5.4.2 Varying Database Sizes and Workloads 128
5.4.3 Recovery 131
5.4.4 SLA Based Placement 134

6 **Conclusions and Future Work** **135**

7 **Related Work** **138**
7.1 Systems and Tools for Building Data-Driven Web Applications . . 138
7.2 Client-Server Partitioning for Distributed Applications 141
7.3 Graphical Tools For Building Data-Driven Web Applications . . . 143
7.4 Shared Data Hosting Systems 146

Bibliography **149**

LIST OF TABLES

3.1 Operations in CMS 64
3.2 Average Response Time and Data Transmission for CMS 65
3.3 Operations in TCP-W Online Bookstore Application 68
3.4 Average Response Time and Data Transmission for TPC-W . . . 68

5.1 Different Strategies for Performing Read Operations. 112
5.2 Serializability with Strict 2PL Optimization + 2PC 113
5.3 Experiment Settings and Results for SLA Based Placement 133

LIST OF FIGURES

1.1 Tiers in a Data-Driven Web Application 2
1.2 Sortable Table Example 5

2.1 Course Management System 18
2.2 AUnits in CMS 26
2.3 BNF grammar for a User-Defined AUnit 27
2.4 The CMSRoot AUnit 29
2.5 Student AUnit. 30
2.6 Activation Phase 34
2.7 Reactivation Phase 35
2.8 Grammar for AUnit Inheritance. 39
2.9 NavCMS inherits from CMS 40
2.10 PUnit example 40

3.1 Web Page for edit staff and the underlying activation tree 43
3.2 Activation tree with different partition strategies 44
3.3 System Architecture 45
3.4 Average Response Time for Operations in CMS 66
3.5 Average Amount of Data Transmitted for Operations in CMS . . 66
3.6 Average Response Yime for Operations in TPC-W 69
3.7 Average Amount of Data Transmitted for Operations in TPC-W . 69

4.1 AppForge System Architecture 71
4.2 AppForge GUI 72
4.3 Adding a form for creating new events. The resulting form is shown in Figure 4.4. 74
4.4 Adding a table to show existing events. The resulting table is shown in Figure 4.5. 74
4.5 Adding the presentation column to the table. Organizers can add speakers for presentations. The resulting table is shown in Figure 4.6. 75
4.6 Adding the topic column to the presentation nested table. The resulting table is shown in Figure 4.2. 76
4.7 Start creating the View Volunteers page by creating a view over *Speaker* and *Event* 77
4.8 Adding the volunteer column to the *Event* nested table 77
4.9 The View Volunteers Page 77
4.10 Start creating the View Comments page by creating a view over speaker, event and attendee 78
4.11 Creating a filter to show only past events 78
4.12 Adding a ratings column for each attendee 79
4.13 The View Comments page 79
4.14 Automatically generated database schema 80

4.15 Flat and Nested Tables 82
4.16 The schema generated for the Create Event page (Figure 4.2) . . 95
4.17 The schema generated from View Volunteer page (Figure 4.9) . . 96
4.18 The schema generated from View Comments page (Figure 4.13) . 97
4.19 Multiple levels of abstractions for developers 101
4.20 Personalization 101
4.21 Viewing Pages as a Specific User 101

5.1 System Overview 104
5.2 Architecture of a DB Cluster. 107
5.3 Within each DB Cluster: Recovery. 107
5.4 Within each DB Cluster: Load Balancing. 108
5.5 Throughput for browsing mix. 129
5.6 Throughput for shopping mix. 129
5.7 Throughput for ordering mix. 130
5.8 Deadlock rate for browsing mix. 130
5.9 Deadlock rate for shopping mix. 131
5.10 Deadlock rate for ordering mix. 131
5.11 Throughput during recovery. 133
5.12 Total time to finish recovery. 133
5.13 Rejected transactions per application during recovery. 134

CHAPTER 1

INTRODUCTION

In this thesis, we propose novel technologies for both professional developers and non-technical users to develop, optimize and host data-driven web applications. We make four contributions: (1) HILDA, a new domain specific language, that provides a declarative and unified approach to build data-driven web applications. (2) A platform that takes advantage of HILDA to automatically optimize the system performance across tiers. (3) A WYSIWYG interface that enables non-technical users to build data-driven web applications. (4) A scalable backend data hosting system that can host large number of applications created by users.

1.1 Declarative Language For Developing Data-Driven Web Applications

An important class of applications are *data-driven web applications*, i.e., web applications that run on top of a backend database system. Examples of such applications include online shopping sites, online auctions, and business-to-business portals. Data-driven web applications normally follow the standard three-tier architecture (Figure 1.1): (1) *Database* at the backend, such as MySQL and DB2 which manage and store the persistent data. (2) *Application Server* in the middle layer, such as JBoss and Websphere which host the application logic for performing user actions. (3) *User Interface* at the frontend, such as web browsers in personal computers or PDAs which allow users to interact with the system.

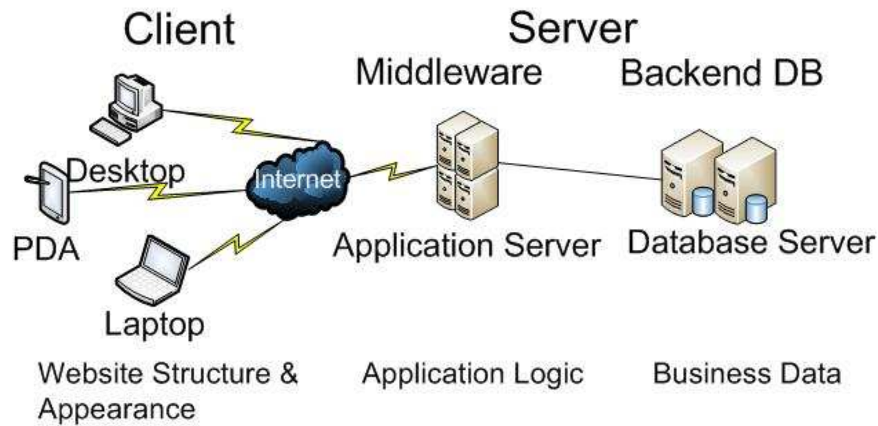

Figure 1.1: Tiers in a Data-Driven Web Application

While developing data-driven web applications is a complex and challenging task, the application development interface provided by existing platforms is often too low-level or does not provide a unified model across the different application layers. Specifically, while technologies such as J2EE, Java Servlets/JSPs, ASPs, PHP, WebML and Strudel simplify application development to some extent, they suffer from some of the following shortcomings.

Impedance mismatch: Most existing languages provide a different data model for each application layer (e.g., relational model for databases, Java objects for application logic, hyperlinks for website structure, and form variables for web pages). This "impedance mismatch" makes it hard to develop, maintain, and optimize applications.

Not declarative: In contrast to declarative high-level database query languages such as SQL, web application development languages such as Java are low-level and procedural. This increases application development time and limits optimization opportunities.

Manual-partition of the application logic across tiers: Exposing the bound-

aries between tiers to the programmer requires them to manually partition the logic across tiers during development time. This not only lays extra burden on developers and reduce the productivity but also can result in suboptimal solutions. Currently, manual-partitions are mostly done based on the developer's personal experience and not on systematic metrics and criteria. The optimal solutions normally depend on runtime properties, e.g., data size, network conditions, client types (PDA vs desktop clients) that cannot be known accurately in advance during the development.

No support for application conflict detection: Multi-user, data-driven applications, by their very nature, have a potential for conflicts due to concurrently issued application updates. As we shall illustrate in Chapter 2.1.3, such *application-level conflicts* cannot always be handled by database transactions, and in complex applications, such conflicts can be very hard to detect. Existing systems do not provide support for conflict detection.

Mixing of application logic and presentation: While there is a broad agreement that application logic should be kept separate from presentation, many existing languages do not enforce this separation; it results in code that is hard to understand, modify, and extend.

No unified handling of queries and updates: While some tools such as AutoWeb [53] and Strudel [44] can declaratively specify the structure and content of web sites, they focus mostly on read-only applications. Consequently, they do not provide a uniform framework for handling applications that deal with both queries and updates.

No structured programming for web sites: Website specification tools such as

WebML [43] and Strudel [44] represent a data-driven web site as a graph, where the nodes in the graph are web pages and the edges are links between the pages. Consequently, the control flow of the application can jump from one web page to another so long as there is a connecting edge. This is similar to programming with goto statements in the domain of web pages, and has similar disadvantages as compared to structured programming [33].

To address the above issues, we proposed HILDA, a new domain specific language designed specifically for data-driven applications.

The design of HILDA embodies several key concepts. First, HILDA uses a single data model, the relational model [24], to represent all application state. Second, it captures application logic as a sequence of state transitions from one valid application state to another. Each transition contains a condition on when to trigger the transition, and an action on how to transform current state to the new state. The application query and update operations are declaratively specified using SQL. Third, HILDA provides an application building block called an *AUnit* (for Application Unit), analogous to a UML class [41]. AUnits support encapsulation like a regular UML class, but the creation and manipulation of AUnits is specified declaratively and provides natural support for conflict detection in the face of concurrent application updates. AUnits are single-entry and single-exit, which facilitates structured programming. Finally, HILDA separates the application logic, which is represented as AUnits, from the presentation, which is represented as *PUnits* (for Presentation Units) with embedded HTML code.

First Name ▲▼	Last Name ▲▼	Grade ▲▼
Fan	Yang	A
Brain	Foley	A+
David	Brooks	B-

Figure 1.2: Sortable Table Example

1.2 Across Tiers Optimization for Data-Driven Web Applications

As mentioned in the previous section, in current systems, developers need to partition the application logic across multiple tiers manually during the development time. For example, in a Course Management System [36], we have a sortable table showing student information (Figure 1.2). The table can be sorted based on different columns, depending on which column users click on. To implement the sortable table, we can either sort using SQL at the backend database, Java in application server or cache the data at the client side and sort in Javascript. Which implementation is the best strategy depends on several runtime properties, e.g., the size of the table and types of the clients. If the table size is large or the client is a PDA with very limited main memory resource, the server-side solution should be used. Otherwise, pushing data and functionality to the client should give better a user experience. With current systems, the decisions are made manually and statically. Such practice has four significant drawbacks.

Increased development time. Having different programming models in different tiers makes it hard to develop, maintain, and optimize applications, as the developer must manually bridge the differences between the individual models

(for example, the relational model, EJBs, and HTML forms).

Complex logic due to partitioning. Partitioning application logic across the tiers requires complex logic to synchronize the client-server state of the application. For example, in order to enable partial updates (a well known strategy in AJAX [3]), data can be cached at the client side.

Suboptimal partitioning. Since the decision of how to partition the application is left to the developer (who may have little data on which to base her decisions), the resulting division of the application may be be suboptimal in terms of system performance.

Expensive re-partitioning. Once a partitioning of the application has been implemented, moving functionality between layers is complex. As in the sortable table example, it may initially be implemented in Java and SQL. Then later we decide to move it to the client side to improve responsiveness, the sortable table must be reimplemented in Javascript which is not a trivial change.

Our system solves the problem by partitioning HILDA programs between clients and servers based on the monitored runtime behavior of the application — all of this is completely transparent to the developer. The system automatically synchronizes state between client and server without the developer having to write any additional code to achieve this. A web application developer thus can focus on the core application logic without worrying about partitioning the application or about changes to the partition.

1.3 Building Data-Driven Web Applications by Non-Technical Users

While we have designed HILDA to simplify the process of developing applications for professional developers, there is also an increasing need to enable non-technical users to build their own applications, as the world moving towards Web 2.0. For example, in Facebook and Yahoo! Groups, different groups of users have different needs, and it is difficult for these websites to build applications that satisfy all of these needs. Thus websites are starting to open up their APIs to their advanced users so that they can build new applications that can be deeply integrated with the websites, e.g., the Facebook Platform [82], Yahoo! Widgets [105] and Google Gadgets [55].

However, current APIs and tools are primarily targeted towards developers who have programming and database knowledge. Consequently, they are beyond the reach of the majority of users who lack this knowledge, but would nevertheless like to create and share their own custom applications. For instance, members of a book club in Yahoo! Groups may wish to create a custom application for managing their club events (since no third party application is available to satisfy their specific needs), but the group members may not have the necessary programming expertise to develop this application. Even though there has been a lot of work on designing languages and tools to simplify application development, ranging from high level programming languages such as Ruby on Rails [79] to visual programming tools such as Visual Basic [10] and Oracle Forms [50] to various CASE tools such as UML [16] and WebML [19], the abstractions that these tools provide is still too complex for non-technical users.

Recently, there has been a flurry of activity on providing online Web application creation services for advanced users[1]. Examples of such websites are Yahoo! Pipes [81], Microsoft Popfly [85], App2You [5], CogHead [25], Zoho Creator [30], Ning [77], Dabble DB [31], WyaWorks [109], JotSpot [66] and SalesForce [90]. These websites allow developers to graphically build web pages and the associated application logic in browsers, thereby greatly lowering the bar for building Web applications. However, these systems suffer from at least one of the following three drawbacks, which limit their applicability and generality.

1. **Non-WYSIWYG development environment.** Most systems (e.g., [5], [25], [30], [77], [81], [90], [109]) have at least two modes: (1) *development mode*, where developers can edit the page structure, application logic and/or database schema, and (2) *execution mode*, where developers and users can actually run and test the application. Consequently, developers have to visualize what they want in the execution mode (i.e., what the end-users will see) and mentally map these into corresponding constructs in the development mode, which results in a significant impedance mismatch. As a loose analogy, consider two popular typesetting tools: LaTeX and Microsoft Word. In LaTeX, users have to mentally map what they want in the final document to the corresponding LaTeX commands, while in Microsoft Word, they directly edit the final document using a WYSIWYG interface. While both approaches have their advantages, the WYSIWYG environment is more accessible to a larger class of users, as also pointed out in [64].

2. **Limited support for creating stateful applications with complex struc-**

[1]Henceforth, to avoid confusion with end users, we shall refer to advanced users as *developers*; these are not to be confused with professional developers.

tures such as relationships. Some systems (e.g., [81], [85]) only support stateless web applications with read-only operations, while some other systems (e.g., [66]) only support stateful applications with predefined and restricted structures. A few systems (e.g., [25], [30], [31], [109]) do support sophisticated stateful applications, including advanced features such as relationships between entities, but they require developers to be familiar with relational database schema design. As an illustration, consider a book club event planning application, which includes information about events, speakers and attendees, and also about the rating provided by each attendee for each speaker in the event. In effect, the application state contains three entities — event, speaker and attendee — and a 3-way relationship with rating information that connects the three entities. In order to capture this state using the aforementioned systems, developers have to explicitly create a table that connects event, speaker and audience, e.g. using foreign key columns, and also create a column in that table for storing the rating information. In general, such a process is equivalent to creating an Entity-Relationship (E-R) graph [20] and translating it to relational tables, which is challenging.

3. **Limited support for publishing views over multiple related entities.** Many Web applications need to publish pages with complex views of the application state, which could include multiple related entities. For example, in our book club application, we may wish to display the audience for each event, which requires "joining" the events with their corresponding audience through a relationship. As pointed out in [64], such join queries in their traditional form are unnatural for average users. Current systems either do not support the creation of such views (e.g., [66]) or

assume that developers have database schema and programming knowledge (e.g. [25], [30], [31], [77], [109]).

To address the above issues, we have developed the AppForge system, which enables developers to graphically build sophisticated applications inside Web browsers without much programming or database knowledge. AppForge offers the following advantages over existing systems.

1. **AppForge provides a WYSIWYG environment.** AppForge seamlessly integrates the process of page design, application logic design, schema design, deployment and testing of applications. As developers interact with the system by changing the presentation model on web pages, AppForge automatically generates the underlying database schema and application logic on the fly. Developers get instantaneous feedback on pages when modifications are made to the application logic, database schema and database queries. This allows developers to test, run and continuously refine the application as they are constructing it. The WYSIWYG interface is especially important in our setting, since the developers we are targeting at are expected to constantly make mistakes before producing the desired output.
2. **AppForge enables non-programmers to create sophisticated stateful applications.** Developers just need to focus on building what they want to show in each application page, and AppForge automatically infers the entities and relationships in the underlying database schema. In our book club example, developers can graphically build forms for entering speakers and events, and a view for displaying and editing the speakers for each event. As the pages are being built, AppForge automatically generate two

entities, *Speaker* and *Event*, and a *Presentation* relationship between the two entities. The key technical contribution here is an algorithm that translates a sequence of developers' actions based on only *two* simple context-dependent graphical primitives into complex schemas in the E-R model. We also prove that this algorithm is capable of generating a large class of E-R models, including those with entities, n-way relationships and aggregations.

3. **AppForge allows non-programmers to create complex views over multiple related entities.** AppForge provides a new navigation paradigm over (automatically generated) E-R models called a Schema Navigation Menu. This menu enables developers to visualize a complex E-R graph as a hierarchical menu and create views with sophisticated operators such as joins, aggregations and selections, without having to understand the details of these operators. The key technical contribution here is an algorithm that generates a Schema Navigation Menu from an arbitrary E-R graph, and then translates developers actions on this menu to sophisticated view definitions. We also prove that this algorithm can generate a large class of nested relational algebra [2] views, including those with primary-foreign key joins and nested structures.

We have used AppForge to prototype a wide range of applications such as a book club event planning system, a recruiting management system, an online course management system and an item trading system. We have also conducted a small user study to test the usability of AppForge. Based on results of this study, we have identified and fixed some of the main issues that confuse developers and also identified directions for future exploration.

1.4 Hosting A Large Number of User-Created "Small" Applications

While AppForge provides an intuitive interface allowing non-technical users to create applications, hosting those applications becomes a new challenge to the current systems. Even though each application often has relatively small data sizes and throughput requirements in contrast to large enterprise applications, since there can be a large number (say tens of thousands) of such applications in a large social network, building a data platform for this setting is not an easy task. Specifically, the combined data size and workload of the set of applications is quite large, of the order of peta bytes of data and millions of concurrent user sessions.

Unfortunately, existing data management solutions are ill-suited to handle a large number of small applications for one or more of the following reasons:

Cost: Commercial database systems (e.g., Oracle Real Application Cluster [23], DB2 Enterprise [9], and SQL Server [93]) have leveraged decades of research in the data management community and have achieved impressive scalability with respect to database sizes and throughput. However, this scalability comes at a large monetary cost. Large installations of such software are expensive and require complex management, which further adds to the monetary cost. Open-source alteratives to commercial database systems are less expensive (free!), but do not scale well because they do not have the sophisticated scalability features that are built into most commercial systems. In fact, part of the reason that commercial systems charge a premium for large installations is that the technology

for scaling is complex and difficult to get right.

Lack of sophisticated data management features: There has been a lot of recent interest in peer-to-peer (P2P) technologies such as Distributed Hash Tables (DHTs) [88, 89, 95] and ordered tables [1, 29, 65]. Such systems are designed to scale using commodity hardware and software (low cost) to a large number of nodes, thereby achieving excellent scalability and throughput performance. However, these systems only support very simple data management operations which essentially translate to equality and range lookups on a single table, which are inadequate for most Web applications. Further, such systems lack sophisticated transaction support, which is again crucial for Web applications. While there have been some notable attempts to incorporate more sophisticated data management features such as joins in P2P systems [1, 61], such systems still lack sophisticated transaction support such as ACID transactions, which are difficult to achieve at such scales.

There have also been emerging data platforms for Web applications such as BigTable [40], PNUTS [57] and SimpleDB [94]. These data platforms aim to handle large scale of data and operations for Web applications. However, in order to achieve large scale, these Web data platforms trade off consistency and richness of query-processing functionality. In particular, none of these system support ACID transactions across multiple records and tables, and they restrict the kinds of queries applications can issue. The set of applications we are targeting, small but full-functional applications on the Web, need consistency and richness of query-processing functionality. Therefore, many of the emerging Web data platforms are not suitable for hosting a large number of small applications. In fact, many of these platforms are specifically designed for applications

that need access to a large amount of data (that does not fit into a small number of machines, let alone a single machine).

Lack of multi-tenancy support: Most scalable data management systems are targeted towards scaling one or a few large databases, and do not provide explicit multi-tenancy support for multiple applications running on shared resources. While it is easy to control the Service Level Agremeements (SLAs) for one or a few databases by explicitly adding resources, it becomes a complex manageability problem when we try to meet the SLAs for many thousands of applications using *shared* resources (using dedicated resources is not a cost-effective option because of the large number of applications).

Our goal is to design a low-cost, full-featured, multi-tenancy-capable data-management solution that can scale to tens of thousands of applications. While this is a difficult problem in general, we exploit the fact that the applications are "small" and can comfortably fit (with possibly other applications) in a single machine without violating SLAs. This property enables us to design a low-cost data-management solution that meets our design objectives.

The architecture of our system is a set of database clusters. Each database cluster consists of a set of database machines (typically 10 of such machines) that are managed by a fault tolerant controller. Each database machine is based on commodity hardware and runs an instance of an off-the-shelf single-node DBMS. Each DBMS instance hosts one or more applications, and processes queries issued by the controllers without contacting any other DBMS instance. This architecture is clearly low-cost because it only uses a single-node DBMS (we use free MySQL in our prototype) as the building block. It also supports

sophisticated data management functionality, including joins and ACID transactions, because queries and updates are processed using a full-featured DBMS instance.

Given the above architecture, there are two main technical challenges that arise. The first is that of fault-tolerance: when a database machine fails, we need to ensure that all the applications running on that machine can be recovered without violating the ACID semantics of transactions. One of the technical contributions here is a provably correct way of managing database replicas using commodity data-management software, while still ensuring ACID semantics. The second challenge is ensuring that the SLAs of applications are met, especially when multiple application databases reside on the same machine. Another contribution is formalizing the notion of database SLAs, mapping these to measurable parameters on commodity data-management software, and formulating and solving an optimization problem that minimizes the required number of resources.

We have implemented and evaluated a prototype system based on the above architecture, deployed it on a cluster of machines, and evaluated the system using multiple TPC-W database applications. Our preliminary results show that the proposed techniques are scalable and efficient.

1.5 Organization of the Dissertation

The rest of the dissertation is organized as follows:

In Chapter 2, we present the details of the HILDA language, using a course

management application as a running example.

In Chapter 3, we describe how we can automatically partition a data-driven web application dynamically between client and server tiers in a way that is completely transparent to the developer.

In Chapter 4, we describe the AppForge system that is designed to allow non-technical users without much database and programming expertise to build fairly complex data-driven web applications.

In Chapter 5, we describe the design and implementation of our data management platform that can host a large number of small applications.

We give our conclusions and discuss future work in Chapter 6 and related work in Chapter 7.

CHAPTER 2

HILDA: HIGH LEVEL DECLARATIVE LANGUAGE FOR DATA-DRIVEN WEB APPLICATIONS

In this chapter, we present the details of the HILDA language, which is designed specifically to simplify the development of data-driven applications. We first describe a course management application that we use as a running example in Chapter 2.1, then in Chapter 2.2, we describe basic components in HILDA.

2.1 Case Study: A Course Management System

We first illustrate some of the shortcomings of existing application development platforms with CMS [36] – a course management system application. We developed CMS at Cornell to simplify the management of large courses. Figure 2.1 depicts the functions supported by CMS. CMS is currently being used by over 2000 students in 40 courses in computer science, physics, economics and engineering. CMS uses a standard three-tier architecture, with a back-end database server, middle-tier application servers and front-tier client browsers. The initial version of CMS was developed using PHP, while the current version was developed using J2EE.

We use four features of CMS to highlight some of the limitations of existing development platforms. Since the issues are similar for both versions of CMS, we focus on the J2EE version. As noted in Chapter 1.1 and Chapter 7, other development platforms suffer from similar shortcomings.

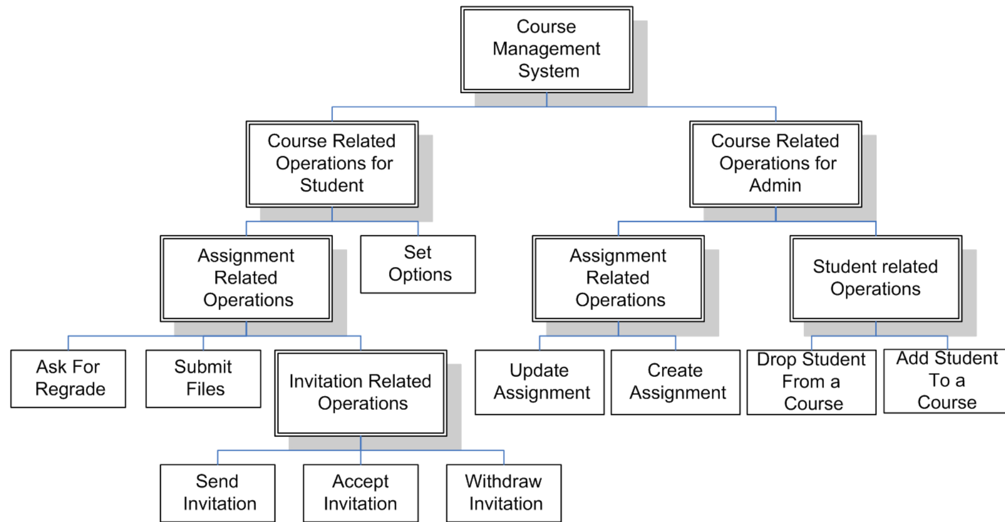

Figure 2.1: Course Management System

2.1.1 Assignment Creation

In CMS, the admin of a course can create an assignment for the course by specifying the name of the assignment, the release date, the due date and the set of problems etc.

Impedance mismatch: During assignment creation, the user input is obtained and temporarily stored using HTML forms in the web browser. When the user submits the assignment for creation, this input data is copied into the corresponding assignment Java Bean in the application server. While tools such as Struts [32] simply the mapping between forms and Java Beans to a certain extent, a lot of low-level code is still required to map between the different data models.

Mixing Application Logic and Presentation: When a user is in the process of creating an assignment, CMS performs some application-level sanity checks such as determining whether the due date of an assignment occurs after the release date.

Normally, such checks are performed in the web browser (using, say, JavaScript) so that the user obtains an immediate response, without the overhead of contacting the server. However, this causes application logic to be mixed with presentation (in the web browser), which makes applications harder to understand and maintain. Note that design patterns such as Model-View-Controller [37, 32, 39] do not help here because they are server-side solutions.

2.1.2 Viewing Student Grades

CMS allows students and staff to view relevant grades.

Impedance Mismatch: The student, course, and grade data are stored in database as separate tables and are exposed to application developers as corresponding Java Beans. However, for performance reasons, application developers have to directly work with relational tables to produce a list of students and their grades. Specifically, since each course, student, and grade is represented as a separate Java Bean object, in order to compute the grade for each student in the course, a join operation has to be performed in Java. It is far more efficient to issue a single SQL query to compute this information because the database can then optimize this query. Consequently, application developers must manually bridge the gap between the J2EE and relational data models and issue SQL queries.

2.1.3 Student Group Management

CMS allows students to form groups for a given assignment in a course. A student can initiate group creation by extending an invitation to another student. The other student can either accept the invitation (in which case a new group is formed) or decline the invitation. A student can also withdraw an outstanding invitation and groups can be disbanded at any time.

No support for conflict detection: When a student issues a request to accept or decline an invitation, CMS needs to guard against possible conflicting actions such as the inviting student withdrawing the invitation. In addition, there are a variety of other cases unrelated to group management where the action should not be performed, including if the student is dropped from the course (by the course administrator), if the inviting student is dropped from the course, if the assignment has been dropped, if the course itself has been dropped, and so on. Using current programming paradigms and tools, it is very difficult for application developers to correctly identify *all* conditions that need to be checked before performing a specific task such as accepting or declining an invitation. Database constraints cannot be used to solve this problem, because application-level constraints often do not translate directly into database constraints. For instance, dropping an assignment in CMS does not delete the assignment but only sets a "hidden" flag for that assignment (so that it can be resurrected if necessary). Thus, the database will not identify the invitation accepts and declines for a dropped assignment as a violation of database constraints.

Not declarative: Even if the application developer were to correctly identify the correct precondition for performing an action, he or she would have to make an *a priori* decision about how to enforce the condition. For example, the appli-

cation developer could decide to hold transaction locks for the entire duration of the user input and action, or alternatively, check the precondition just before performing the user action. However, since this precondition cannot be specified declaratively, the system cannot dynamically optimize for the preferred strategy given the current workload, nor can it explore other possibly more efficient strategies such as using triggers to invalidate actions.

2.1.4 Web Site Structure

No structured programming for websites: CMS supports a rich navigational interface whereby various pages (such as the course overview page) can be reached through multiple paths. While this interface is intuitive for the user, it is very difficult for the application developer to understand the "control flow" between different pieces of application logic spread over interconnected pages. Programming the structure of the web site is reminiscent of programming with goto statements, which make programs difficult to understand and maintain. What is missing is a more "structured programming" paradigm for websites, which nevertheless provides the same rich navigational interface for end-users.

2.2 The HILDA Language

HILDA is designed to address the above problems. We begin by motivating some of our design decisions.

2.2.1 Design Decisions

First, HILDA is based on UML [41], a well-accepted modeling framework. HILDA's main construct are **AUnits**, which correspond to UML classes. The local state of an AUnit corresponds to UML class attributes. As classes can have operations, AUnits can have $Activators$. With data and associated operations, the HILDA programming model is state-based in that a HILDA programmer specifies what operations are allowable in a given state of the program. The main difference from the traditional use of UML is that the object creation and operations are specified declaratively[1], which enables the HILDA compiler to automatically perform various optimizations without burdening the user with performance issues.

Second, HILDA uses a single data model - the relational model - to represent the state of all parts of the application, including the database, application logic and the client. This eliminates the impedance mismatch problem and also enables the application logic to be specified declaratively using SQL. The choice of the relational model also allows for a practical and efficient implementation since most existing database systems are relational.

Third, HILDA *logically* separates server and client state to enable highly concurrent execution. The server maintains the current state of the application, and each client sees a (possibly out-of-date) version of it locally. Whenever a client wants to perform an update operation, it checks with the server to see if this operation is still valid in the current system state (to avoid application conflicts). Notice that this separation between client and server state is only *conceptual*. The

[1] This is also the main reason we use different names for otherwise standard object-oriented concepts, so that declarative and non-declarative constructs are easily distinguished.

real separation can be different and should be done by the HILDA compiler or runtime environment based on certain optimization criterion, e.g., sanity checks can be pushed to client side to save bandwidth and round trip time.

Fourth, HILDA models the application logic and associated control flow as a hierarchy. This decision is based on our experience in developing data-driven web applications: since navigation can be very complex, and since the operations that a user can perform at any time depend on complex conditions that have to be satisfied by the current state of the user's session, we need a way to cleanly specify these preconditions. HILDA specifies preconditions hierarchically; this helps the programmer to think in high-level abstractions which are then further broken down into smaller steps further down in the hierarchy. HILDA's hierarchical structure also enables encapsulation as the hierarchy naturally limits the scope of the data access of an object. HILDA's control flow goes along the same hierarchy. It is like structured programming, with a tree-like execution structure. It is powerful enough to capture complex graph control flows, but makes the specification of operations more structured and confined to small parts of the code.

Fifth, HILDA uses inheritance to separate application logic from web site structure. Specifically, application developers can derive a web site AUnit by inheriting from the corresponding application logic AUnit. The use of inheritance for this purpose has two advantages: (1) the same structured programming model can be used for both application logic and web site structure, and (2) the same application logic can be reused for multiple web site structures.

Finally, HILDA provides a HTML-based presentation construct called a PUnit (Presentation Unit), which is associated with an AUnit and describes how

the content of the AUnit is to be presented. PUnits ensure a clear separation of application logic from presentation because they deal only with presentation issues like page layout, font size and background color, while AUnits deal only with application logic and web site structure.

In the remainder of this chapter we describe the HILDA language in detail. We start by overviewing HILDA's core construct, the AUnit, in Chapter 2.2.2, and we describe AUnits in detail in Chapter 2.2.3. We then discuss inheritance in Chapter 2.2.4 and PUnits are described in Chapter 2.2.5.

2.2.2 AUnits Overview

An AUnit is a single-entry single-exit programming construct that is associated with an (optional) *input schema* and an (optional) *output schema*. The input and output schemas are both relational schemas. Given an AUnit, one or more *instances* of the AUnit can be created. Each instance of an AUnit takes in an input conforming to the input schema of the AUnit and returns an output conforming to its output schema. The act of creating an instance of an AUnit is called *activation*, and the act of destroying an instance of an AUnit is called *deactivation*.

There are three types of AUnits: *Basic AUnits*, *User-Defined AUnits* and *External AUnits*. Basic AUnits are predefined by the system and provide functionality to interact with end users. For example, an instance of the *ShowRow* AUnit shows the attribute values of the input(a single row) to the user and returns no output. Similarly, an instance of the *GetRow* Basic AUnit returns a row of values entered by a user; it takes in no input and returns a single row as an output. Other Basic AUnits for other common interaction tasks are defined similarly.

A User-Defined AUnit corresponds to a functional component in the system. Just as components can have subcomponents, each instance of a User-Defined AUnit also contains zero or more instances of child (User-Defined or Basic) AUnits, which are called *child AUnit instances*. AUnits (like sub-components) can be reused in more than one place. The definition of a User-Defined AUnit contains the application logic of activating and deactivating child AUnit instances, preparing input for child AUnit instances, updating local state and processing output of child AUnit instances and it's own input and output schemas.

External AUnits are used to express small parts of the application logic that do not lend themselves to declarative specification. For example, if an application requires the use of a max-flow min-cut algorithm, it will be awkward to program this using SQL (even though it can theoretically be done with order-based functions and recursion in SQL'99). External AUnits support the same API as other AUnits, but are specified in an imperative language such as Java. Since most data manipulation can be specified declaratively, we expect only a small part of the code to be written using External AUnits; in fact, applications such as CMS do not need External AUnits at all.

One AUnit in the hilda program is designated as the *root AUnit*, which intuitively corresponds to the "main" function in a program. A new instance of the root AUnit is activated each time a new user connects to the HILDA application, and this instance is deactivated when the user disconnects.

CMS consists of the User-Defined AUnits pictorially depicted in Figure 2.2, which map directly to the functional components of CMS (Figure 2.1). The root AUnit is the CMSRoot AUnit.

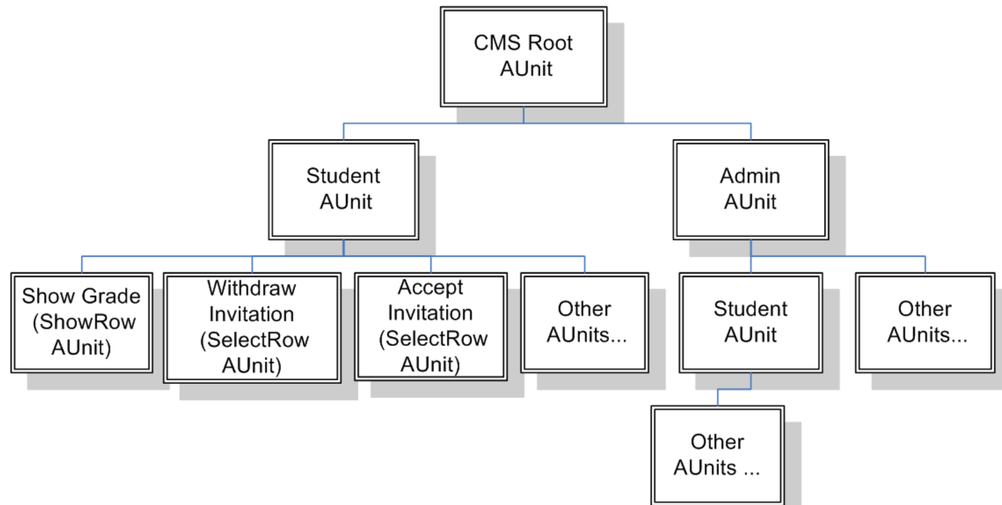

Figure 2.2: AUnits in CMS

2.2.3 User-Defined AUnits

Figure 2.3 shows the BNF grammar for a User-Defined AUnit. As shown, each AUnit has a name (line 2) and a number of other components which we discuss in the next few sections. We will use the code for some of the CMC AUnits (Figures 2.4, 2.5, and 2.9) to illustrate these components.

Schemas

A User-Defined AUnit has optional input and output schemas (Figure 2.3, lines 3-4). Here `Schema` is a non-terminal that describes a relational schema (the production rules for `Schema` are not shown). As a convenient shorthand, a AUnit can have an inout schema (line 5) when the same schema is used for both input and output. We use the notation in.X and out.X to refer to the input and output versions, respectively, of a table X in an inout schema.

An AUnit can also have a persistent schema (line 7). The data stored in a

```
01) AUnit ->
02)     AUnitName:STRING '{'
03)     ['input' 'schema' '{' Schema '}']
04)     ['output' 'schema' '{' Schema '}']
05)     ['inout' 'schema' '{' Schema '}']
06)
07)     ['persist' 'schema' '{' Schema '}']
08)     ['persist' 'query' '{' Assignment* '}']
09)
10)     ['local' 'schema' '{' Schema '}']
11)     ['local' 'query' '{' Assignment* '}']
12)
13)     Activator*
14)     '}'
15) Activator ->
16)     'activator' ActName:STRING : AUnitName:STRING '{'
17)     ['activation' 'schema' '{' Schema '}'
18)     'activation' 'query' '{' Query '}']
19)     ['input' 'query' '{' Assignment* '}']
20)     Handler*
21)     '}'
22) Handler ->
23)     [return] 'handler' HandlerName:STRING '{'
24)     ['condition' '{' Query '}' ]
25)     'action' '{' Assignment* '}'
26)     '}'
27) Assignment -> TableName:STRING ':-' Query
```

Figure 2.3: BNF grammar for a User-Defined AUnit

persistent schema has two important properties: (1) it is *persistent* across AUnit instance activations and deactivations, and (2) it is *shared* between different instances of the same AUnit. The data in the persistent schema is initialized by evaluating the persistent query the very first time the HILDA program is run (line 8). A persistent query is a set of `Assignment`s, each of which assigns the result of an SQL query to a table in the persistent schema (line 7). Here `Query` is a non-terminal that describes a SQL query (the production rules for `Query` are not shown). In addition to a persistent schema, an AUnit can have a local

schema (line 10). The data stored in the local schema is initialized when a new instance of an AUnit is activated, by evaluating a local query (line 11). The data stored in the local schema is private to a specific instance and is not shared between instances. When an AUnit instance is deactivated, the data in its local schema is destroyed. Another way to view them is that local schema captures session states that are private to each instance in a single user session and persistent schema represents the shared states that can be accessed and updated by multiple instances in difference sessions.

CMS Example: Consider the CMSRoot AUnit in Figure 2.4. CMSRoot has an input schema (line 2) that specifies the name of a user logged in to the system. CMSRoot does not have an output schema since it is the root AUnit. CMSRoot also has a persistent schema (lines 4-13) that describes the data that the course management system works with – courses, students, assignments, etc. The data stored in the persistent schema is shared among all CMSRoot instances, hence different users can access that data. Since CMSRoot does not have a persistent query, all the tables in the persistent schema are initially empty. CMSRoot does not have a local schema.

As another example, consider the CourseStudent AUnit in Figure 2.5. CourseStudent captures the application logic for a student in a course, who can view grades and manage his or her groups. CourseStudent takes in the student id, the set of course assignments (lines 2-4), and course group information (lines 5-8) as input, and returns the new course group information (lines 5-8) as output.

```
01 AUnit CMSRoot // Obtain the name of the user as input
02 input schema { user(name:string) }
03// Store information about admins, courses, students, etc. Initially, all tables are empty.
04 persist schema {
05   course(cid:int, cname:string)
06   staff(sid:int, cid:int, sname:string, role:string)
07   student(sid:int, cid:int, sname:string)
08   assign(aid:int, cid:int, name:string, rel:date, due:date)
09   problem(pid:int, aid:int,name:string,weight:float)
10   group(gid:int, aid:int)
11   groupmember(gmid:int, gid:int, sid:int, grade:float)
12   invitation(iid:int, gid:int, invitersid:int, inviteesid:int)
13 }
14 // Activator to activate a student AUnit for each course. Each student can place, withdraw, accept invitations
from other students to form a group
15 activator ActCourseStudent : CourseStudent {
16    activation schema { acourse(cid:integer) }
17    activation query {
18      SELECT C.cid
19      FROM course C, student S, user U
20      WHERE C.cid = S.cid AND S.sname = U.name
21   }
22   // Prepare the assignments corresponding to the course
23   input query {
24      Student.invitation :-
25      SELECT G.*
26      FROM assign A, group G, invitation I, Student S, user U
27      WHERE A.cid = activationTuple.cid AND A.aid=G.aid AND G.gid=I.gid
28         AND S.sname=U.name AND (I.invitersid=S.sid OR I.inviteesid=S.sid)
29    ...
30   }
31   handler UpdateInv {
     action{
32         //update assignment
33         invitation :-
34            SELECT *
35            FROM invitation I
36            WHERE I.iid not in
37            (SELECT * FROM Student.in.invitation)
38            UNION
39            SELECT * FROM Student.out.invitation
40      }
41   }
42 }
43 ... (similarly for course staff, system admin, etc.)
```

Figure 2.4: The CMSRoot AUnit

```
01 AUnit CourseStudent
02 input schema {
03   curstudent(sid:int)
04   assign(aid:int, name:string, release:date, due:date)}
05 inout schema {
06  group(gid:int, aid:int)
07  groupmember(gmid:int,gid:int,sid:int,grade:float)
08  invitation(iid:int,gid:int,invitersid:int,inviteesid:int)}
09 // Show the student's grades for each assignment
10 activator ActShowGrades : ShowRow(string,float) {
11   activation schema {
12     agrade(aid:int,assignname:string,grade:int) }
13   activation query {
14     SELECT A.aid, A.name, GM.grade
15     FROM groupmember GM, student S, assign A LEFT OUTER JOIN Group G ON A.aid = G.aid
16     WHERE G.gid = GM.gid and GM.sid = S.sid}
17   input query{
18     ShowTable.input :- SELECT activationTuple.assignname, activationTuple.grade}
19 }
20 // Withdraw an invitation
21 activator ActWithdrawInv : SelectRow(int,int) {
22   activation schema {
23     aassign(iid:int,inviteesid:int) }
24   activation query {
25         SELECT I.iid, I.inviteesid FROM invitation I, curstudent WHERE I.invitersid = S.sid}
26   input query {
27      SelectRow.input :- SELECT activationTuple.iid, activationTuple.inviteesid}
28   return handler {
29     //delete the invitation we withdrew
30     invitation :- SELECT * FROM invitation I, SelectRow.output O WHERE I.iid <> O.iid}
31 }
32 // Accept an invitation
33 activator ActAcceptInv : SelectRow(int,int) {
34   activation schema {
35     aassign(iid:int,invitersid:int) }
36   activation query {
37         SELECT I.iid, I.invitersid FROM invitation I, curstudent S WHERE I.inviteesid = S.sid}
38   input query {
39      SelectRow.input :- SELECT activationTuple.iid, activationTuple.invitersid}
40   return handler {
41     //delete the invitation accepted
42     ...
43     //update group, groupmember tables
44     ...}
45 }
... (place, decline invitations, etc.)
```

Figure 2.5: Student AUnit.

Activators: Overview

Continuing our discussion of the grammar in Figure 2.3, AUnits can have zero or more activators (line 13). Activators are used to control (1) how child AUnit instances are activated, (2) how a return of a child AUnit instance is processed, and (3) how child AUnits are reactivated after a child AUnit return has been processed. These three tasks correspond to the *activation phase*, the *return phase*, and the *reactivation phase*, respectively. The activation and reactivation of child AUnit instances is specified declaratively using an "activation query" (to be described soon), which enables HILDA to automatically detect application level conflicts. The return of child AUnit instances are also processed declaratively using SQL-based "handlers". We now describe the three phases.

Activators: Activation Phase

As shown in Figure 2.3, each activator contained in an AUnit has a name, `ActName`, which is unique within the scope of the containing AUnit. Each activator also specifies the name of the child AUnit, `AUnitName`, whose instances it activates (line 16). Each Activator also has an activation schema (line 17) and an activation query (line 18). An activation schema is a relational schema that contains exactly one table (the table can contain any number of columns). The activation query produces a set of tuples that conform to the activation schema; the activation query can refer to the tables in the containing AUnit's input schema, local schema and persistent schema. Whenever an instance of an AUnit is activated, each activator contained in the AUnit is processed as follows: for *each* tuple produced by the activation query, a child AUnit instance is activated. This enables an activator to activate multiple child AUnit instances.

Each activator also has an (optional) input query (line 19), which is used to compute the input for each activated child AUnit instance. The input query can refer to the tables in its containing AUnit's input schema, local schema, and persistent schema. In addition, the input query can refer to a special table called `activationTuple`. The activationTuple table has the same schema as the activation schema. Consider a child AUnit instance X that is activated since there exists a tuple x in the activation schema. The activationTuple table for that child AUnit contains exactly x. Thus, the contents of the activationTuple table are different for each child AUnit, so activationTuple can be used to tailor the input for a given child AUnit instance based on its associated tuple in the activation query.

Note that the above process, whereby an AUnit instance recursively activates child AUnit instances, creates a tree of active AUnit instances, with the root of the tree being an instance of the root AUnit. We refer to this tree as an *activation tree* and use the term *parent AUnit instance* to denote the parent of an AUnit instance in the activation tree. We refer to the set of activation trees corresponding to all active root AUnit instances as the *activation forest*.

CMS Example: When a user first connects to CMS, a new user session is created by activating a new instance of CMSRoot, the root AUnit. Figure 2.6 Session 1, shows the activation tree of a new instance of CMSRoot. When a new instance of CMSRoot is activated, CMSRoot uses its activators – ActCourseStudent (lines 15-42) and other activators (not shown) – to activate child AUnit instances. The ActCourseStudent activator is used to activate instances of the CourseStudent AUnit (line 15) for each course for which the current user is a student. This activation is controlled by the activation schema (line 16), which contains the ids of

the relevant courses, and the activation query (lines 17-21), which produces the ids of all courses for which the current user is an administrator. The input query (lines 23-30) produces the input (i.e., information about the student groups) for each activated CourseStudent instance.

Each activated CourseStudent AUnit (Figure 2.5) instance recursively activates child AUnit instances using its activators: ActShowGrades (lines 10-19), which shows the student grades, ActWithdrawInv (lines 21-31), which allows the student to withdraw outstanding invitations, and ActAcceptInv (lines 33-45), which allows the student to accept invitations. Again, the activation query associated with these handlers declaratively specifies the condition under which the child AUnit instances are to be activated.

Figure 2.6 shows the activation forest that results when two students connect to CMS in separate sessions. In Session 1, the set of course ids for which the current user is a student is $\{10, 11\}$ (this information is computed from the data in the persistent tables, part of which are shown in the figure). For *each* of these course ids, a new instance of the `CourseStudent` AUnit is activated. Each `CourseStudent` AUnit instance recursively activates child AUnit instances for displaying grades, accepting invitations and withdrawing invitations, as shown in the figure. The activation phase for Session 2 is similar. Note that the different instances of CMSRoot share the same persistent schema (by definition of the scope of persistent schemas).

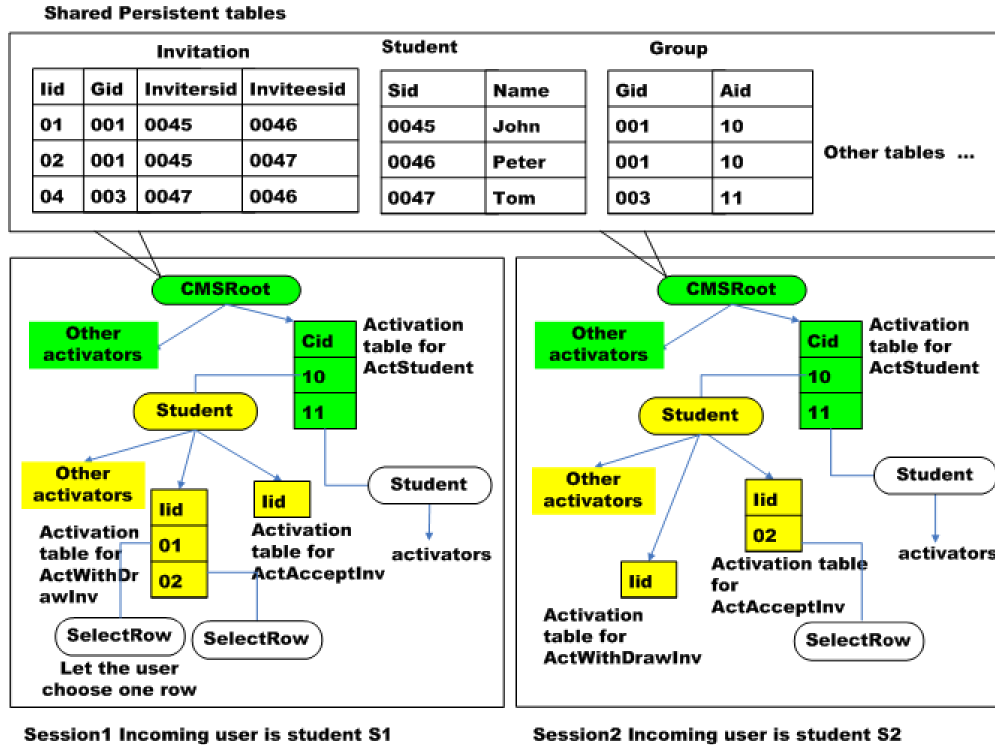

Figure 2.6: Activation Phase

Activators: Return Phase

The return phase is initiated when a Basic AUnit instance returns. Since Basic AUnits deal with Input/Output functions, the return phase is typically initiated by a user action such as selecting a row. When a Basic AUnit instance returns, its output is processed by an activator *handler*. The handler can perform certain actions and can (optionally) cause the parent AUnit instance (of the returning AUnit instance) to return, recursively. After all returns have been processed, the return phase ends and the system transitions to the re-activation phase (discussed in the next section).

Returning to Figure 2.3, the return of a child AUnit instance is processed by zero or more `Handlers` in the activator that activated the child AUnit instance (Figure 2.3, line 20). Each handler has a name, `HandlerName` (line 24), which

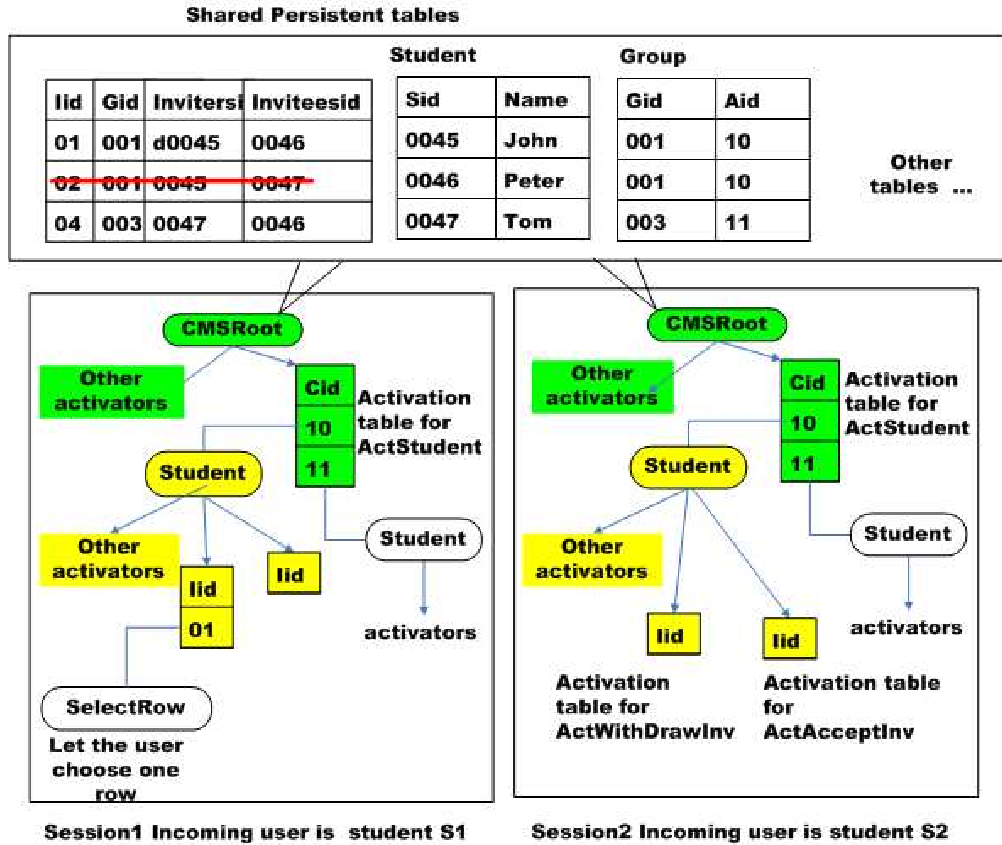

Figure 2.7: Reactivation Phase

is unique within the scope of the containing activator. Each handler also has an (optional) condition (line 25) and an action (line 26). Whenever a child AUnit instance returns, the conditions of all the handlers contained in the activator are checked. One of the handlers whose condition evaluates to true is non-deterministically chosen and its action is performed. Then, if the handler has the keyword `return` (line 24), the enclosing AUnit also returns and its return is recursively processed. If the handler does not have the keyword `return`, then the system enters the re-activation phase. The action of a handler is specified as an `Assignment`. The queries in the `Assignment` can refer to the same tables as the query in the condition. The action of a return handler can modify only the tables in the persistent schema and output schema of the containing AUnit. The

action of a non-return handler can modify only the tables in the local schema and persistent schema of the containing AUnit.

CMS Example: Consider the activation forest in Figure 2.6 and assume that the user in Session 1 wishes to withdraw an invitation and causes the Basic AUnit instance with ID 20 to return. This return will be processed by the appropriate handlers (Figure 2.5, lines 28-30, Figure 2.4, lines 34-37), which will cause the invitation to be removed from the persistent schema.

Activators: Reactivation Phase

As described above, during the return phase, the activator handlers along the branch of the activation tree that returns can change the contents of the local and persistent schemas. Consequently, the activation forest has to be "reactivated" so that it is consistent with the new schema contents. The reactivation phase is similar to the activation phase, with special semantics to deal with local schema values and with concurrent user actions. Specifically, suppose AUnit instance a did not return and its activation tuple (in the parent AUnit instance) remains present during reactivation, then instance a is said to be *preserved* across the reactivation and its local schema remains unchanged. The intuitive reason is that an AUnit instance should not lose its temporary state as long as its activation is not affected by the return of a different AUnit instance.

For concurrent user actions, HILDA guarantees a correctness notion analogous to database serializability: the resulting activation forest and user output are *as though* the actions were performed in some serial order. A subtle issue arises here: although two (or more) user actions may be valid in a given acti-

vation forest, only one of them may be valid in any serial processing of these actions. For example, consider a student A who has invited a student B to join his group. Two actions are possible in this activation forest: A can withdraw the invitation to B, or B can accept A's invitation. However, if both these actions are submitted concurrently, only one of them can complete successfully (since either of them invalidates the other). A similar situation occurs if A withdraws the invitation to B, but B has still not refreshed her page and tries to accept the invitation.

One of the advantages of HILDA is that it can *automatically* detect such application-level conflicts. The key to detecting such conflicts lies in using the activator conditions of AUnit instances. If an AUnit instance is deactivated (due to an update that causes its activator condition to be false), then pending actions on the AUnit instance cannot be performed since the AUnit is not preserved during reactivation. For instance, after Student 1 withdraws the invitation to Student 2 in Figure 2.6, the activation forest is updated appropriately (Figure 2.7) so that Student 2 can no longer accept the invitation from Student 1.

We note that the above semantics of HILDA relaxes the traditional notion of serializability. Two Basic AUnit instance returns (transactions) are said to conflict iff one Basic AUnit instance return violates the activator condition of the other Basic AUnit instance (or any of its ancestors). HILDA's notion of correctness thus specifies conflicts in terms of application-level conditions (which are automatically inferred from activator conditions) and can be viewed as a specific extended transaction model [22, 56, 74, 103] that is tailored to data-driven web applications. Note that the processing of the activation-return-reactivation phases of user actions are still fully serializable since the actions are (logically)

performed one after the other; the application-level conditions are used only to check whether a user action is still valid after updates to the activation forest.

2.2.4 AUnits: Inheritance

Like conventional object-oriented languages, HILDA supports a notion of *inheritance* for extending the functionality of AUnits. HILDA inheritance can be used to add new application logic and also (as we shall see) to specify the structure of an application web site. We use the term *extended AUnit* to refer to an AUnit that uses inheritance, and we use the term *base AUnit* to refer to the AUnit from which an extended AUnit inherits. An extended AUnit inherits all the schemas from its base AUnit. An extended AUnit also inherits all the activators from the base AUnit. In addition, an extended AUnit can add new activators and extend existing activators in the base AUnit. An activator in a base AUnit can be extended in two ways: (1) by adding new handlers, and (2) by filtering the set of activation tuples so that only a subset of the child AUnit instances are activated. The filtering of the set of activation tuples is specified as a query that returns a non-empty set iff the current activation tuples corresponds to a child AUnit instance that should be activated. Such filtering is usually used to structure the web site by selecting the child AUnit instance that should be presented to a user at a given time.

CMS Example: Consider the NavCMS AUnit in Figure 2.9. NavCMS inherits from CMSRoot and structures it as a web site that only shows the currently active course selected by the user (recall that CMSRoot activates *all* relevant courses). NavCMS adds this new functionality by defining its own local schema

```
01) ExtendedAUnit ->
02)   'AUnit' AUnitName:STRING
03)      'extends' BaseAUnitName:STRING '{'
04)
05)   ['input' 'schema' '{' Schema '}']
06)   ['output' 'schema' '{' Schema '}']
07)   ['inout' 'schema' '{' Schema '}']
08)
09)   ['persist' 'schema' '{' Schema '}']
10)   ['persist' 'query' '{' Assignment* '}']
11)
12)   ['local' 'schema' '{' Schema '}']
13)   ['local' 'query' '{' Assignment* '}']
14)
15)   (Activator | ExtendedActivator)* '}'
16)
17) ExtendedActivator ->
18)   'extend' 'activator' BaseActName:STRING '{'
19)   ['filter' 'activation' '{' Query '}']
20)   Handler* '}'
```

Figure 2.8: Grammar for AUnit Inheritance.

to store information about the currently active course (line 2). It also defines a new activation handler to get user input on the current active course (lines 4-11). In addition, it extends the ActCourseStudent activator (Figure 2.4, line 18) in CMSRoot so that the CourseStudent child AUnit is only activated for the currently active course; this condition is specified in the activation filter query (Figure 2.9, lines 13-17), which returns a non-empty result only for the current active course.

2.2.5 PUnits

AUnits use a unified model to describe the application logic and the structure of the application website. However, AUnits do not specify presentation details,

```
01 Aunit NavCMS extends CMSRoot
// Keeps track of the currently active course
02 local schema { currcourse(cid:integer) }
03 //Allows user to select from list of courses
04 activator ActSelectCourse : SelectRow(integer,string){
05    input query {
06       SelectRow.input :- SELECT * FROM course
07    }
08    handler {
09       currcourse :- SELECT O.1 FROM SelectRow.output
10    }
11 }
12 activator extending ActCourseStudent {
13    filter activation {
14       SELECT *
15       FROM currcourse CC
16       WHERE activationTuple.cid = CC.cid
17    }
18 }
19 ... (similarly for showing admin courses, etc.)
```

Figure 2.9: NavCMS inherits from CMS

```
punit ShowNavCMS for NavCMS {
    <body bgcolor="yellow">
        <hr>
        <punit activator=''ActSelectRow''
               name=''ShowSelectRow''>
        <hr>
        <punit activator=''ActCourseAdmin''
               name=''ShowCourseAdmin''>
        <hr>
        ...
    </body>
}
```

Figure 2.10: PUnit example

such as background colors and the page layout in the web browser. In HILDA, such details are specified using PUnits (for presentation units). This enforces the separation of application logic from presentation.

HILDA associates one or more Basic PUnits with each Basic AUnit. Each Basic PUnit describes how a Basic AUnit is to be displayed. For example, the SelectRow Basic AUnit can have one or more Basic PUnits that specify how SelectRow is to be presented to the user (e.g., as form entries or pull-down menus).

HILDA allows users to develop User-Defined PUnits. Each User-Defined PUnit is associated with a User-Defined AUnit and has embedded HTML code that generates part of the HTML page corresponding to that AUnit. Since each web page is composed of a hierarchical tree structure of nested AUnit instances, the User-Defined PUnits associated with User-Defined AUnits can render their part of the page, and recursively invoke the PUnits associated with the child AUnits to build the remaining part of the HTML page. This idea of recursively building up presentation units is similar to the technique proposed in [38].

CMS Example: An example User-Defined PUnit specification for CMS is given in Figure 2.10. The ShowNavCMS PUnit is associated with the NavCMS AUnit. The PUnit has embedded HTML code to set the page background and draw horizontal lines (<hr>) on the page. In addition, it uses the <punit> tag to invoke other PUnits – ShowSelectRow, ShowCourseStudent – to build up the HTML page. In this example, ShowSelectRow is the PUnit associated with the SelectRow AUnit, which is invoked by the ActSelectRow activator of NavCMS. ShowCourseStudent is similarly associated with the CourseStudent AUnit, which is invoked by the ActCourseStudent activator.

2.3 Discussion

We note that HILDA is not a general-purpose programming language for developing arbitrary applications; rather, it is specifically designed for developing data-driven web applications. HILDA achieves this goal by tightly integrating with the data model and declarative query language of the underlying database system. While we have used the relational model and SQL,HILDA could be extended to use other data models and associated query languages, e.g., XML and XQuery.

As with all new programming languages, we expect that programmers learning to program in HILDA will have a learning curve. However, the fact that HILDA has only a few simple constructs and offers many potential benefits might help ease this transition. It is an open question at this point as to whether the potential benefits of HILDA offset the overhead of switching to a new language.

In next chapter, we will focus on the runtime system, which exploits the declarative nature of HILDA language specifications to generate high-performance application code.

CHAPTER 3

AUTOMATIC CLIENT-SERVER PARTITIONING FOR DATA-DRIVEN WEB APPLICATIONS

In this chapter, we describe how we can automatically partition a HILDA program into client-server logic during runtime based on performance criterion. We first show an example of such partitioning in Chapter 3.1. Then we will describe details of the run-time environment in Chapter 3.2. We model the client-server tier partitioning as an optimization problem and provide solutions in Chapter 3.3. We show how we can use trace data to instantiate the optimization model and show the efficacy of our techniques in a thorough experimental evaluation using a technical benchmark and a real application in Chapter 3.4.

3.1 Partitioning Example

The right part of Figure 3.1 shows the web page for faculty to edit staff list in a course by a faculty. The underlying activation tree is shown to its left. Each activated AUnit instance corresponds to a sub-page. NavigationBar_1 corresponds

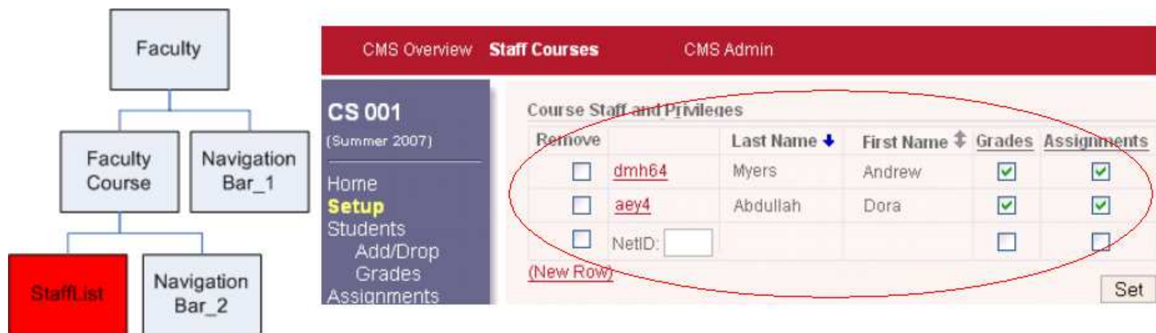

Figure 3.1: Web Page for edit staff and the underlying activation tree

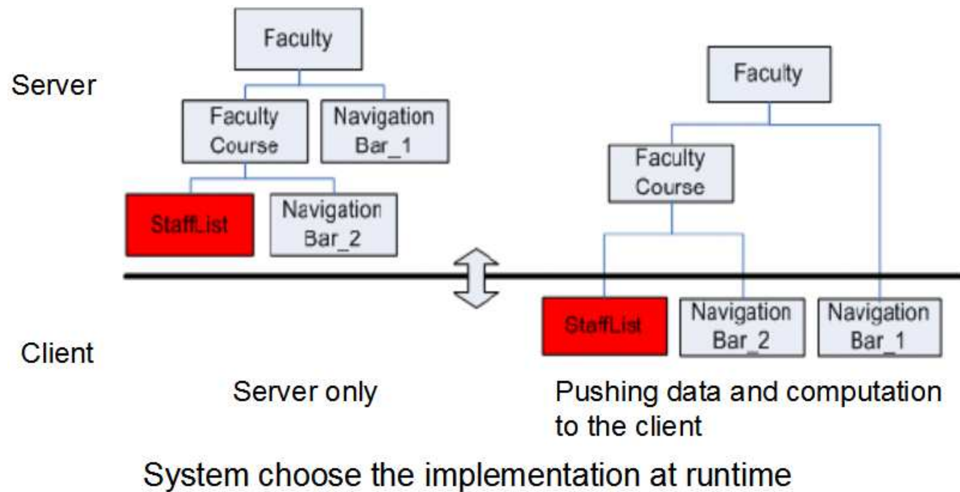

Figure 3.2: Activation tree with different partition strategies

to the navigation bar at the top of the page, and NavigationBar_2 corresponds to the one at the left. StaffList corresponds to the sub-page that allows a faculty to edit staff in the course. Figure 3.2, shows two different partitions of the activation tree. In the first case, we keep the complete activation tree at the server side, while in the second case, we keep part of it at the client side. The main drawback of running everything at the server side is that the client must contact the server for every operation the user performs. The server needs to resend the entire refreshed page in HTML format to the client. Instead, while the navigation bars and *Stafflist* instances are maintained at the client, the run time system can cache the data needed by them. Then, if the user adds or removes staff, the list of staff members is updated locally in the client, and the server is contacted only when the *Submit* button is clicked by the user. Only after this step are the updates in the staff list sent to the server, which updates the database. Maintaining AUnit instances at the client can therefore result in better system response time and a better user experience. This can also be seen from our experiments, discussed in Chapter 5.4. Similar caching logic for partial updating of pages can

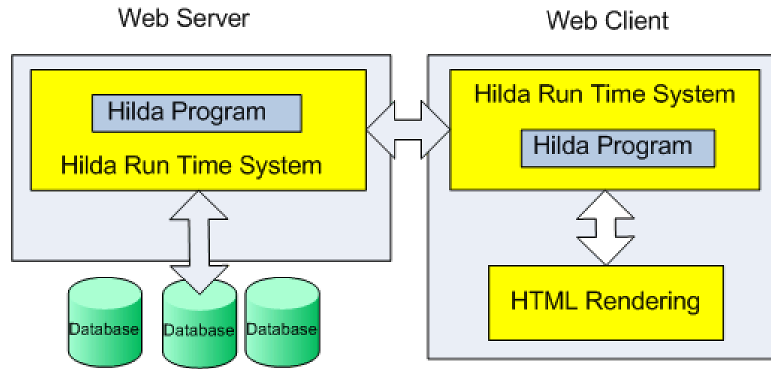

Figure 3.3: System Architecture

be implemented in frameworks like AJAX only by extensive client side coding. In our framework, such partitions are enabled by the runtime system of HIDLA and chosen automatic based on quantitative performance criterion.

3.2 The HILDA Runtime System

The HILDA runtime systems, on both the server and the client, are evaluation engines for HILDA programs. They execute the application logic specified in a program by maintaining the activation trees, and maintain consistency between the client and server states. We use RTSS to refer to the run time system residing at the server, and RTSC for the system residing at the client. The RTSS is a Java servlet running in some application server (such as JBOSS or Weblogic), and has connections to the back end database. It communicates with the RTSC. The RTSC runs as a "sticky" applet, which resides in the secondary cache of the client and is available for quick loading by browsers [60].

Based on the semantics of HILDA, the RTSS and RTSC coordinate with each other to maintain the activation tree. An AUnit instance is activated when its activation condition is satisfied and is deactivated when the conditions fails to

be satisfied. The location (client or server) where an instance is activated is not predefined by the application developer; instead, it is determined and can be changed at run time by the run time system.

The RTSC caches local data at the client. It uses this data to generate webpages dynamically (e.g., student info list), and to store a user's temporary input (e.g., items in a shopping cart). This temporary data is stored in main memory, and reused by the RTSC. The RTSC contacts the RTSS to check for updates to its input data. The system imposes an upper bound on how out-of-date the client state can be by periodically contacting the server using *heartbeat* messages. To avoid sending the cached data back and forth between the client and the server, the RTSS maintains a copy of the data sent to each client. On receiving an update request, the server checks for updates to the client input data, and responds with only the updated data. To limit the amount of server-side data required for each client session, the developer can specify a maximum life span for the data in the server cache, as well as the heartbeat frequency of the client. Our cache consistency strategy is similar to a detection based approach for transactional client server caching [51, 108, 80], although the system allows for the integration of other strategies in the future.

Client-server partitioning is done based on the *activation profile* of an application. The activation profile specifies which AUnit instances, identified by their unique key, should be activated in the client. When the run time system activates an instance of some AUnit, it refers to the configuration profile to determine whether the RTSS or the RTSC should activate the instance. The activation profile is generated automatically, based on the observed workload of the system. We will discuss its generation in the next section.

3.3 Model of Client-Server Partitioning

In this section, we present a cost model for client-server partitioning. We first define the problem and formulate it as an optimization problem. We show that the problem is NP-hard. We then give an algorithm that approximates the optimum partition, and prove a bound on the approximation error.

3.3.1 Partitioning Philosophy

A plausible method of solving the client-server partitioning problem would be partition at the granularity of AUnit definitions; i.e., to partition the set of AUnit definitions into two sets, one corresponding to AUnits whose instances will run on the server, and the other corresponding to AUnits whose instances will run on the client. This method would not capture the fact that different instances of the same AUnit may require very different amounts of computation and data transfer. For example, in the Course Management System, the EditCourse AUnit provides the functionality for course staff to edit courses. The amount course-related data, such as the number of students enrolled, the number of assignments, etc., can differ substantially between courses. We may want to ship the data and computation to the client for small courses while keeping big courses at the server side to save bandwidth. This motivates our decision to group similar instances together, profile their execution and then partition programs at the level of AUnit instances based on the profiles.

Different types of clients can have very different computing and storage resources. For example, moving computing and data to a powerful desktop client

may be desirable, while doing the same for a PDA client may adversely affect its response time. This motivates us to partition the application based on the types of clients. We make the assumption that the cost associated with a partition is independent of the load on the server, so that partitions corresponding to different clients do not interfere with each other's performance. Essentially, we are assuming that server load can be managed using existing load-balancing techniques; improving server system scalability is outside the scope of the thesis. The solutions for each client type thus obtained can be combined to yield an overall optimal solution for the application. Therefore, we describe the cost model for only a single client type.

3.3.2 Terminology

Recall that HILDA models an application in a hierarchical manner, where each AUnit contains other AUnits. Let aid be a unique identifier associated with each AUnit definition. Then, we define the *class graph* of a HILDA program P as:

Definition 1: $ClassGraph(P) = (V, E)$ where $V = \{v|v$ is an AUnit definition in $P\}$, and $E = \{(v, w)|v, w \in V$ and w is a child AUnit of $v\}$ $\diamond$

For a valid HILDA program, the class graph must be a DAG. However, instances of an AUnit may be activated for different keys. These instances can be uniquely identified by the pair (aid, key), where key corresponds to the set of evidence that leads to the activation of a given AUnit instance. This leads us to the definition of the *key tree* of a given HILDA program P:

Definition 2: $KeyTree(P) = (V, E)$ where $V = \{(aid, key)|$ aid is the identifier

of some AUnit definition in $P\}$, and $E = \{(v, w) | v, w \in V$ and *w.aid* corresponds to a child AUnit of the AUnit corresponding to *v.aid*$\}$ $\diamond$

Note that each AUnit corresponds to a different key, where the key includes the key of the AUnit's parent node. Therefore, an instance of any AUnit, except the root, is activated by exactly one parent. Thus the key tree must be a tree. Note that the class graph of a HILDA program is effectively an aggregated version of the key tree, obtained by merging nodes that have the same aid. In order to estimate the response time of a system, we require for each node of a key tree, various annotations such as the expected time for processing the AUnit instance and expected data to be processed for that instance. We next define an annotation function for the key tree of a HILDA program P as:

Definition 3: The *annotation function* $\mathcal{A}\ (P) : V \rightarrow R^4$ for the key tree (V, E) of a hilda program P is a function that maps each node v of the key tree onto a 4-tuple, where the fields correspond to the following: the probability p_v that a randomly chosen activation over the space of all executions of P is on the AUnit that v is associated with; the expected time t_v for processing queries of the node, the expected sum d_v of the size of input and output data, and the expected number l_v of connections established by this node between the client and the server, respectively. By the definition of the probabilities p_v we also have $\sum_v p_v = 1$. $\diamond$

We describe here the partitioning of a HILDA program into the client part and the server part, at the granularity of its key tree. Whether an AUnit instance is activated and evaluated at the client or the server depends on how the partitioning is done. Let $\zeta : V \in KeyTree(P) \rightarrow \{client, server\}$ be a function speci-

fying where AUnit instances in the key tree of program P are located. Then, we define a parameter α, which expresses the proportion blowup in computation time between the client and the server, as:

$$\alpha = \frac{t_u}{t_v} \text{ where } \zeta(u) = server \text{ and } \zeta(v) = client.$$

The data size of any given AUnit instance is assumed to be independent of ζ. This is because the input and output data of any instance remains the same, regardless of whether the instance is located at the server or the client. The partition of a hilda program P, then, is defined by a cut C as:

Definition 4: $Partition(P) \equiv C = (G_s, G_c)$ a cut in the tree $KeyTree(P) = (V, E)$ s.t. G_s and G_c are disjoint, G_s is connected, the root node belongs to G_s, and $V_s \cup V_c = V$. $\diamond$

In this definition, $G_s = (V_s, E_s)$ is the part that runs on the server and $G_c = (V_c, E_c)$ is the part that runs on the client, i.e., an AUnit instance *a* will be activated and maintained at client side iff $\exists v \in V_c \ni a.key = v.key$. We denote the set of edges between the two sides of the partition by $E_{cut} = E - (E_s \cup E_c)$.

3.3.3 Cost Model

We now define our cost model. Our goal is to optimize the average response time for users. Optimizing other goals, such as system throughput, would involve a similar analysis but a different cost model. We leave this as future work. Recall that we assume that the key trees corresponding to different types of

clients are independent of each other, and do not affect the cost model for any given tree. We therefore consider the key tree for only a single type of client.

Given the key tree $KeyTree(P) = (V, E)$ of a program P, the annotation function $\mathcal{A}$, and the cut $C = (G_s, G_c)$ that partitions the tree into server and client subgraphs, we define the expected user response time as:

$$cost_C(P) = \sum_{v \in KeyTree(P)} p_v \times t_v^C$$

The time to perform AUnit instance processing, given a partition, includes the time to process the AUnit instance at the client (return queries and later reactivation queries), the time to send query results to and from the server and the time to process the queries at the server:

$$t_v^C = t_v^{client} + t_v^{data} + t_v^{server}$$

where t_v^{client} is the expected time for processing v at the client, t^{data} is the expected time for sending result sets to and from the server, including the time for preparation of the data, and t_v^{server} is the expected time for processing queries at the server. Based on our earlier assumption that the time to process an AUnit instance at the client is proportional to the time to process the same instance at the server, and assuming that the data transmission time for transferring a result set between client and server is proportional to the size of that result set, we have:

$$t_v^{client} = \begin{cases} 0 & \text{if } \zeta(v) = server \\ \alpha \times t_v & \text{if } \zeta(v) = client \end{cases}$$

$$t_v^{server} = \begin{cases} t_v & \text{if } \zeta(v) = server \\ 0 & \text{if } \zeta(v) = client \end{cases}$$

We also have

$$t_v^{data} = \begin{cases} \gamma \times d_v + L \times l_v + d_v/\beta & \text{if } \exists u \text{ s.t. } (u, v) \in E_{cut} \\ 0 & \text{otherwise} \end{cases}$$

Here, the data transmission cost t_v^{data} consists of three parts: the expected time for preparing the data to transfer, expected overhead of the handshaking process for establishing TCP connections, and the expected time for transferring the data. We assume that the expected time for preparing and transferring data is proportional to the expected amount of data transferred, with proportionality constants γ and β, respectively. L is the expected overhead for the handshaking process(initial round trip time), which allows us to take into account the number of connections.

These definitions yield the following optimization problem to choose a cut C for a program P:

$$\arg\min_C cost_C(P)$$

We define an additional constraint to take into account client memory limitations. Let $M_C(T)$, the memory usage at the client given the cut C, be given by:

$$M_C(P) = \sum_{v \in KeyTree(P), \zeta(v) = client} m_v$$

where m_v is the maximum memory that is used by any query of the AUnit instance v. Then, if $\hat{M}$ is the maximum memory available for the application at the client, we have the constraint $M_C(T) \leq \hat{M}$.

Before presenting our solution for the problem, we want to justify several simplification we made in our cost model. First, we ignore the cost at server side for synchronization and processing heart beat messages, because it is done asynchronously and thus does not noticeably affect users' response time. Second, we do not consider the cost of transferring the run time system and HILDA code to the client side. These are implemented as sticky applets and can be reloaded from the client machine at low cost after being downloaded for the first time. Finally, we ignore web browser rendering time, which should be the same across different partitioning scenarios.

3.3.4 Solution For Partitioning

The problem of finding an optimal partition with constraints for a given key tree can be proven to be NP-hard.

Theorem 1: The HILDA client/server partitioning problem is NP-hard.

Proof: We can prove this theorem by a reduction from the well-known 0-1 Knapsack problem, which is NP-hard.

Consider a Knapsack problem instance with n items, each having a profit p_i and a weight w_i ($i = 1, 2, \ldots, n$). The goal is to find a subset of items with their total weight no more than a given bound W, and their total profit maximized. We can construct the following instance of the client/server partitioning prob-

lem: the key tree will contain $n+1$ items, n of which are leaves, and correspond to the n items in the Knapsack problem; the memory cost for node i being on the client side is w_i, and the client-side memory bound is W; the computing cost for node i being on the server side is p_i, and the computing costs for client side are all 0; the data transfer cost are all 0.

The client/server partitioning problem is minimizing the sum of server-side computing costs for nodes at server side, which is equivalent to maximizing the sum of server-side computing costs for nodes at client side. It is then obvious that solving the client/server partitioning problem instance is equivalent to solving the Knapsack problem instance. Therefore the HILDA client/server paritioning problem is NP-hard. $\square$

Therefore, we design an approximation algorithm, which guarantees to give a result which is within three times of the optimal in the worst case. The technique we use is Randomized Rounding[99]: we first formulate the problem as an Integer Programming (IP) porblem, relax it to a Linear Programming (LP) problem, solve it, and use a randomized algorithm, similar to that in [59], to round the solution to an integral one that is not much worse.

Given a key tree $KeyTree(P) = (V, E)$, for every node $v \in V$, we define a variable x_v and for every edge $e \in E$, we define a variable y_e. The optimal partition problem for a given HILDA program P, with the annotation function $\mathcal{A}$ can then be formulated as the following IP problem: Constraints:

- $x_{root} = 1, root$ is the root of KT
- $\forall v \in V, x_v \in \{0, 1\}$
- $\forall e \in E, y_e \in \{0, 1\}$
- $\forall e(v_1, v_2) \in E$ $x_{v_1} \geq x_{v_2}$ and $y_e \geq x_{v_1} - x_{v_2}$

- $\sum\limits_{v \in V} (1 - x_v) * M(v) \leq \hat{M}$

Minimizing function:

- $\sum\limits_{v \in V} x_v * s(v) + \sum\limits_{v \in V} (1 - x_v) * c(v) + \sum\limits_{e \in E} y_e * n(e)$

For each node $v \in V$ and edge $e = (u, v) \in E_{cut}$,

$c(v) = \alpha \times t_v$ is the computing cost at client side

$s(v) = t_v$ is the computing cost at server side

$M(v) = m_v$ is the memory cost at client side

$n(e) = (1/\beta + \gamma) \times d_v + L \times l_v$ is the data transfer cost

The optimal solution for above integer programming will give us an optimal partition $c = (G_s, G_c), E_{cut}$ in the following way:

$$x_v = \begin{cases} 0 & \text{if } v \in V_s \\ 1 & \text{if } v \in V_c \end{cases} \quad \text{and } y_e = \begin{cases} 0 & \text{if } e \notin E_{cut} \\ 1 & \text{if } e \in E_{cut} \end{cases}$$

We can relax the above problem, by allowing $x_v \in [0, 1]$ and $y_e \in [0, 1]$, and get an LP problem that is solvable in polynomial time, with solution X^*. We can then round each x_v^* to 0 or 1 with a threshold uniformly randomly chosen from $[1/3, 2/3]$. This special rounding technique guarantees that the objective function and constraints are still within a reasonable bound. The following algorithm find a number t so we can round each x_v to 0 if $x_v \leq t$ and to 1 if if $x_v > t$

1: **RoundingCut:**

Input: (KT, C, S, N, M)

// KT is the key tree, C, S, N, M are the client cost, server cost, bandwidth cost, main memory cost for each node and edge in KT

Output: c // estimated optimal partition on KT

2: Construct the linear programming problem as mentioned above on (KT, C, S, N, M)

3: Solve the linear programming problem and get optimal solutions (X, Y) where $X[v]$ gives the optimal solution for variable x_v and $Y[e]$ gives the optimal solution for variable y_e

4: $X_{optimal} \leftarrow NULL$

5: $min \leftarrow 0$

6: **foreach** t in X **do**

7: construct X' where X'[v] = 0 if if $x_v \leq t$ and and X'[v] = 1 if $x_v > t$

8: construct Y' where Y'[e] = X'[v] - X'[w] and e=(v,w)

9: $optimal \leftarrow$ evaluate minimizing function on X' and Y'

10: **if** $min \leq optimal$ **then**

11: $min \leftarrow optimal$

12: $X_optimal$ = X'

13: **end if**

14: **end for**

15: Construct c based on $X_{optimal}$ according to the method mentioned above

It is provable that the response time of the partition generated by our algorithm is at most three times as much as the optimal response time obtained

using LP, which is no more than the real optimal response time.

Theorem 2: The approximated solution produced by RoundingCut algorithm is at most 3 times as much as the optimal partition solution. The client side memory consumption M under the approximated solution are at most 3 times as much as the constraint $\hat{M}$.

Proof: If all the variables $x_v(v \in V)$ and $y_e(e \in E)$ are constrained to be 0 or 1, then the integer programming will give the optimal solution to the partition problem. By relaxing the variables to take real values in $[0, 1]$, we get a linear program, whose solution gives a lower bound to the value of the optimal partition. So we only need to construct an integral solution that has value within a constant factor of the optimal solution to the linear program.

Consider the following randomized rounding algorithm:

- Solve the LP optimally, and denote the optimal solution to it as $x_v^*(v \in V)$ and $y_e^*(e \in E)$.
- Generate t uniformly at random from $[\frac{1}{3}, \frac{2}{3}]$.
- For all $v \in V$ s.t. $x_v^* \geq t$, put v at the server side, and the remaining nodes are at the client side.

If we were doing the rounding with t chosen uniformly at random from $[0, 1]$, the *expected* solution will satisfy all the constraints and have the value as the LP optimal. However each particular rounded solution might not have the optimal value, and to be worse, it might also violate the memory constraints. We want to show that there exists one rounded solution, which *simultaneously* have the following two properties:

- The value is within a constant factor of the optimal solution.
- The memory bound is violated at most by a constant factor.

Let us denote the rounded variables by $\bar{x}_v$ and $\bar{y}_e$, which are random variables depending on t.

Claim 1: The following inequalities hold for the randomized rounding algorithm:

- $\sum_{v \in V} (1 - \bar{x}_v) * M(v) \leq 3\hat{M}$
- $\sum_{v \in V} \bar{x}_v * s(v) \leq 3 \sum_{v \in V} x_v^* * s(v)$
- $\sum_{v \in V} (1 - \bar{x}_v) * c(v) \leq 3 \sum_{v \in V} (1 - x_v^*) * c(v)$
- $E[\sum_{e \in E} \bar{y}_e * n(e)] \leq 3 \sum_{e \in E} y_e^* * n(e)$

Here $E[\cdot]$ means expectation.

Proof: We will prove the first, the second, and the fourth inequalities. The proof of the third one is the same as the second.

Let set $C = \{v \in V | \bar{x}_v = 0\}$. For any node $v \in C$, $x_v^* < t \leq \frac{2}{3}$, i.e., $3(1-x_v^*) \geq 1$. Then we have

$$\begin{aligned}
\sum_{v \in V} (1 - \bar{x}_v) * M(v) &\leq \sum_{v \in C} (1 - \bar{x}_v) * M(v) \\
&= \sum_{v \in C} M(v) \\
&\leq \sum_{v \in C} 3(1 - x_v^*) * M(v) \\
&\leq \sum_{v \in V} 3(1 - x_v^*) * M(v) \\
&\leq 3\hat{M}
\end{aligned}$$

Let set $S = \{v \in V | \bar{x}_v = 1\}$. For any node $v \in S$, $x^*_v \geq t \geq \frac{1}{3}$. Then we have

$$\begin{aligned}
\sum_{v \in V} \bar{x}_v * s(v) &\leq \sum_{v \in S} \bar{x}_v * s(v) \\
&= \sum_{v \in S} s(v) \\
&\leq \sum_{v \in S} 3x^*_v * s(v) \\
&\leq 3 \sum_{v \in V} x^*_v * s(v)
\end{aligned}$$

Now let us prove the last inequality. It is easy to see that $y^*_e = x^*_{v_1} - x^*_{v_2} (\forall e = (v_1, v_2))$. So $\bar{y}_e = 1$, i.e., edge e is included in the cut, iff $x^*_{v_2} < t \leq x^*_{v_1}$. Since t is uniformly picked from $[1/3, 2/3]$, the probability for t to fall into the range $(x^*_{v_2}, x^*_{v_1}]$ is at most $\frac{x^*_{v_1} - x^*_{v_2}}{2/3 - 1/3}$, which is $3y^*_e$. Then the inequality follows. □

From the Claim 1, we know that the first three inequalities are satisfied *absolutely*, and only the last one is about expectation. Therefore *all* the possible rounded results of the algorithm can at most violate the first two constraints by a factor of 3, they will also be within factor of 3 of the optimal value on the first two parts of the objective function. The property of expectation implies that there exists at least one particular rounded solution that satisfies the last inequality. That solution is the one that is guaranteed to be simultaneously within factor 3 from the optimal solution and the bounding constraint. □

Note that the theoretical bound given here is a worst-case bound. In practice, we have found that the response time obtained using our algorithm is very close to the optimal response time for the applications that we considered in our experimental evaluation.

3.4 Experimental Evaluation

In this section, we first describe the setup for the experiments we performed to evaluate the performance our HILDA system(Chapter 3.4.1). We then compare the performance of a HILDA and a J2EE implementations of a real world application (CMS) and a technical benchmark (TPC-W). These comparisons show the benefits of automatic client-server partitioning (Chapter 3.4.2).

3.4.1 Experimental Setup

We first discuss how we estimate the annotation of a key tree using a trace of the running application. We then describe how we apply the result of the optimization problem to achieve a partition of the application, and we give an overview of the physical setup for the experiments.

Parameter Estimation

A trace consists of a sequence of AUnit activations, along with meta data for the time, data and number of connections associated with each activation.

Definition 5: Let P be a HILDA program. A *trace* $Trace(P) = \langle (i, v_i, t_i, d_i, l_i, t_i^{\gamma}) | 1 \leq i \leq n \rangle$ of P is a sequence of five-tuples called *events*. The number i is the sequence number of the event, $v_i = (aid, key)$ uniquely identifies an AUnit instance in P, t_i is the time taken to process the queries in this instance, d_i is sum of the size of the input and output data for the instance, l_i is the number of connections established between the client and the server, and t_i^{γ}

is the time spent to prepare the data by this instance. $\diamond$

Given the above definition, the annotation function of the keytree of program P can be estimated through an aggregated version of the trace. Since multiple events in the trace may be associated with the same node v of the key tree, we can estimate the value of v's annotation by counting and aggregating the trace data for each node. More precisely, we estimate the annotation function for a node $v \in KeyTree(P)$ as follows. Let $\mathcal{A}(v) = (p, t, d, l)$. Then we can estimate (p, t, d, l) with $(\hat{p}, \hat{t}, \hat{d}, \hat{l})$ as follows:

$$
\begin{aligned}
\hat{p} &= \frac{|\{i|\exists(i, v', t', d', l', t'') \in Trace(P)\}|}{n}, \\
\hat{t} &= \frac{\sum\{t|\exists(i', t, d', l', t'') \in Trace(P)\}}{p \times n}, \\
\hat{d} &= \frac{\sum\{d|\exists(i', t', d, l', t'') \in Trace(P)\}}{p \times n}, \\
\hat{l} &= \frac{\sum\{l|\exists(i', t', d', l, t'') \in Trace(P)\}}{p \times n}.
\end{aligned}
$$

The other parameters for optimization were specified according to the physical setup. We ran the experiments on the PlanetLab network. Given that only powerful desktop clients are used in PlanetLab, we assumed that the client and the server have similar computing power. Therefore, we set parameter $\alpha = 1$, and no bound was imposed for the memory available at the client. The bandwidth (β) of the network was roughly 300KB, and the round trip time L was approximated as 10ms. We could also have estimated these parameters automatically at runtime; this is left as future work. We also estimated γ as follows:

$$
\gamma = \frac{1}{n} \sum_{i \leq n} \frac{t_i^{\gamma}}{d_i}.
$$

Partitioning Logic

The client-server partitioning for a program P is done at the granularity of key trees. Given a cut $C = (G_s, G_c)$ in the key tree, we ship the data of the AUnit instances in V_c to the client. However, note that our constructed annotation function assumes that the future workload is very similar to the one seen before. In practice, the future workload can contain AUnit instances that have never been encountered before. Therefore, the partitioning is also done at the class graph level, using nodes from the class graph as representatives for instances not yet seen in the trace. For unseen instances, we will position the instance based on the computed partitions for the class graph.

Physical Setup

We illustrate the benefits of HILDA using a Course Management System and an Online Book Store application that is based on the TPC-W benchmark. We compare responsiveness of the system (a.k.a. average users' response time) of a HILDA implementation and a J2EE implementation of the two applications. The applications were deployed in a JBOSS application server setup on a 2.66Ghz machine having 4GB of RAM, and used MS SQL 2005 as the backend database management server. The client simulators were deployed on the PlanetLab network, and included the HILDA RTSC.

We measured the response time for each operation, i.e., the time taken to submit a request, process it at the server/client and receive the resulting page from the server. Therefore, this measure includes the time spent on the server to process the request, the time spent at the client and the network transmission

time. However, we did not take into account the time taken by the web browsers to render the resulting HTML pages. Also, in order to reduce the error due to the erratic nature of the PlanetLab network, the experiments were conducted twice. The values we present in the next section are therefore averages over two runs of the simulation.

3.4.2 Experimental Results

We now present experimental results from two applications: a CMS and an Online Book Store.

Course Management System

Our first experiments were performed on CMS, a Course Management System developed at the Cornell Computer Science Department which is currently in use by more than 2000 students, staff and faculty [36]. The original version of CMS was developed using traditional application development tools such as J2EE/EJB, JavaScript and HTML, while a new version has been developed using HILDA.

The J2EE version of the CMS was developed by experienced programmers, and therefore included extensive client-server partitioning that was done manually. Most of the client-side application logic was implemented using Javascript, and thus allowed updating the webpages dynamically. For example, features such as sorting tables based on selected column values, showing or hiding portions of a web page, and caching users' input temporarily in the browser were

Table 3.1: Operations in CMS

Operation	Description	Number
O1	View CMS homepage	24994
O2	View course management system summary	244
O3	Add/remove courses	18
O4	View course property page(as instructor)	219
O5	View course property page(as admin)	83
O6	Edit course property	91
O7	View course homepage(as student)	7912
O8	View course homepage(as instructor)	1858
O9	View student list page	9
O10	View add students page	133
O11	Add/edit students	867
O12	Drop students	48
O13	Update students final grades	25
O14	View adding assignment page	158
O15	View editing assignment page	841
O16	view assignment list	846
O17	View assignment details	20923
O18	Editing assignment	497
O19	View adding category page	205
O20	View edit category schema page	120
O21	View edit category content page	150
O22	Add/remove/edit columns in category schema	103
O23	Add/remove/edit rows of category content	16

Table 3.2: Average Response Time and Data Transmission for CMS

System	Average Response Time(ms)	Average Data Transmission(KB)
J2EE	278.80	17.99
Server Only	312.64	19.09
Client Server	270.01	12.74

already implemented at the client side.

To calculate the average response time, we emulated the operations performed on the CMS in one semester. A usage log consisting of 60000 operations was collected from the J2EE version of the system, along with the necessary parameters. Table 3.1 lists the operations that the users performed. The first three thousand operations from this log were used as a trace to construct the annotation function, which was then used to calculate a partition for the application. The rest of the operations where then tested based on the calculated partition. The average response time shown in Table 3.2 does not include the time to collect the trace. Table 3.2 also presents the performance measure of the application when it is deployed at the server without any partitioning.

It is evident from Table 3.2 that the HILDA version of CMS with automatic partitioning is comparable to the J2EE version in average response time. The automatically partitioned version, however, reduces the average data transferred between the client and the server by roughly 30%.

Figure 3.5 shows the average data transfer for each operation. Owing to the fact that caching user input at the client reduces the amount of data transferred between the client and the server, operations such as O3, O11, O12, O13, O14, O19, O22, O23 that involve updates result in comparatively less data transfer.

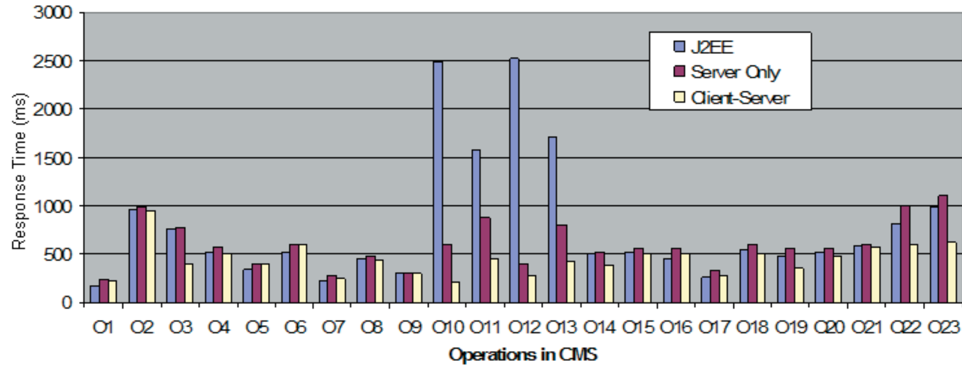

Figure 3.4: Average Response Time for Operations in CMS

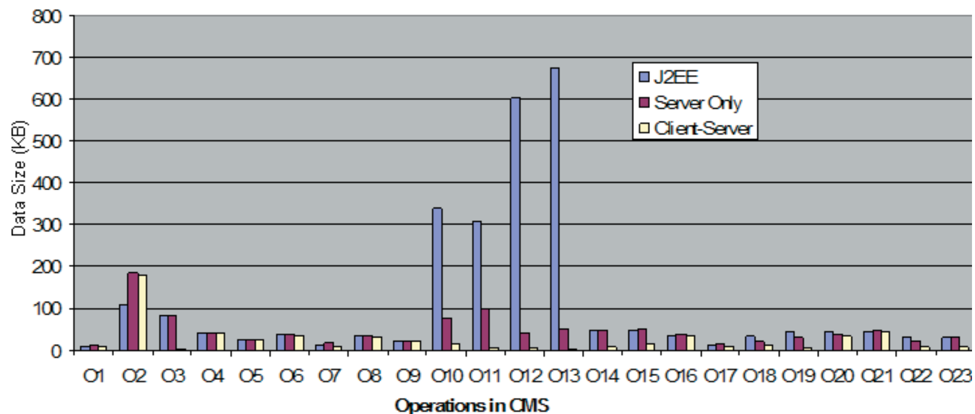

Figure 3.5: Average Amount of Data Transmitted for Operations in CMS

For example, consider O3 – after the system administrator creates a new course, the page is refreshed with a new list of courses. However, if the AUnit for the course list gets pushed to the client, the page generated at client side is able use locally cached data.

The J2EE version of the CMS allows a web browser to cache webpages for later visit, at the page level, while the HILDA run time system caches data at the AUnit (subpage) level. For example, a navigation bar that is present on most pages includes the list of available courses, and contains the assignment and category list corresponding to each course. After partitioning, the HILDA run

time system keeps the AUnit instances for the navigation bar at the client, including the data and the logic to generate HTML segments for navigation bars. Such partial updating yields benefits in the response time for the operations O1, O7, O8, O16, O17, O19, O20. The HILDA runtime system also makes sure that the data for a navigation bar (list of assignments, courses and categories) is up to date, by periodically checking with the server for any changes.

Bad design decisions may sometime result in suboptimal performance. In the J2EE version of the CMS, the logic for users to sort tables based on different columns is always pushed to the client. However, all pages in the system are assembled dynamically, and the Javascript generated on the fly is embedded in the HTML pages. This Javascript makes the size of pages with sortable tables very large (600K on average). It increases the network transmission time and results in poor response time even compared to the HILDA version without any partitions (Figure 3.4: O10, O11, O12 and O13).

Online Book Store

The TPC-W [26] benchmark specifies an online book shop application as the test case for evaluating application server performance. In this application, users can register, view book details, manage their shopping carts and check out, while managers can add new book details into their inventory. We implemented the application using both J2EE and HILDA, and evaluated the average response time of the two systems using a trace synthesized according to the specifications in the benchmark. In the J2EE version, we did not implement any application logic at the client side except for the basic HTML presentations. We took the first 5 percent of the workload as training set for the system to collect

Table 3.3: Operations in TCP-W Online Bookstore Application

Operation	Description	Number
O1	View website homepage	118
O2	Register as new user	999
O3	Add a book to product list	2098
O4	Register an author of a book	970
O5	View book details	1542
O6	Add a book into shopping cart	4593
O7	View shopping cart details	814
O8	View checkout page	918
O9	Checkout	1799
O10	View order status	920

Table 3.4: Average Response Time and Data Transmission for TPC-W

System	Average Response Time (ms)	Average Data Transmission (KB)
J2EE	221.80	21.7
Server Only	231.88	21.9
Client Server	143.48	3.3

statistics, and then measured the response time after the application ran with the computed optimal partition for the HILDA version with partitioning enabled.

Table 3.4, Figure 3.6 and Figure 3.7 show the average response time and the average data transmission for each operation of the application, in the J2EE version and the HILDA versions with and without automatic partitioning. The HILDA system benefits from activating instances of shopping cart AUnit at the client side. A user can add the book she viewed (O5) into the shopping cart (O6)

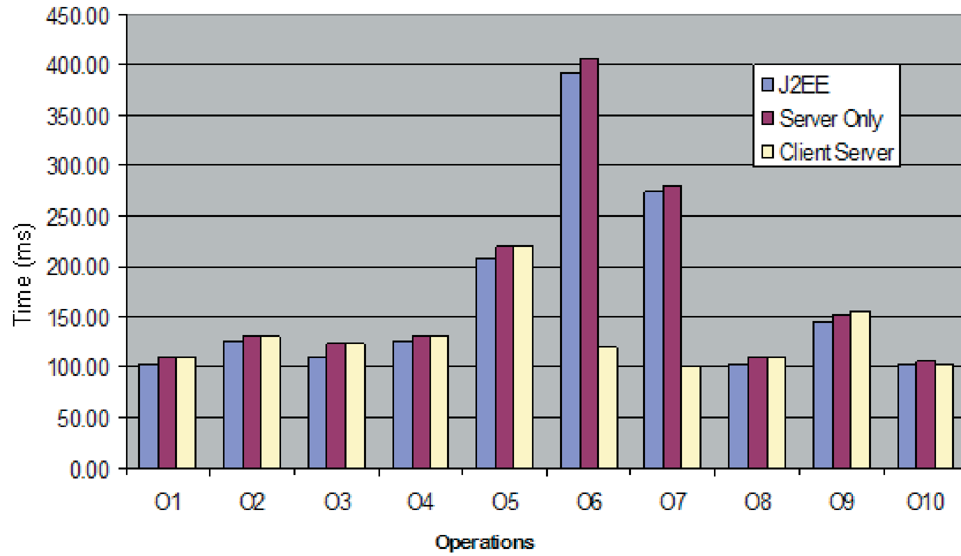

Figure 3.6: Average Response Yime for Operations in TPC-W

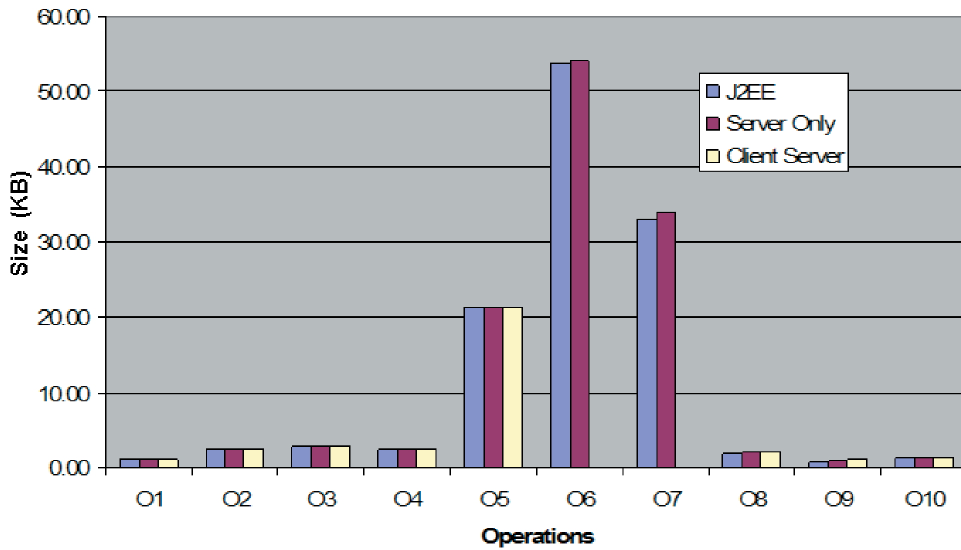

Figure 3.7: Average Amount of Data Transmitted for Operations in TPC-W

and view the details at a later time (O7), possibly before checkout. The shopping cart and the details about the books in the shopping cart are cached along with the AUnit instance, which make the add to the cart (O6) and view detail (O7) operations locally executable, resulting in a much better response time.

CHAPTER 4

WYSIWYG DEVELOPMENT FOR DATA-DRIVEN WEB APPLICATIONS

In this chapter, we introduce the AppForge system, which is designed to allow non-technical users without much database and programming expertise to build fairly complex data-driven web applications. We first give an overview of the system and a running example in Chapter 4.1. Then in Chapter 4.2, we introduce background knowledge and our model of web applications. In Chapter 4.3 and Chapter 4.4, we discuss in details how users can create complex views and how schema are generated automatically based on page views.

4.1 AppForge System Overview

We first provide an overview of the AppForge system architecture, then describe the AppForge GUI using a running example.

4.1.1 System Architecture

Figure 4.1 shows the architecture of the AppForge System. In the front-end, AppForge provides a graphical interface for building and running applications. As developers build an application, the system automatically generates the schema and application logic, and stores this information in the back-end. Developers can run the application at the same time as they are editing it.

The back-end system consists of two sub-systems: Application Creation System and Application Runtime System. The Application Creation System creates

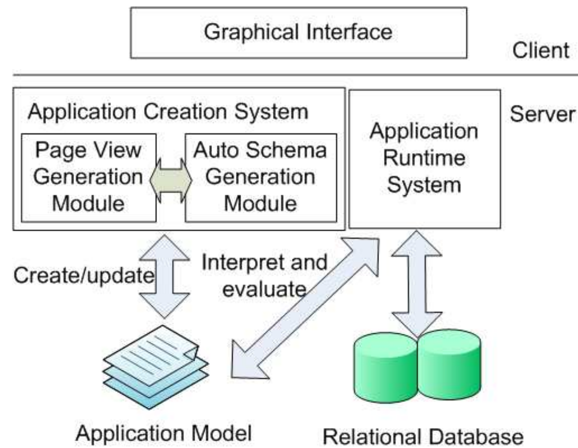

Figure 4.1: AppForge System Architecture

and updates the *application model* based on developers' actions. The application model includes the specification of page views, application logic and database schema. The Page View Creation module provides an interface for creating and updating webpages. Developers' actions at the front-end for creating/editing page views are translated into commands in the Page View Creation module. The Automatic Schema Generation module automatically generates the appropriate relational database schema from page views. Note that in AppForge, building page views and generating the schema is an iterative process: new views are built by navigating the existing schema, and the schema is implicitly updated when page views are updated.

The application model created by the Application Creation System is stored in the file system, while the application state is stored in a relational database system. At start-up time, the Application Runtime System loads the application model into memory, and then serves end users' requests by interpreting the model and issuing SQL queries over the relational database system.

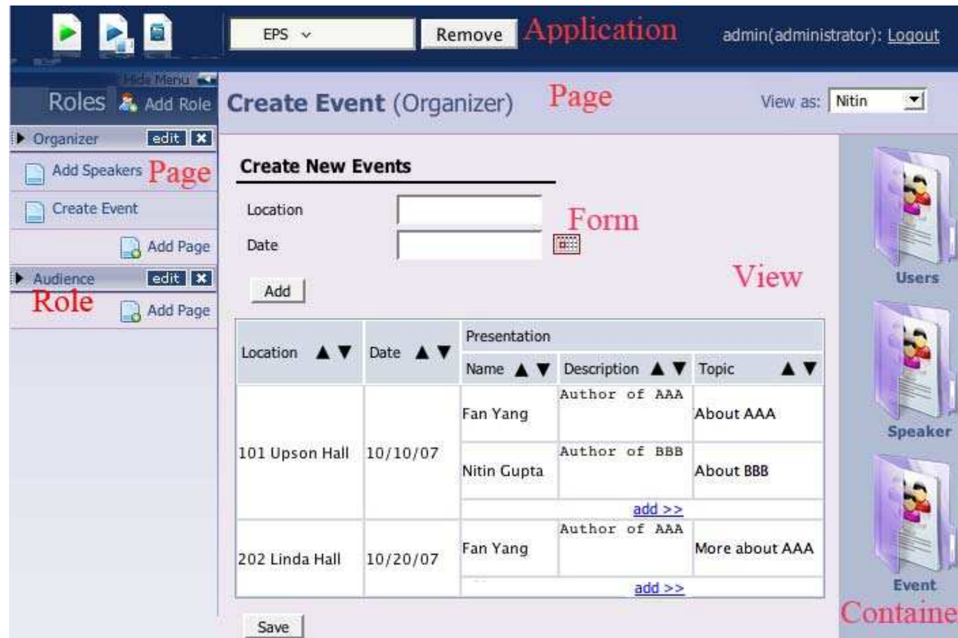

Figure 4.2: AppForge GUI

4.1.2 AppForge GUI

Figure 4.2 shows a screen shot of the AppForge GUI. As shown, the GUI exposes the following abstractions to developers:

- **Application.** Developers can create and manage multiple applications. Each application can be pre-populated with a list of users. For example, in a Yahoo! Group application, the users can be initialized to be all the members of the group.
- **Role.** Users of applications can be divided into multiple roles. Users in different roles can view pages with different content and allowable actions.
- **Page.** Users in each role can access a set of pages. Each page can contain one or more Forms and Views.
- **Form.** Users can use forms to enter new data. Forms are associated with the logic needed to update the relevant database tables. In AppForge, we support

many types of form components such as input fields and drop down boxes.

- **View.** Users can view and update the application state using views. By default, views are presented as nested tables, but other formats such as unnested lists and charts can also be supported.
- **Container.** Containers corresponds to entities in an application. Containers are automatically created when developers add new forms and views to pages. Containers are used as a visual aid and only developers can see them.

4.1.3 Running Example

We now illustrate the AppForge GUI using a running example. Consider a book club in Yahoo! Groups that organizes regular events with invited speakers to give presentations on different books. While there are many event planning sites such as Evite [49], none of them support the specific features required by the book club. Consequently, the book club members decide to build their own customized Event Planning System (EPS).

There are two roles in EPS: organizers and attendees. Organizers can add candidate speakers, create events, and view registered attendees and their feedback after each event. Attendees can register for an event and provide feedback on each speaker. They can also volunteer to help speakers in each event, e.g., by providing transportation.

Using AppForge, members of the group can create such an EPS easily. We now illustrate this process by building several key pages for organizers, including the Create Event page (Figure 4.2), the View Volunteers page (Figure 4.9), and the View Comments page (Figure 4.13). Note that the following screen-

Figure 4.3: Adding a form for creating new events. The resulting form is shown in Figure 4.4.

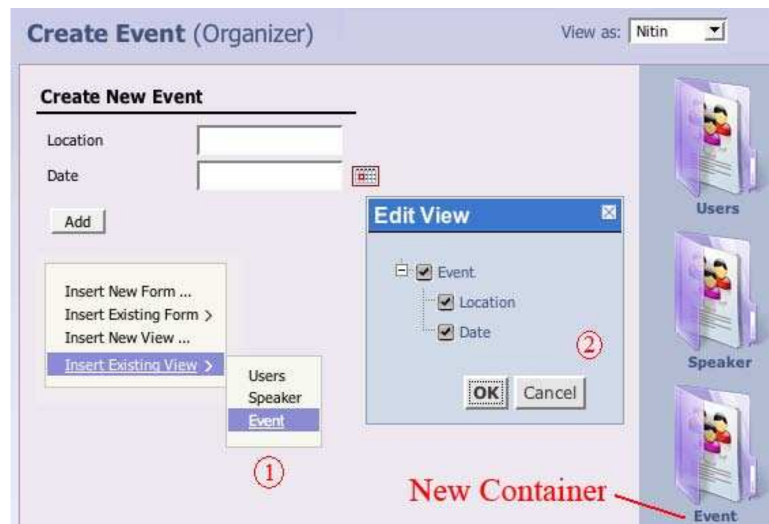

Figure 4.4: Adding a table to show existing events. The resulting table is shown in Figure 4.5.

shots show **all** the steps needed to create these pages, and are thus indicative of the easy-to-use aspect of AppForge. In the following discussion, assume that we have already created an application named EPS with a pre-populated container Users, which contains all book club members, and a container Speaker, which contains information about speakers.

Figure 4.5: Adding the presentation column to the table. Organizers can add speakers for presentations. The resulting table is shown in Figure 4.6.

Create Events Page (Figure 4.2). Organizers can create new events, and add/edit speakers and presentation topics for each event (note that adding speakers and presentation topics associated with an event updates not just the relevant entities, but also the relationship between these entities).

1. Create a form named *Event* with fields *Location* and *Date* as in Figure 4.3. The resulting form is shown in Figure 4.4.

   Automatic schema updates: a new *Event* entity (container) with attributes *Location* and *Date* is created.

2. Add a view over the *Event* container (Figure 4.4.1)[1] and select the columns to show in the view (Figure 4.4.2).

3. Click beside the view (Figure 4.5.1), add a new column named *Presentation* of type *Speaker*, and select columns in *Speaker* to show in the view (Figure 4.5.2).

[1]In the menu, "insert existing view" means inserting a view over an existing container.

Figure 4.6: Adding the topic column to the presentation nested table. The resulting table is shown in Figure 4.2.

Automatic schema updates: a 2-way relationship named *Presentation* between the *Event* and *Speaker* entities is created.

4. Click on nested table for *Presentation* (Figure 4.6.1) and add a new column named *Topic* (Figure 4.6.2). The resulting page is shown in Figure 4.2. End users can click on the link *add >>* under the *Presentation* column to add speakers to each event.

   Automatic schema updates: a *topic* attribute is added to the *Presentation* relationship.

View Volunteers Page (Figure 4.9). Organizers can view the members who volunteered to assist speakers. Volunteers are associated with each speaker in each event.

1. Add a view over *Speaker* and *Event* by navigating from *Speaker* to *Event* in the schema navigation menu (Figure 4.7.1, 4.7.2). The resulting view is shown in Figure 4.8.1.

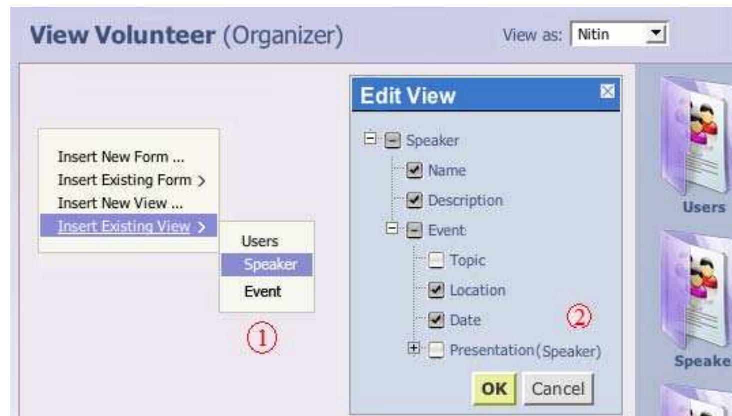

Figure 4.7: Start creating the View Volunteers page by creating a view over *Speaker* and *Event*

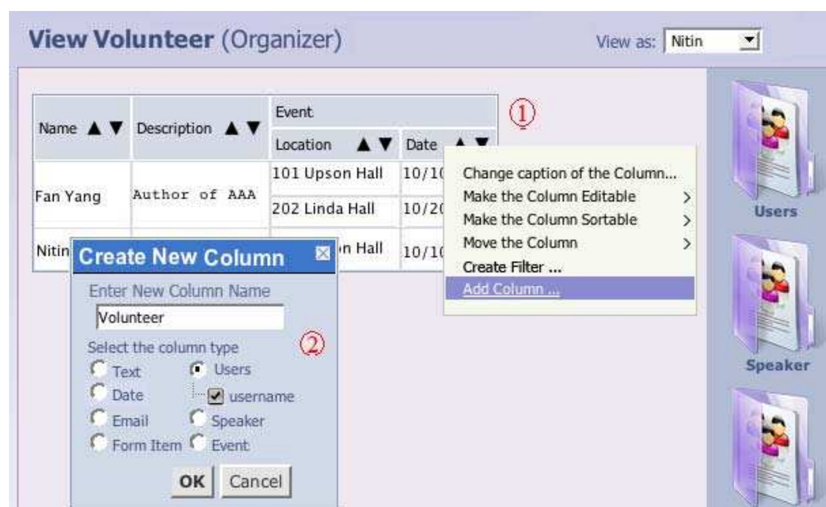

Figure 4.8: Adding the volunteer column to the *Event* nested table

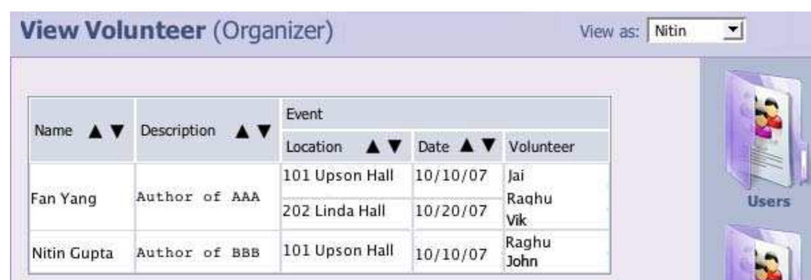

Figure 4.9: The View Volunteers Page

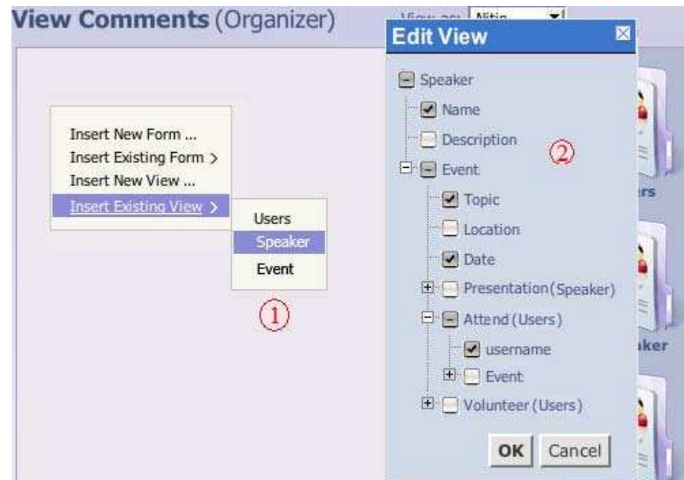

Figure 4.10: Start creating the View Comments page by creating a view over speaker, event and attendee

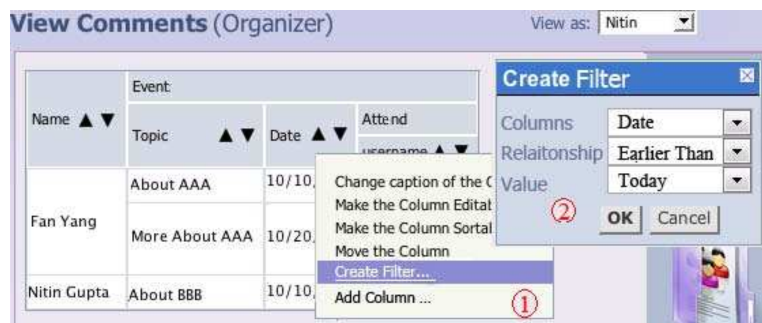

Figure 4.11: Creating a filter to show only past events

2. Click on the nested table for *Event* (Figure 4.8.1) and add a column *volunteer* of type *Users* (Figure 4.8.2).

   Automatic schema updates: an aggregation of *Speaker* and *Event* is created, and then a 2-way relationship between the aggregation and the *Users* entity is created.

View Comments Page (Figure 4.13). Organizers can view comments by attendees on event speakers. The view should only show events that have occurred in the past.

Figure 4.12: Adding a ratings column for each attendee

View Comments (Organizer) View as: Nitin

Name	Event			
	Topic	Date	Attend	
			username	Rating
Fan Yang	About AAA	10/10/07	Jai	Very Interesting
	More About AAA	10/20/07	Andrew	Not Interesting
			Vik	Somewhat Interesting
Nitin Gupta	About BBB	10/10/07	Jai	Very Interesting

Users

Figure 4.13: The View Comments page

1. Add a view over *User*, *Event* and *Speaker* by navigating through the schema navigation menu (Figure 4.10.1) The resulting view is shown in Figure 4.10.2.

2. Click on the *Date* column (Figure 4.11.1) and create a filter that specifies that the event date is earlier than the current date (Figure 4.11.2).

3. Click on the *Attend* nested table (Figure 4.12.1) and create a new column named *Rating* 4.12.2).

   Automatic schema updates: a 3-way relationship with an attribute *Rating* that connects *Speaker*, *Event* and *Users* is created.

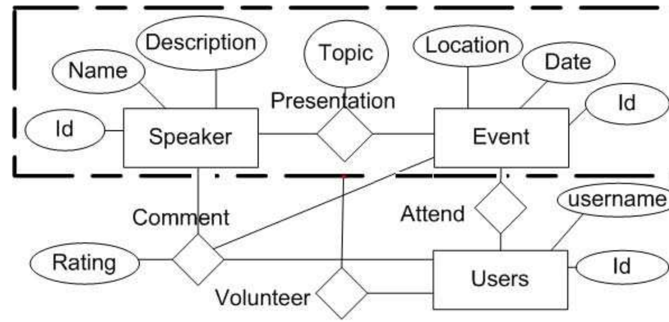

Figure 4.14: Automatically generated database schema

The above examples illustrate how AppForge provides a WYSIWYG environment. Developers always view the application the same way as the end users, and they focus on what they want to present in webpages while the underlying schema is created/updated automatically (Figure 4.14 shows the final schema automatically generated in our running example). Also using the Schema Navigation Menu (Figures 4.4.2, 4.7.2 and 4.10.2), developers can easily navigate through the automatically generated schema and graphically construct complex views (Figures 4.2, 4.9 and 4.13).

4.2 AppForge Application Model

As mentioned in the introduction, two of the key technical contributions of this paper are (a) an algorithm for generating views based on developers' actions and (b) an algorithm for generating the database schema based on page views. In AppForge, the application views and schema is captured formally using an underlying Application Model, which fully characterizes an application. We now introduce the Application Model and describe the algorithms in the subsequent sections.

4.2.1 Background

The Application Model is an extension of the well-known E-R model [20], which is commonly used to model database entities and relationships, and the Nested Relational Algebra (NRA) [2], which is commonly used to represent nested views. We now briefly review the E-R and the NRA model.

The E-R model models the world in terms of entities and relationships between entities. Figure 4.14 shows the database schema automatically generated for our running example in the E-R model. Entities are represented as rectangular boxes, e.g., *Speaker*, *Event* and *Users*, and attributes of entities are represented as ellipses. Relationships are represented as diamond boxes, e.g., *Presentation* and *Comment*. Presentation is a 2-way relationship that connects Speaker and Event, which captures the meaning that speakers present in events. Comment is a 3-way relationships connecting Speaker, Event and Users, which captures the meaning that an attendee gives ratings for each speaker in each event. In the E-R model, a relationship and all its participating entities can be treated as an *aggregation* for the purpose of taking part in another relationship. For example, the dashed rectangular boxes in Figure 4.14 is an aggregation that aggregates Speaker and Event pairs. The aggregation participates as an entity in the Volunteer relationship, which captures the meaning that a club member can volunteer to help a speaker who presents in a event.

In AppForge, views are tables in the nested relational model. The nested relational model extends the relational model by relaxing the first normal form assumption, i.e., a column can contain a nested table. It is more flexible than the relational model because it can model hierarchical data, which are commonly used in Web applications. The nested relational algebra has two extra operators

Name	Location	Date
Fan Yang	101 Upson Hall	10/10/07
Fan Yang	202 Linda Hall	10/20/07
Nitin Gupta	101 Upson Hall	10/10/07

Flat Table

Name	Events	
	Location	Date
Fan Yang	101 Upson Hall	10/10/07
	202 Linda Hall	10/20/07
Nitin Gupta	101 Upson Hall	10/10/07

Nested Table

Figure 4.15: Flat and Nested Tables

compared to the relational algebra: nest (ν) and unnest (μ). ν_C groups all the columns other than C based on the value of C. μ is the reverse operation of ν. As an illustration, in Figure 4.15, the left table is a flat table that shows a list of speakers, and the date and location of the corresponding events. Nesting the table on the name column (ν_{name}) would produce the right table. Columns other than name are aggregated based on name and form a nested table. The effect of unnest is the reverse of nest. Unnesting the right table on the location and the date columns ($\mu_{location,date}$) would produce the left table. The schema of nested tables can be expressed as a nested set of columns. For example, the schema for the right table in Figure 4.15 is {name, {location, date}}.

4.2.2 Application Model

The AppForge Application Model contains the following components.

Database Model: this specifies the schema and constraints for the application states.

- **Schema.** The database schema is represented as an E-R graph. Figure 4.14 represents the automatically generated schema for our running example.
- **Constraints.** Besides a database schema, an application can have addi-

tional constraints on valid application states. In our running examples, users can provide a rating for a speaker in an event (the 3-way relationship in Figure 4.14). However, this relationship only makes sense if the speakers presented in the event (a 2-way relationship) and the users attended that event (another 2-way relationship). Such constraints between n-way and n-1 (and lower) way relationship are captured in the application model and enforced by the Application Runtime System.

Page Model: The page model specifies the content, structure and presentation of webpages.

- **Content and Structure.** The content and structure of a view (and similarly, a form) is specified as a nested relational algebra expression over the E-R graph. For example, the view in Figure 4.2 can be defined by the following algebra expression: $\nu_{Location,Date}(\Pi_{Location,Date,Name,Description,Topic}(Event \bowtie_{Left_{Event.id=Presentation.eventid}} Presentation \bowtie_{Left_{Speaker.id=Presentation.speakerid}} Speaker))$. It joins Event and Speaker through Presentation, projects on necessary columns and nests on columns for Event. The schema of the view is {Location, Date, {Name, Description, Topic}}.
- **Presentation.** These capture presentation aspects of views and forms such as background color, column captions and which columns are updatable.

As mentioned earlier, the application model is automatically generated based on developers' actions such as those illustrated in section 4.1. Specifically, the Page Model is generated by the Page View Creation module and the Database Model is generated by the Automatic Schema Generation mod-

ule (Figure 4.1). Further, the entities and relationships are mapped to relational tables, and nested relational algebra queries are converted into SQL queries at run-time. We now discuss the core abstractions and algorithms used in the Page View Creation and Automatic Schema Generation modules.

4.3 Constructing Views

The Page View Creation module (Figure 4.1) constructs views based on the database schema and developers' actions. The main challenge is making this functionality accessible to developers without database and programming knowledge. Specifically, we would like to enable developers to (a) navigate through a database schema without exposing the complexity of an E-R graph, and (b) create complex NRA expressions without exposing the details of NRA operators such as join and nest.

We address the above two challenges as follows. First, we introduce the Schema Navigation Menu as a visual utility to transform the E-R graph into a navigational tree menu. Using this menu, developers can easily navigate an E-R graph. Second, we describe a set of three graphical primitives for creating and editing NRA expressions over the schema. We then prove that using only these three primitives, developers can construct views that correspond to the large set of NRA expressions with joins on primary/foreign key.

4.3.1 Schema Navigation Menu

A Schema Navigation Menu is a tree structured menu whose root is an entity in the E-R graph. The construction of a menu is initiated when a developer selects the root entity (Figures 4.4.1, 4.7.1 and 4.10.1). The options and structure of the menu are determined by the attributes and relationships among entities in the E-R graph (Figures 4.4.2, 4.7.2 and 4.10.2). At each level of the menu tree, the list of checkable options are produced using Algorithm 1. Note that this algorithm is recursively invoked on demand for each level of the Schema Navigation Menu to produce the hierarchical structure displayed to the developer.

In Algorithm 1, we use term *currentStep* to denote the entity that we are currently expanding. It is initialized to be the root entity. We use term *navigationPath* of *currentStep* to represent the list of entities and relationships through which we have navigated from the root of the menu tree to *currentStep*. *link* represents the relationship through which we just reached *currentStep* from its parent in the menu tree. If the current step is the root entity, *link* is null. At each level of the tree, the following list of checkable options are presented.

- *Entity Attributes.* The list of attributes in *currentStep* (line 2).
- *Relationship Attributes.* The attributes of *link* are shown as if they were attributes of *currentStep* to avoid explicitly exposing the relationship to developers (line 3). For example, in Figure 4.10.2, topic, which is an attribute of the Presentation relationship, is shown along with other attributes of Speaker. For each n-way ($n > 2$) relationships that *currentStep* participates in, we check if navigationPath of the *currentStep* contains all the entities that the n-way relationship connects. If so, we show the attributes of the

Input: *currentStep* : The current entity being expanded
link : The relationship through which *currentStep* was reached
Output: *Items* : List of options that can be selected by developers for currentStep

```
AttrForNextStep (link, currentStep)
Items = currentStep.attributes
Items += link.attributes
foreach Relationship r that currentStep is involved in do
  if r is 2-way relationship then
    nextStep = r.otherSide(currentStep)
    if nextStep is not an aggregation then
      Items += nextStep
    else
      Items += all entities in the nextStep aggregation
    end if
  else if navigationPath of currentStep contains all entities participating in r
  then
    Items += r.attributes
  end if
end for
if link forms an aggregation Agg then
  foreach relationship r that Agg is involved in do
    if r is 2-way relationship then
      nextStep = r.otherSide(Agg)
      if nextStep is not an aggregation then
        Items += nextStep
      else
        Items += all entities in the nextStep aggregation
      end if
    else if navigationPath of Agg contains all entities participating in r then
      Items += r.attributes
    end if
  end for
end if
```

Algorithm 1: Algorithm for transforming a database schema into a Schema Navigation Menu. The algorithm specifies how to generate options for each step in the menu tree.

n-way relationship as well (lines 12-13).

- *Navigational Link.* If *currentStep* is connected with other entities by 2-way relationships, those entities will be shown in the menu (lines 4-8). For example, in Figure 4.7.2, Event is shown under Speaker since they are connected by the Presentation relationship. If *currentStep* is connected with an aggregation through a 2-way relationship, all the entities in the aggregation will also be included in the menu (lines 4-10). A navigational link is shown as an expandable item. Selecting this item will expand the menu to show the options for that entity.

  If *link* participates in a relationship as an aggregation, it is treated in the same way as currentStep (lines 15-25). For example, the Presentation relationship forms an aggregation and connects with Users through the Volunteer relationship (Figure 4.14). When navigating from Speaker to Event (*link* is Presentation), the option volunteer (Users) is shown under Event in the menu (Figure 4.10.2).

When displaying an entity name, we sometimes also include the relationship name if we can navigate to the same entity through more than one relationship. For example, in Figure 4.10.2, when starting from Event, we can reach Users as attender or volunteer; the relationship names are used to distinguish these cases.

4.3.2 Graphical Primitives for Editing Views

AppForge provides the following graphical primitives for developers to edit views. These primitives are automatically translated into NRA expressions.

Select Menu Item. From the Schema Navigation Menu, we can select the following options, each of which updates the underlying view specification.

- *Entity Attributes and Relationship Attributes.* Developers can select what attributes to show in the view. This action corresponds to the projection operator in NRA. For example, in Figure 4.4, developers can select which attributes of Event are to be shown in the table.
- *Navigational Link.* By navigating to a new entity, the underlying view will be updated by joining the new entity through the navigation relationship. By default, a nested column is created to show the attributes selected after each navigation. For example, if we navigate from Speaker to Event and then to Users as in Figure 4.10.1, and select attributes to show along the way, the view will be created by joining the three entities through the Presentation and Attend relationships. Nested columns will be created for columns of Event and Users, producing the view shown in Figure 4.11.

Move up/down columns. Developers can change the nesting structure of the view by moving columns up and down the view. If they move a column down, they will be asked which nested column it should be moved into, or if the system should create a new nested column. For example, moving down both the Location and Date columns in Figure 4.15 (Left) into a newly created column called Event will produce the nested table in Figure 4.15 (Right). Similarly, moving up the Location and Date columns in Figure 4.15 (Right) will produce Figure 4.15 (Left).

Create filter. We can limit the data shown in a view by specifying a filter predicate of the form (*column operator value*). Operator can be any comparison op-

erators supported by the underlying database system. Developers can input a constant value or select from a list of context variables supported by the system, e.g., the current date. Figure 4.11 shows an example filter that selects past events.

4.3.3 Expressiveness Theorem

We now formally characterize the set of NRA views that can be constructed using AppForge. For ease of exposition, we assume a simple translation from the E-R model to the relational model that maps each entity and each relationship into a separate table.

Definition 6: Let R be a n-way ($n \geq 2$) relationship that relates entities $A_1..A_n$, and let e_1 and e_2 be nested relational algebra expressions whose output schema contains the ids of $A_1...A_m$ and $A_{m+1}...A_n$ ($1 \leq m \leq n$), respectively. We define operators:

- $e_1 \bowtie_{R(A_1...A_m;A_{m+1}...A_n)} e_2$

  $= e_1 \bowtie_{(R.A_1id=A_1.id...\wedge R.A_mid=A_m.id)} R \bowtie_{(R.A_{m+1}id=A_{m+1}.id...\wedge R.A_nid=A_n.id)} e_2.$

- $e_1 \bowtie_{left_{R(A_1...A_m;A_{m+1}...A_n)}} e_2$

  $= e_1 \bowtie_{left_{(R.A_1id=A_1.id...\wedge R.A_mid=A_m.id)}} R \bowtie_{(R.A_{m+1}id=A_{m+1}.id...\wedge R.A_nid=A_n.id)} e_2.$

where $\bowtie_b$ and $\bowtie_{left_b}$ are the join and left join operators, respectively, and b is the joining condition. $\diamond$

Intuitively, the two operators represent the join and left join based on foreign key and primary key between two entities that are connected by a relationship.

For the rest of the paper, we interpret the left join operator as being right associative, i.e., $A \bowtie_{left} B \bowtie_{left} C = A \bowtie_{left} (B \bowtie_{left} C)$.

The following definition defines the set of NRA expressions that can be constructed using AppForge.

Definition 7: E is recursively defined as follows:

- For every entity *en*, $en \in E$
- If $e \in E$, then $\Pi_c e \in E$, $\sigma_p e \in E$, $\mu_c e \in E$ and $\nu_c e \in E$, where p is a logical expression on columns in schema of E. c is columns in schema of E.
- If $e_1, e_2 \in E$, then $e_1 \bowtie_{R(A;B)} e_2 \in E$ and $e_1 \bowtie_{left_{R(A;B)}} e_2 \in E$, where A, B are sets of entities.

◇

Theorem 3 Algorithm 1 in conjunction with the graphical primitives in Section 4.3.2 can construct all and only expressions in E.

Proof: Without loss of generality, we assume that all the attribute names are unique. We first inductively prove that all the expressions that can be constructed using the AppForge graphical primitives are in E. Assume that expressions e, $\tilde{e} \in E$ are constructed using a sequence of AppForge graphical primitives, and after applying another primitive, we get a new expression e'. We need to show that $e' \in$ E. If the operation applied is:

- **Select Menu Item.**

- **Entity or Relationship Attributes** If we select a set of attributes m shown in the menu, then $e' = \Pi_m e$
- **Navigational Link.** If we navigate through a link from entity n (e is an expression over n), and reach entity m ($\tilde{e}$ is an expression over m) by following the link r, and then e' = $e \bowtie_{left_{r(m,n)}} \tilde{e}$

- **Move up/down columns.** Let $NODE(t)$ represent the nested table that contains t as an attribute, $ATTR(T)$ represent all the attributes of table T, $NS(T)$ represent the schema for nested table T, and $PARENT(T)$ represent the table that contains the nested table T as a column. Assume that we want to move column t, and T_1 = $NODE(t)$. So, we have $t \in NS(T_1)$.
  - Move up columns: We can move t out of the nested column to the upper level in the table. Let T_2 = $PARENT(NODE(t))$, where T_2 is the destination we want to move t to. The resulting expression would be $e' = e[\nu_{NS(T_2)-\{NS(T_1)\}} \rightarrow \nu_{NS(T_1)\cup\{t\}-\{NS(T_2)-\{t\}\}}]$
  - Move down: We can move t down to an existing nested column or create a new nested column. In the former case, assuming T_2 is the schema tree for the nested column we want to move t into, e' is $e[\nu_{NS(T_1)-\{NS(T_2)\}} \rightarrow \nu_{NS(T_1)-\{t\}-\{NS(T_2)\cup\{t\}\}}]$. In the latter case, e' is $e[\nu_{NS(T_1)-\{NS(T_2)\}} \rightarrow \nu_{NS(T_1)-\{NS(T_2)\}}\nu_{NS(T_1)-\{t\}}]$

- **Create filter.** We can create a filter as a boolean predicate p then $e' = \sigma_p e$

So after each graphical command, the resulting expression is still a valid expression in E.

Next, we inductively prove that for every expression in our algebra E, we can construct it using the set of graphical primitives. Assume we can construct

e_1, e_2 using graphical commands, let e be an expression built from e_1, e_2 by following the inductive steps in Definition 2.

- If $e = en$, where en is a relation, we can construct e by selecting en as the root entity or by navigating to en.
- If $e = \Pi_c e_1$, we can construct e by selecting the set of attributes c from the menus for the table corresponding to e_1.
- If $e = \sigma_p e_1$, we can construct e by creating a filter p on the table corresponding to e_1.
- If $e = \mu_c e_1$, we can construct e by moving the columns c down in the table corresponding to e_1.
- If $e = \nu_c e_1$, we can construct e by moving the columns c up in the table corresponding to e_1.
- If $e = e_1 \bowtie_{left_{R(A,B)}} e_2$, we can find an attribute of A in columns/nested columns of e_1. We start navigation from that column and reach B through link R. The we can construct e_2 using graphical commands based on the inductive assumption.
- If $e = e_1 \bowtie_{R(A,B)} e_2$. Since we can use the left join operator and the *not NULL* predicate to represent the join operator, we can use the previous procedure with an extra predicate *B.id is not NULL* to create $e = e_1 \bowtie_{R(A,B)} e_2$. □

Besides proving the expressiveness of the UI operators, Theorem 3 also illustrates how views (NRA expressions) can be constructed through UI operations.

4.4 Automatic Schema Generation

In the previous section, we described how developers can graphically construct arbitrarily complex views over a given database schema. However, constructing a database schema itself is not an easy tasks for developers. To address this issue, the Automatic Schema Generation module automatically generates complex schemas based on just two simple developers actions: (a) creating forms/views, and (b) adding columns to forms/views. The graphical context (position in form/view) of these two actions is powerful enough to construct arbitrarily complex schemas, including those with n-way relationships and aggregations.

The schema generation algorithm is given in Algorithm 2 . We now walk through this algorithm for the different cases.

4.4.1 Editing entities

Entities are created when developers add *new* forms or views (lines 1-2). The columns of tables and fields of forms map to the attributes of entities. The attributes types information are inferred from the types of graphical components used in the page. The type can be a primitive type such as link, text, email and form components, or it can be an entity type (Figure 4.5.2) . Developers can edit forms and views later by adding fields and columns. If developers add new columns of a primitive type by clicking besides the view or on the top level columns of the view, new attributes will be added to the root entity of the view (lines 4-5, 10-11).

Input: *name* : Name of the new forms/views
attrs : Fields/columns in the new form/view

```
// Triggered while developers add columns to views.
onNewFormViewEvent (name, attrs)
AddEntity (name, attrs)
```

Input: *target* : The position in the view where the developer clicks
newAttrName : The name of the column to be added
type : The type of the column to be added

```
// Triggered when developers add columns to views.
onAddAttributeEvent (target, newAttrName, type)
if target is a non-nested column of the view or beside the view then
    targetEntity = root entity of the view
else
    targetEntity = the entity that the target column belongs to
end if
if NOT isEntity(type) then
    if targetEntity is root entity then
        AddAttribute (targetEntity, newAttrName, type)
    else
        navigationPath = getNavigationPath(targetEntity)
        if navigationPath contains two entities then
            r = the relationship that connects the two entities in navigationPath
        else if exists relationship r that connects all entities in the navigationPath AND exists a constraints that r
        depends on all 2-way relationships in the navigationPath then
            r = getTheRelationship(navigationPath)
        else
            r = createRelationship(navigationPath)
            create a constraint that r depends on all 2-way relationships in navigationPath.
        end if
        AddAttribute (r, newAttrName, type)
    end if
else
    if targetEntity is root entity then
        createRelationship (targetEntity, getEntity(type), newAttrName)
    else
        navigationPath = getNavigationPath(target)
        if exists an aggregation over the navigationPath then
            aggregation = getAggregation(navigationPath)
        else
            aggregation = createAggregation(navigationPath)
        end if
        createRelationship(aggregation, getEntity(type), newAttrName)
    end if
end if
```

Algorithm 2: Algorithms for automatically generating a database schema when editing views

As an illustration, in Figure 4.3, adding a form automatically creates an Event entity (Figure 4.4) and the fields in the form are mapped to the attributes of the entity. An id attribute is also automatically created, which is the key for the entity.

4.4.2 Editing Relationships

Relationships are created when developers create and edit views that show information about multiple entities.

2-way Relationships without Aggregation

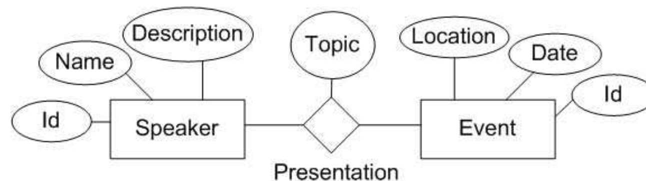

Figure 4.16: The schema generated for the Create Event page (Figure 4.2)

When a developer adds a new column of type entity to a table, a new relationship is created to connect the entity associated with the table and the entity associated with the new column (lines 25-26). As an illustration, in the Create Event page (Figure 4.2), creating a view over Event and then adding a new column to the view (Figure 4.5.1) of type Speaker (Figure 4.5.2), creates a 2-way relationship that connects Speaker and Event.

Attributes can be added to 2-way relationships as follows. When developers add a primitive type column to a nested table, the system adds a new corresponding attribute to the relationship between the top level and nested entities

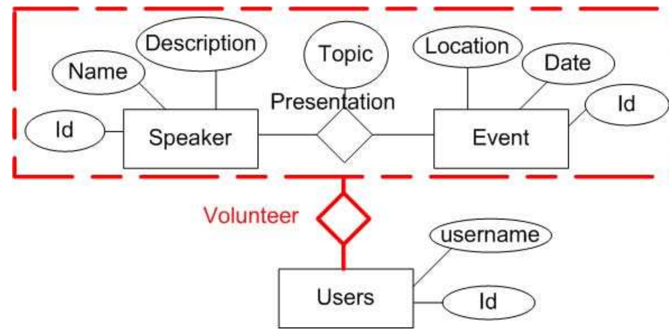

Figure 4.17: The schema generated from View Volunteer page (Figure 4.9)

(lines 13-15, 22). For example, in Figure 4.6, adding the topic column to presentation adds a corresponding attribute to the Presentation relationship because this relationship relates Event and Speaker. Note that this is the desired semantics: topic is associated with a speaker-event pair. The schema generated for the Create Events Page is shown in Figure 4.16.

2-way Relationships with Aggregation

Adding a column of type entity to a nested table establishes a relationship between an entity and the aggregation of the related entities in the view (lines 29-34). As an illustration, in the View Volunteers page (Figure 4.9), adding a volunteer column of type Users to the event nested table creates an aggregation of the Event and Speaker entities, and a 2-way relationship between the aggregation and the Users entity. Note that this is the desired semantics because volunteers are associated with speaker-event pairs. Figure 4.17 shows the schema generated from the View Volunteers page.

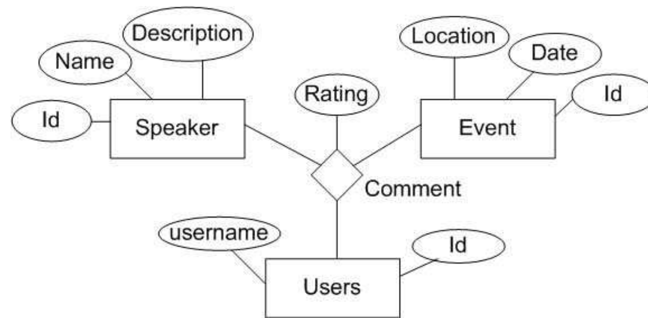

Figure 4.18: The schema generated from View Comments page (Figure 4.13)

n-way relationships

n-way relationships are created by adding primitive type columns to nested tables in views. If an n-way relationship that relates all the entities in the nested view already exists, then an attribute corresponding to the new column is added to the relationship; else the n-way relationship is first created (lines 16-22) before adding the new column. In the View Comments page (Figure 4.9), adding a new column rating to the nested table for attendees creates a three way relationship between users, events and speakers as in Figure 4.18, and adds the rating attribute to the relationship. Note that this is the desired semantics because the rating is associated with a group member for a particular speaker in a specific event.

Note, however, that there are some semantic constraints that are not captured here in the E-R graph. The 3-way relationship should only connect users that attend the event and speakers that present in the same event. Put another way, the 3-way relationship should connect users, events and speakers that are connected by the two 2-way relationships through which we construct the underlying view (Figure 4.7.2). Such constraints cannot be captured by participa-

tion constraints in E-R model because they related multiple inter-related relationships. So besides the 3-way relationship, AppForge will also create a data constraint that the 3-way relationship depends on the two 2-way relationships. By saying that a n-way relationship depends on a set of $n-1$ 2-way relationships, we mean the instances of entities that are connected by the n-way relationship also have to be connected by the $n-1$ 2-way relationships.

4.4.3 Expressiveness Theorem

We now formally characterize the set of E-R graphs that can be constructed using AppForge. An E-R graph can be formally defined as a graph $G = (EN, RE, E)$ where EN represents the set of entities and RE represents the set of relationships. E represents the set of edges that connects entities with relationships and edges that connects relationships with relationships as in the case of aggregations, i.e., $E \subseteq \{(u,v)|u \in EN$ and $v \in RE$ or $u \in RE$ and $v \in RE\}$.

Definition 8: For $e_1, e_2 \in EN \cup RE$, we define $R(e_1, e_2)$ = $\{r|(e_1, r) \in E$ and $(e_2, r) \in E\}$. $R(e_1, e_2)$ is the set of 2-way relationships that exist between entities/aggregations e_1 and e_2. $\diamond$

Definition 9: For $EA \subseteq EN \cup RE$ and $|EA| > 2$, we define $M(EA)$ = $\{r|\forall e \in EA \exists (e,r) \in E\}$. $M(EA)$ is the set of n-way ($n = |EA|$) relationships that connects all the entities/aggregates in EA. $\diamond$

The following theorem fully characterizes the set of E-R graphs that can be constructed using AppForge.

Theorem 4 Algorithm 2 generates all and only E-R diagrams that satisfy the following constraints: $\forall EA \subseteq EN \cup RE$ where $|EA| > 2, |M(EA)| \leq \Pi_{e_1,e_2 \in EA}|R(e_1, e_2)|$ □

The intuition is that n-way relationship created will depend on $n-1$ 2-way relationships, so the number of n-way relationships that could be created on top of a set of n entities cannot be more than the product of the number of 2-way relationships between any 2 entities in the set.

4.5 Preliminary user study of AppForge interface

Given that our primary aim was to support developers who are not experts in databases, we carried out a preliminary user study to test our first interface iteration. The user study consisted of three groups of two people, pairs, who were given three tasks to complete. The tasks were described as follows:

Members of a Yahoo! Group would like to give away unwanted stuff for free. Please create an application that provides the following functionality to members:

1. Post items that they want to give way. Each item includes a name, a description and the original owner (who posted the item).
2. List all the items posted by everyone up to now. Each listing should include the name, description and the owner of the item, and the list of members who have placed a request for the item. The current member can add herself to the requesters list.

3. List the items given away by the current member. Each listing should show the name and description of an item, and the persons requesting the item.

Group 1: Our first pair were two researchers who have advanced degrees in computer science. Both are actively involved in designing, programming and using databases.

Group 2: Our second pair were both researchers trained with advanced degrees in computer science, but neither is a database expert.

Group 3: Our third pair were both experienced computer users. One trained in computer science, but currently in a managerial position with no programming responsibilities; and the other a recruiter familiar with using complex database-backed web applications, but with little formal training in computer science.

Groups were given up to an hour to complete the tasks. Each group was videotaped interacting with the AppForge interface, and all conversation and questions were recorded. The system developers were present to listen to the user interactions, with one of our developers providing advice when needed. Following the trials the development team watched the videos together, made notes and excerpted issues from the sessions, and various redesigns of the interface and considerations for the application were generated and prioritized.

The main finding of our user study was that people who had extensive database experience found mapping the visual presentation we offered to the underlying system structure and logic very easy, completing all three tasks within 20 minutes. Those who were less experienced with database program-

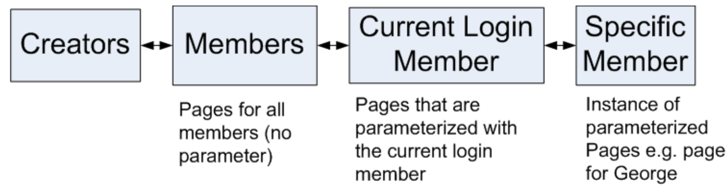

Figure 4.19: Multiple levels of abstractions for developers

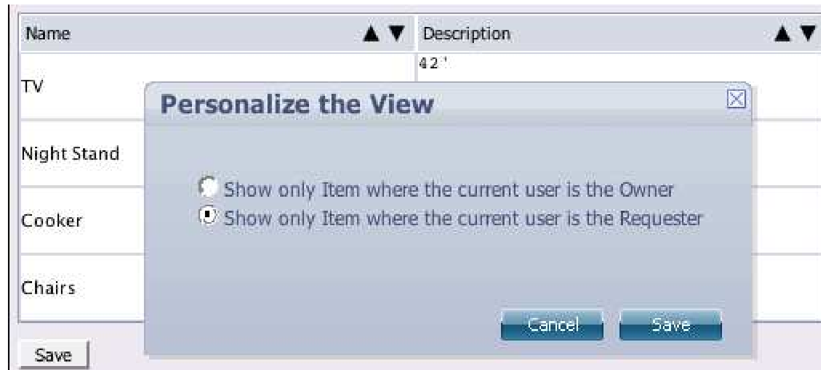

Figure 4.20: Personalization

ming did not find the visual presentation quite so intuitive. Group 2 had minor issues with terminology and interface presentation, taking slightly longer to complete the tasks, and asking more questions of us.

Group 3 were the most challenged, and for us the most interesting of the groups, as they most closely represent our target audience. Therefore we paid special attention to the issues they encountered and have addressed these in our interface redesign.

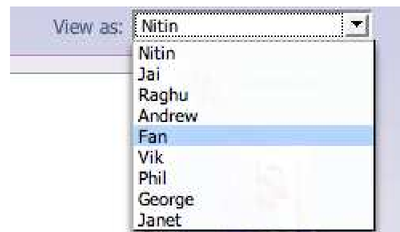

Figure 4.21: Viewing Pages as a Specific User

In particular, Group 3 were confused by the different levels of abstraction that they were required to switch between while developing the application. These levels are illustrated in Figure 19. When creating/adding pages, AppForge developers are the creators of the application, while when viewing and interacting with the pages they have created, they are viewing the pages as members of the Yahoo! Group. In addition, some pages are non-parameterized as for all members (Task 1), and others are parameterized pages where the parameter is the current logged in member (Tasks 2 and 3). Creating parameterized pages proved confusing, with our Group 3 participants struggling to understand the difference between an instance and a variable in place of an instance. We note that these issues are commonly noted in research with novice programmers, and often require careful interface and instruction based scaffolding.

To address this problem, we first redesigned our initial interface to distinguish between the operations AppForge developers can perform as creators and as the intended end users of the applications that they create. We put all the operations for creators in pop-up windows and accessible by right mouse clicks, while all operations for end users were interactive components in the page, e.g., input fields, buttons. To help AppForge users create parameterized pages, we developed the personalization pop-up window, Figure 20, to give suggestions for how to personalize the views (Owner and Requestor are relationships that relate members to items). Our AppForge developers can now choose to view the parameterized pages as a specific user (Figure 21, an instantiated page) which effectively fills in the parameter with a specific user. Using this method, the AppForge developer can see what their intended user would see for the page. Essentially, we developed a WYSIWYG and also a WYSIWTS (what you see is what they see) interface.

In addition to the problem discussed above, we developed a clearer model of containers and views. In our original implementation, we tried to hide the concept of the containers from AppForge users, but our Group 3 participants got confused when multiple forms or views were mapped from the same container. We therefore exposed the notion of containers as collections of data in the visual interface.

Other minor issues exposed during the user study include confusion with the database terminology and poorly delineated interactive areas in the application window (right-clicking on different areas of the application interface revealed different menus). We have addressed these by creating an introductory help panel and a wizard where terms are explained in a Tool-Tips fashion. We also created visual indications of interactive/non-interactive areas, and created consistent menu pop-up and selection.

Having implemented these changes in response to our study results, we are planning a further user study to assess the effect of our modifications on AppForge usability.

CHAPTER 5

HOSTING A LARGE NUMBER OF "SMALL" DATABASE APPLICATIONS

In this chapter, we describe the design and implementation of a data management platform that can host a large number of small applications, i.e., applications that can comfortably fit in a single machine while meeting the desired SLA. We give an overview of the system in Chapter 5.1. In Chapter 5.2 and Chapter 5.3, we discuss various techniques we developed for database replication, migration, and SLA management that ensure the ACID semantics of transactions while still providing full-featured database features such as complex queries and updates, all using commodity hardware and software components. We present our experimental results on multiple TPC-W applications and show that the proposed techniques are scalable and efficient in Chapter 5.4.

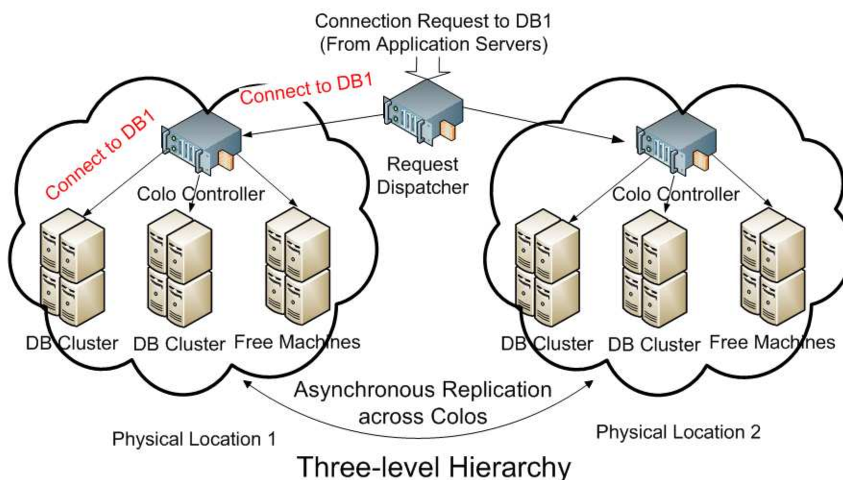

Figure 5.1: System Overview

5.1 System Overview

At a high level, the proposed system provides the illusion of one large centralized fault-tolerant data management system that supports the following API:

(1) Create a database along with an associated SLA. (2) Connect to a previously created database using JDBC and perform the set of operations supported by JDBC interface, including complex SQL queries and ACID transactions. Such connections can be established from Application Servers or any other middleware system.

The main restriction that the system imposes is that the size and SLA requirement for each database should fit in a single physical machine. All other aspects of data management such as failures, resource management, and scaling are managed by the system.

Figure 5.1 and Figure 5.2 show the proposed system architecture.

System Controller. When a client initially connects to a database in the system, the System Controller routes the request to one of the Colos in the system (a Colo is a set of machines in one physical location). The overall system consists of multiple colos with each residing at a different geological location. Database contents are replicated asynchronously across colos to prevent data loss due to disasters. As a common practice, ACID transaction semantic is not guaranteed in such disaster cases. The routing of the client to a particular Colo depends on the replication configuration for the database, the load and status of the Colo (with respect to availability, etc.), as well as the geographical location of the client.

Colo Controller. Each Colo contains one or more clusters. All replicas of a database are hosted by one cluster. The Colo Controller manages the configuration and performance of a Colo. It performs two tasks: (1) Route each connection request to the right cluster that hosts the database, (2) Manage the

pool of free machines and add them to the clusters that need more resources, based on their data size and operation workload requirement.

Cluster Controllers. After the request is routed to the right cluster, a connection will be established between the DB cluster and the application server. Machines within each cluster are interconnected through high-speed ethernet, possibly within the same server rack. Within each cluster, there is a set of controllers that (Figure 5.2 , 5.3 and 5.4) organize the set of DB servers and provide (1) fault tolerance against single machine failure and (2) databases placement based on the service level agreement (SLA), with little manual intervention.

Within each cluster, we have the following architectural components as shown in Figure 5.2 , 5.3, and 5.4.

Connection Controller. (Figure 5.2). Multiple replicas for each database instance are maintained across machines within a cluster to prevent data loss against machine failures. It maintains a map from the database instance to the location of all its replicas. The Connection Controller redirect each query to the correct server using this map. Strong consistency is maintained among replicas within the cluster using synchronous replication. As shown in Figure 5.2, the system execute write requests on all replicas of a database and read requests on one of the replicas following the read-one-write-all strategy [12]. Effectively, each client transaction is mapped into a distributed transaction. To provide transactional semantics to clients, we use two phase commit (2PC) protocol for distributed transactions and the Connection Controller works as the transaction coordinator and each server acts as a resource manager in 2PC.

Recovery Controller. (Figure 5.3). When machines fail, the system continues

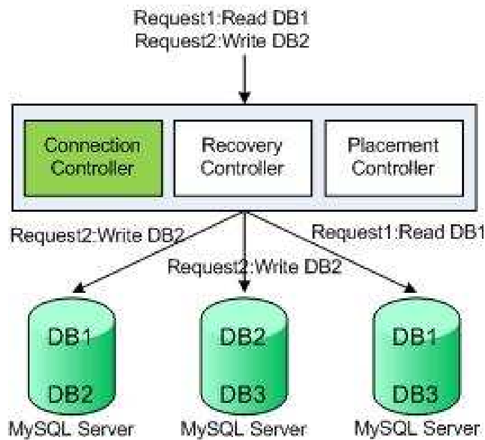

Figure 5.2: Architecture of a DB Cluster.

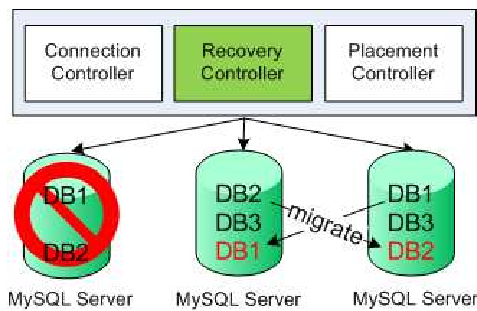

Figure 5.3: Within each DB Cluster: Recovery.

execution with the remaining replicas of the databases. The system runs in a sub-fault-tolerant mode since further machine failures might cause data loss. The Recovery Controller will recover the system to the fault-tolerance state by restoring enough replicas for each database.

As shown in Figure 5.3, after the failure of the first server, new replicas of DB1 and DB2 will be created on the remaining servers. Creating new replicas in our system is different from creating a copy of a database using hot-backup utilities. For hot-backup, the copy can be fuzzy as it is asynchronously brought up to date by applying the change log segment since the start of the copy operation. In the case of synchronous replication, the requirements are more challenging

to satisfy. The replicas created should be in synch with the primary copies in an exact manner.

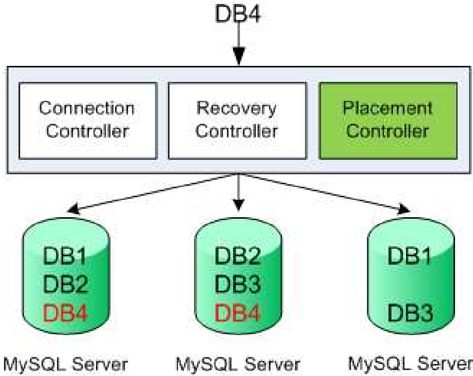

Figure 5.4: Within each DB Cluster: Load Balancing.

Placement Controller. (Figure 5.4).When a new database is added to a cluster, we need to determine which physical servers should host its replicas. The Placement Controller determines how to co-host databases on each server so the SLA of each application is not likely to be violated while minimizing the number of machines used.

MySQL Server. MySQL servers run independently without the knowledge of each other in the cluster. Each of them accepts requests from the Connection Controller and behaves as participants of distributed transactions. Each database replica is hosted entirely on one machine and each machine can host multiple databases at the same time.

5.2 Fault Tolerance

At a high-level, there are two types of failures: (1) Machine failures within a colo. This type of failure is very common, but usually, very few independent

machine failures happen at the same time. (2) Colo failures. This is mostly due to disaster situations and is hence, not very common.

In situation (1), we provide strong ACID guarantees during recovery while for (2), as discussed earlier, we provide weaker guarantees by using asynchronous replication, which is needed for scalability and performance.

The focus for this section is on solutions to situation (1) (For situation (2), we use standard asynchronous replication supported in database systems.) The main idea is simple: we create multiple replicas of a database within a cluster, and use them to mask failures. However there are two technical challenges that arise in this context:

- Ensuring consistency among replicas. While there have been a lot of work on read-one, write all replication strategy [12], they usually prove the correctness based on an atomic commit procedure. However, commercial implementation of two phase commit (2PC), which is commonly used for atomic commit, implement various optimizations, such as early release of read locks etc, which could lead to non-serializable schedules if we are not careful. In this context, we identify such potential cases, and provide provably correct algorithms for ensuring serializability under these cases.
- Automatically creating database replicas using commodity database software tools, while still ensuring ACID transactional consistency and minimal downtime for applications. Specifically, we propose algorithms that coordinate database operations across replicas while table-level migration is in progress, and prove that the algorithms ensure ACID transaction semantics.

5.2.1 Implementing Synchronous Replication

To provide fault tolerance, the system maintains multiple replicas for each database and maps transactions issued from the clients to distributed transactions following the read-one-write-all policy and provides one-copy serializability [12] guarantee. The connection controller and DB servers work as the transaction manager and the resource managers respectively in the distributed transactions. In detail, the Connection Controller performs the following tasks while processing queries:

- For read only transactions, it redirects the query to one replica of the database without using 2PC (our JDBC-like interface include methods for clients to notify the system that if the current transaction is read only).
- For transactions with both read and write operations, it maps each read operation $r_i(x)$ into $r_i(x_A)$, where x_A is some copy of x and maps each write operation $w_i(x)$ into $w_i(x_{A_1})$, . . . ,$w_i(x_{A_n})$ for all the copies x_{A_1}, . . ., x_{A_n} of x (n > 1). [1]
- It works as the coordinator in the 2PC protocol.

The Connection Controller provides a JDBC interface for clients and there are two design decisions we need to make for the implementation (1) how to implement the logic to issue update operations to all replicas and 2) how to issue read operations to one of the replicas.

When an update query comes in, the Connection Controller finds out all the replicas of the database and issues the updates to all of them in parallel.

[1] $r_i(x)$ and $w_i(x)$ represent, respectively, a read and a write operation on data item x by the transaction i.

- **Aggressive Return**: As shown in Algorithm 3, the Connection Controller returns results to the client as soon as after all updates are sent and one update thread completes successfully. Other update threads continue their updates on other replicas in parallel, and if any one of them fails, the transaction will roll back.

Input: u //the update query
Let D be the set of replicas of the database u is updating.
for Each d in D **do**
  Use a thread to issue u to d.
end for
After all updates are sent, as soon as one update thread returns correctly, return the result to the client
If for any thread, the update fails, mark the current transaction as failed, roll back the current transaction and reject all future operations from the clients for this transaction.

Algorithm 3: Aggressive Return

- **Conservative Return**: As shown in Algorithm 4, the result is returned to the clients only after updates to all the replicas complete successfully.

Input: u //the update query
Let D be the set of replicas of the database u is updating.
for Each d in D **do**
  Start a thread to issue u to d.
end for
Return the result to the client only when all the update threads complete successfully.

Algorithm 4: Conservative Return

The Aggressive Return improves response time and system throughput by reducing the time to wait for slow updates compared with Conservative Return.

When a read operation comes in, we need to find a replica to read from. There are three policies we can use as summarized in Table 5.1.

Table 5.1: Different Strategies for Performing Read Operations.

Policy Number	Granularity	Read
1	Per Database	Primary Site
2	Per Transaction	Primary Site
3	Per Transaction	Any Site

- Policy 1, one primary site is designated for each database and all read operations for this database are directed to the primary site.
- Policy 2, one primary site is designated for each transaction and all read operations for this transaction are directed to the primary site.
- Policy 3, no primary site is designated, read operations can be performed at any replicas.

Policy 3 is the most flexible one for the load balancing purpose. The read operations can be distributed using the round robin strategy or based on the workload for each replica.

However, it turns out that not all combinations of the above implementations for read and write operations are correct! In fact, we can show that some combinations of these implementations can lead to non-serializable results as shown in Table 5.1. The main reason for this is the following: although we can achieve global one-copy serializability [12] with strict two-phase locking (strict 2PL) and 2PC for distributed transactions, most modern DBMS implements many optimizations for 2PC. For example, they release read locks after PREPARE and before COMMIT for distributed transactions, which requires us to be very careful when dealing with replication (which has slightly stronger

Table 5.2: Serializability with Strict 2PL Optimization + 2PC

	Conservative Return	Aggressive Return
Policy 1	Serializable	Serializable
Policy 2	Serializable	Not Serializable
Policy 3	Serializable	Not Serializable

requirements than when dealing with independent objects).

Returning to Table 5.1, for the Conservative Return, we can always get global serializability with 2PC. However for the Aggressive Return, with early read lock release, the serializability can not always be guaranteed. For example, assume the database has two replicas on site 1 and 2. We have two concurrent transactions $T1 : r_1(x), w_1(y)$, and $T2 : r_2(y), w_2(x)$. Consider the following schedules for $T1$ and $T2$[2].

$r_1(x), w_1(y), p_1, w_2(x), p_2, c_2, c_1$. On Site 1.

$r_2(y), w_2(x), p_2, w_1(y), p_1, c_2, c_1$. On Site 2.

The schedule corresponds to the case when $w_1(y)$ finishes first on site 1 and start committing the transaction by issuing prepare operation to site 1 while $w_1(y)$ is still executing on site 2 and in the meanwhile similar situation happens for transaction 2 where $w_2(x)$ finishes first from site 2 while still running on site 1. The schedule is allowable in Policy 2 and 3 with the Aggressive Return and 2PC but obviously it is not serializable globally.

[2] p_i, c_i are the prepare and commit operations respectively for transaction i. If read lock is released right after p_i and before c_i, we can allow write operations from another transaction before the commit of the current transaction.

In Policy 1, $r_2(y)$ must happen on site 1 instead, then since p_1 doesn't release the write lock, T2 can not be serialized after transaction T1 while committing before T1. Either T2 will be scheduled before T1 or the 2PC protocol will be violated.

Next, we prove that with Policy 1 and the Aggressive Return, we can guarantee global serializability. We first introduce some notations. A serialization graph is a directed graph (V, E) where V is the set of transactions and $(T_i, T_j) \in E$ if and only if transaction T_i and T_j have conflicting operations o_i, o_j and o_i is scheduled to execute after o_j. We use $T_i \leftarrow T_j$ to represent an edge (T_j, T_i) in the serialization graph. Since with a read-one-write-all policy, guaranteeing global serializability is equivalent to showing that the global serialization graph is acyclic [12], which is what we prove below.

Theorem 5: With Policy 1, Aggressive Return, 2PL with early read lock release and 2PC, the global serialization graph (SG) is acyclic.

Proof: Assume there is a cycle S in the global SG, assume $S = T_1 \leftarrow T_2 ....T_n \leftarrow T_1$. On the primary site, the history contains all the transactions $T_1..T_n$ and they form an acyclic SG G. Note that the global cycle S must be formed by adding edges from SG in non-primary sites on top of G. Let E be the edges added to G, each edge $T_i \leftarrow T_j \in E$ must be formed by a write-write conflict ($w_i < w_j$). Since the prepare operation does not release write locks, and the commit operation is atomic, we have $c_i < c_j$ on the non-primary site. Since w_i and w_j also conflict on the primary site, if $T_i \rightarrow T_j$, then we have $c_j < c_i$ which conflicts with $c_i < c_j$ on the non-primary sites. So we have have $T_i \leftarrow T_j$ on primary site. Similarly, we can prove for every other edge, the same edge must appear in the local SG on the primary site. Thus G contains the cycle S which is

a contradiction with the fact that G is acyclic.

5.2.2 Sufficient Condition for Global Serializability

Next, we give a more general criterion for one-copy serializabilty of read-one-write-all replication. We first introduce some notation. In the following, two operations are said to conflict if they operate on the same data item and at least one of them is a write.

Theorem 6 In a distributed DBS with 2PC, the global SG generated is acyclic **if** at each site, the allowable schedule satisfies following properties : (1)serializable. (2) if o_i precedes o_j then c_i precedes c_j. Here o_i and o_j are conflicting operations in transaction T_i and T_j and $c_{i(j)}$ is the commit operation for transaction $T_{i(j)}$.

Proof. Assume that the global SG contains a cycle $S = T_1 \leftarrow T_2 .... T_n \leftarrow T_1$, since $T_1 \leftarrow T_2$ is an edge in global SG, there must exist a site A_1 where the local SG has the edge $T_1 \leftarrow T_2$. Thus there must exist conflicting operations o_1 and o_2 in the projection of T_1 and T_2 on the site A_1 such that o_1 precedes o_2, so we have c_1 precedes c_2 on site A_1 for T_1 and T_2. Together with the definition of 2PC, we know that T_1 commits before T_2, Similarly, we have T_2 commits before T_3 thus T_1 commits before T_3. Similarly for all edges in S, we get T_1 commits before T_n. But with the edge $T_n \leftarrow T_1$ we have T_n commits before T_1, a contradiction. Thus global SG generated must be acyclic.

With Theorem 5, we can see both the conservative strategy + 2PL with early read lock release and the aggressive strategy + strict 2PL without early read lock

release can ensure global one-copy serializability.

5.2.3 Database Migration

Within one cluster, the Recovery Controller monitors the status for all the machines using heartbeat messages. After a machine crashed, it will detect the failure and notify the Connection Controller to steer requests away from it and use the remaining copies to continue serving requests. In the meanwhile, the Recover Controller needs to create a new replica of databases that were hosted in the crashed machine by copying from a surviving replica. We use the term database migration to represent the process that creates new replicas by copying from a remaining replica of the database. During the process, each database in the crashed machine will be in one of the three consecutive states: (1) Before migration. The database is in a weak fault tolerant state and new failures may result in data loss. (2) in the middle of migration. The database is being copied to create a new replica. (3) After migration. The database is restored to a fault tolerant state.

We use off-the-shelf database copying tools, e.g., mysqldump to create a copy of the database and streaming the result to another MySQL server to create a new copy of the database with the same content as well as the other components, e.g., index, stored procedures. We can still serve read and write requests made to databases in state (1) and (3) but have to reject all the updates during the migration to ensure consistency among the old and new replicas. However, we can not rely on the read lock held on the original copy to block update operations. Because updates will be waiting to get write locks and after the migration

is finished, the write lock will be granted and updates will be made to the original replica but not the new one. This will make replicas of the same database inconsistent, so we need to reject them by the Connection Controller instead of relying on lock mechanism in DBMS.

Rejecting any non-readonly transactions of currently migrating database can render the database unavailable for updates for a long time depending on the size of the database to move. Please notice that, the downtime is determined by the individual database size, which is small by our assumption. However, we can reduce the database down time further by rejecting updates on a smaller scale than a database by doing the migration table by table. We only reject operations for the currently migrating table and allow updates on other table to go through.

Algorithm 5 show the procedures performed by the Recovery Controller while Algorithm 6 shows the steps that need to be taken to serve requests during the recovery phase by the Connection Controller. The new replica is always consistent with the original copy by replying on the fact that the SQL language interface does not allow updating more than one table in one query.

Input: d //the database that need to be migrated
  s //a server that hosts a remaining replica of d
  t //another server that does not host a replica of d
  for every table i in the migrating database d **do**
    Mark table i as unavailable for updates
    Copy i from s to t
    Mark table t as available for updates
  end for

Algorithm 5: Recovery Controller: Migrating databases table by table

---

As we can see from the algorithms, transaction can still go through without

Input: req //the request
Let d be the database that req is issued against.
Let S be the set of servers that host replicas of d
Let s be the machine that failed
if s is not in S **then**
  Run the request req over S.
else if New replica of d is being created on $\{s'\}$ **then**
  if if req is read-only **then**
    Run the request req over $S - \{s\}$
  else
    Let t be the table req is updating
    if t is marked as unavailable for update **then**
      Reject the request and abort the transaction.
    else if t is migrated **then**
      Run the request req over $S - \{s\} + \{s'\}$
    else
      Run the request req over $S - \{s\}$
    end if
  end if
else if No new replica of d has been created yet **then**
  Run the request req over $S - \{s\}$.
else if New replica of d has already been created on machine s' **then**
  Run the request req over $S - \{s\} + \{s'\}$.
  Update $S = S - \{s\} + \{s'\}$.
end if

Algorithm 6: Connection Controller: Serving Requests During Recovery

breaking the consistency between replicas as long as the update is not on currently moving table, even if they are updating multiple tables and some of the tables have been migrated while others have not. Thus it greatly reduced the rejected transactions per database which can be verified in the experiment section. The only complication is that the Connection Controller need to parse the SQL query to determine which table it is updating. A conservative but computationally cheap estimation would be to to use a string match routine to check which table name in the schema of the database appears in the update query string.

Next we prove the correctness of the Algorithms.

Theorem 7. With Algorithm 5 and Algorithm 6, the system can ensure one-copy serializability of all transactions performed during migration.

Proof. We can prove the theorem by showing the invariance that when the write operation on data item x, $w(x)$, is performed, the writings are made to all the replicas of x. Thus the read-one, write-all strategy still holds and from the previous sections, we know the one-copy serializability will hold.

Since update queries in SQL only update a table at a time. We can consider the data item x to be a table t and let $allreplica(t)$ be the set of all replicas of the table. $allreplica(t)$ will be changed only by machine failures and migration of t. When no machine fails, from Chapter 5.2.1, we know updates will be propagated to $allreplica(t)$. After one replica rpl of t fails, t will be in one of the three states (1) before migration (2) in the middle of migration and (3) after migration where rpl has been migrated to a new machine s' and a new replica rpl' is created. If t is in state (1), $allreplica(t)$=T-rpl or if t is in state (3) $allreplica(t)$=T-rpl+rpl', in both cases, from Algorithm 6, updates will be made to all machines in $allreplica(t)$. If t is in state (2), no updates can be made to of t. In all cases, read-one, write-all holds.

Finally, please notice that Algorithm 5 assumes that the target and source machines for migration are given. We will discuss the procedure for picking the source and target servers in the next section, after introducing SLA.

5.3 Enforcing Service Level Agreement Guarantees

In our system, multiple databases can share the same DB server. Databases need to be allocated into a set of servers in such a way that SLA for all the databases can be satisfied while minimizing the number of machines used. We will first formalize it as an optimization problem and give initial solutions for the problem.

5.3.1 Problem Definition

We first define the notion of SLAs for databases. The SLA consists of two main components, both of which are specified for a particular query/update transaction workload:

1. The minimum throughput (measured as transactions per second) over a time period T.
2. The maximum percentage of proactively rejected transactions over a time period T.

The first metric serves as a minimum database throughput requirement. The second metric is more intricate. Proactively rejected transactions are those transactions that are rejected due to machine failures, database replication and migration and does not include transactions that fail due to reasons that cannot be prevented and are inherent to the application, such as deadlocks between two application transactions. Our goal is the keep the percentage of proactively rejected transactions below a specified threshold.

Let rs[j] be the resource requirement to meet throughput in SLA of the database instance j and RS[i] be the resource available on machine i. RS and rs can represent CPU times, main memory size, disk size and network bandwidth. The throughput requirement means the sum of resources needed by all databases hosted on a machine should be less than the total available resources provided by that machine.

Let *machine_failure_rate* be the machine failure time over time period T, *migration_time(j)* be the time needed to migrate database j during recovery and *write_mix(j)* be the percentage of update transactions in the SLA of j and *reallocation_rate* be the time over time period T to reallocation the database due to system maintenance and reorganization other than recovery. The availability requirement makes the constraint that

The percentage of proactive rejected transactions $>$ (*machine_failure_rate* + *reallocation_rate*) * (*migration_time(j)* /T) * *write_mix(j)*

In this inequation, *machine_failure_rate* and *reallocation_rate*) can be estimated from historical information. The formula means there should be enough resources left on each machine for the migration to finish within a time interval.

We can then formalize the problem as follows. We have a set of machines C and a set of hosted database instances D and an allocation matrix M tells how the applications are distributed across machines. We have following constraints that need to hold:

- For any $i \in C$ and $j \in D$, M[i][j]=1 if and only if machine i is hosting the

database instance j, otherwise M[i][j] = 0.

- For $\forall j \in D \sum_{i \in C} M[i][j] >= n$. n is the number of replicas we need to insure fault tolerance.
- For $\forall i \in C \sum_{j \in D} M[i][j] * rs[j] < RS[i]$.
- For $\forall j \in D, \forall i \in C$ if $M[i][k] = 1$ then the available resources $R[i] - \sum_{k \in D} M[i][k] * r[k]$ should be able to copy the database j fast enough to avoid violating the above inequality.

5.3.2 Measuring SLA and Resource Requirement

We estimate the appropriate SLA for each database application and measure the amount of resources needed to support such SLA. This can be done by first host the database in a trial mode using a designated stand-alone server. Under the trial mode, we collect the throughput for the database over a certain time period T (per hour, per day) and use it as the suggested throughput SLA.

We consider three types of resources in our system, CPU, main memory and disk I/O bandwidth. The CPU usage and disk I/O bandwidth can be measured directly using the off-the-shelf monitoring tools provided by MySQL. However, since MySQL uses pre-allocated memory buffer pool for query processing, which is determined when the server starts and can not be changed dynamically, the real memory consumption for a database instance can not be measured directly during runtime. It is observed that for most workloads, there is a working set of accessed data, if the buffer pool is smaller than the size of the working set, the system will be thrashing and disk I/O activity will be greatly increased. We will use the size of the minimum buffer pool that do not thrashing

the system as the memory requirement for sustaining the SLA for the database. We use an indirect method to measure it. On the same server, we run concurrent queries against a set of identical measurement databases[3] with known memory consumptions. We find the maximum number of dummy database instances to saturate the system so that the incoming requests for the database on trial is not queued up. By subtracting the memory consumption for the set of dummy databases, we can get a good estimation about the real main memory usage for the database on trial.

The resources needed for migrating the database can be measured in a similar way.

Input: M, n, m // M is the set of available machines. n is the new database that needs to be hosted. m is the number of replicas we want to create for n
  for $counter = m; counter > 0; counter = counter - 1$ **do**
    $allocated = false$
    for Server s in M **do**
      if $\sum_{\text{j hosted on s}} r[j] + r[m] < R[s]$ **then**
        Allocation a replica of n on s
        Remove s from M
        $m = m - 1$
        $allocated = true$
      end if
    end for
    if $!allocated$ **then**
      break
    end if
  end for
  if $m > 0$ **then**
    Add m new machines and allocate one replica of n on each one of them
  end if

Algorithm 7: Adding A New Database to the System

---

[3]They are dummy databases pre-created for the measurement purpose. The memory consumption for each measurement database can be pre-determined manually by changing buffer pool configuration of the MySQL server to different sizes, restart the server and rerun the queries to find the minimum buffer pool size that can support a given query load.

Input: INPUT: n //the maximum number of concurrent migration processes allowed

Let D be the set of databases that were hosted by the failed machine

while D is not empty **do**

For database d in D, find a pair of working servers s and t that

a. s hosts a copy of d.

b. t ($t \neq s$ and t is not hosting a copy of d) has enough free resources to host a new copy of d while satisfying SLA[4] of all databases on t.

We will pick the database d with such s and t that they do not overlap with the source and target machines of any on-going copying over other databases if any.

if The current running threads is less than n **then**

Remove d from D

Create a thread to make a copy of d from s to t.

end if

end while

Algorithm 8: Creating New Replicas

5.3.3 SLA Based Database Placement

When a new database instance k is added to the system, we need to compute a new allocation matrix M' over $C' = C \cup N$ and $D' = D \cup \{k\}$, where N is the set of new machines, that needs to be added to accommodate k and we want to minimize the size of N while still satisfies the above constraints between M', C' and D' as specified in Chapter 5.3.1.

If we do not allow reallocating existing databases in the system to accommodate the new one, the problem is equivalent to the multi-dimensional bin packing problem [69] which is NP hard. Bin packing problem has been studied extensively and many optimization techniques have been proposed with different approximation factors and computing complexity. In our system, we use a simple yet effective online algorithm (Algorithm 7)for placing new database instances which is modified based on the First Fit algorithm [67].

The algorithm will allocate machines to host m replicas of the new database. For each replica, it will find the first available machine that can host the replica without violating the constraints. Each replica will be allocated to a different server. If we can not allocate them, they will be hosted on separate new servers brought in by the Colo Controller.

If we do allow reallocating the existing databases to accommodate the new one, the problem become much harder since we need to consider the cost of transforming M to M' while preserving SLA for all databases. We leave the optimization problem as a future work.

5.3.4 Enforcing SLA During Database Migration

After a server crashes, the Recovery Controller migrates each database that was hosted on the failed servers while makes sure that all the SLAs are satisfied during migration.

Algorithm 8 shows the process for creating new replicas by the Recovery Controller. For every database d that was hosted by the crashed machine, the Recovery Controller will find a pair of source and target machines s, t ($s \neq t$) such that a replica of d is hosted on s and t has enough free resources to host a new replica of d while not violating the SLA constraints. Then a new process will start migrating d from s to t. In order to maximize the benefits of parallelization, we would try to pick, if possible, the databases that have source and target machines that are not overlapping with the on-going migrations. The system imposes a limit on the number of the concurrent migration processes to avoid overloading and thrashing the system. We will study the tradeoff for the

number of concurrent threads in our experiments.

If there are not enough resources left to host new replicas, e.g., hard disk full, the Colo Controller will be notified and allocate more machines to the cluster. The system administrator only need to add free machines at the Colo Controller's disposal and they will be integrated into the clusters when needed without manual intervention.

5.4 Experiments

In following experiments, we study the performance of different strategies for synchronous replications, failure recovery and effectiveness of the Placement Controller.

5.4.1 Experiment Setup

Our experiment focuses on a single cluster in the system. Since the System Controller and Colo Controller do not perform complex tasks other than content based routing for establishing connections between clusters and clients, the system can easily scale with respect to the number of clusters.

Each cluster contains 10 machines running FreeBSD 6. Each machine has two 2.80GHz Intel Xeon CPUs, 4GB RAM and runs MySQL 5. Each MySQL server uses 2GB memory for the query buffer pool. Machines reside on the same server rack.

We evaluate the system using a variance of the TPC-W benchmark. TPC-W

is a transactional web benchmark. The workload is performed in a controlled internet commerce environment that simulates the retail book store activities of a business oriented transactional web server. In this experiment, we bypass the application servers and only focus on the database components of such web applications. The performance metric is the number of database transactions processed per second. We use our system to host databases for a large number of online book store applications with different data sizes and throughput requirements.

We use a multi-threaded Java program to simulate client browsers. Each thread keep a state machine to perform a sequence of browsing and shopping operations based on the workload specification. The simulator connects to the system using JDBC 5 and issues SQL queries for each user's action. We keep each connection to the database saturated by removing think time between each user's actions. TPC-W simulates three different workloads by varying the ratio between read to write: shopping mix, browsing mix and ordering mix.

The system requires clients to specify the start and end of each transaction explicitly. The clients can tell the system that a particular transaction is read-only. In such case, the Connection Controller will issue the query directly to one replica of the database without using 2PC, as an optimization. Otherwise the Connection Controller will translate the transaction into a read-one-write-all distributed transactions across all replicas of the database. The 2PC protocol is implemented by using conservative prepare strategy. Distributed deadlocks are detected using timeouts and the timeout limit is set to 1.5 seconds.

5.4.2 Varying Database Sizes and Workloads

We evaluate the throughput of the system with different database sizes and workload mixes. We generate database content with different sizes (200MB, 400MB, 600MB, 800MB and 1GB) and evenly distribute them among the 10 machines. The total data content generated is 300G without replication and 600G with replication. We experiment with following replication strategy. We first run the simulation for 5 mins to "warm up" the caches of the system and start measuring the system performance for 20 mins. The requests are issued evenly to each database. We perform each experiment three times and take the average.

- No replications. Each database is hosted in one server and no synchronous replication scheme is used. The query is just redirected to the host by the Connection Controller. Apparently, this scheme does not provide fault tolerance and it is used as a baseline in the experiment. At each MySQL server, the transaction isolation level is set to serializable.
- Replication with read to primary site per database. This is the Policy 1 in Chapter 5.2. The primary site is assigned statically for each database. We do it in such a way that they are evenly distributed across DB servers.
- Replication with read to primary site per transaction. This is the Policy 2 in Chapter 5.2. The primary site is assigned when transaction starts using round robin scheme.
- Replication with no primary site for read. This is the Policy 3 in Chapter 5.2. The read operations are distributed across replicas using round robin scheme.

Figures 5.5, 5.6, 5.7 show the throughput of the system and Figures 5.8, 5.9, 5.10 show the deadlock rate of the system, under different workloads.

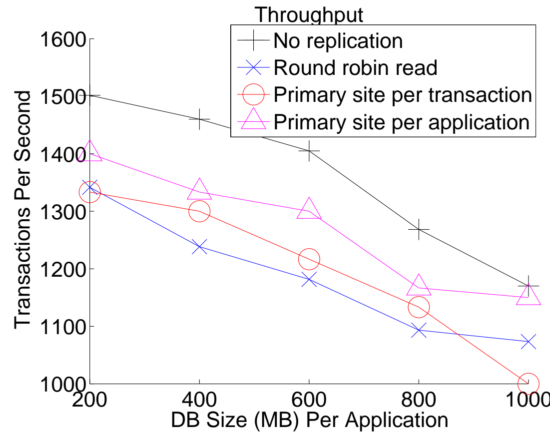

Figure 5.5: Throughput for browsing mix.

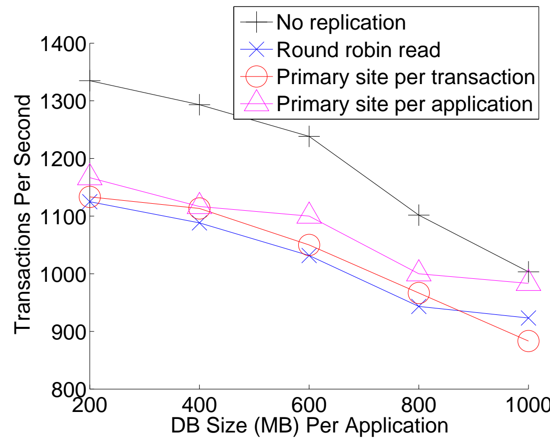

Figure 5.6: Throughput for shopping mix.

From the result, we can tell that with optimization for read only transactions, the overhead for synchronous replications reduced the system throughput by 5-25%, depending on the workloads, over the non-replication case. In three different strategies for read operations, Policy 1 results in the highest throughput on average. The reason that Policy 1 performs better is that with all the reads coming to one primary site, DB servers can have better cache behavior and buffer hit rate.

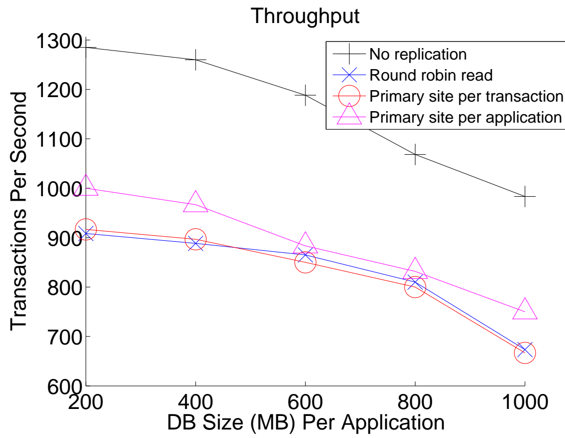

Figure 5.7: Throughput for ordering mix.

Synchronous replications with 2PC increase the chances for deadlocks. This can be explained by longer execution time and more write locks held per transaction. As we can see from the result (Figure 5.8, 5.9 and 5.10), different strategies for replication exhibits similar deadlock rates and Policy 1 is slightly better than others.

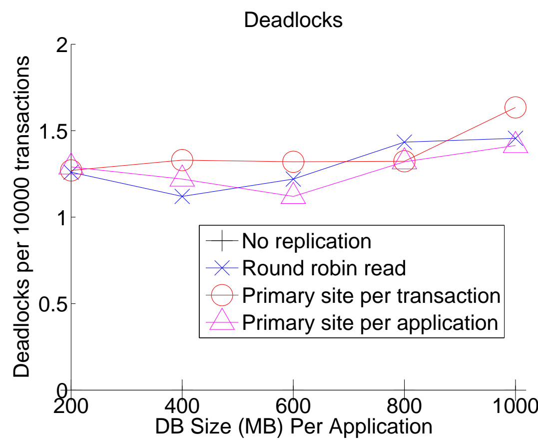

Figure 5.8: Deadlock rate for browsing mix.

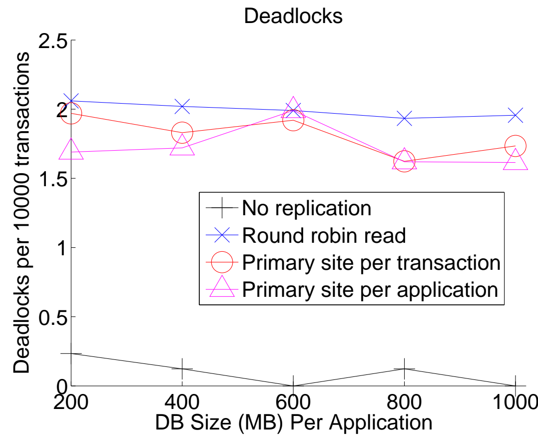

Figure 5.9: Deadlock rate for shopping mix.

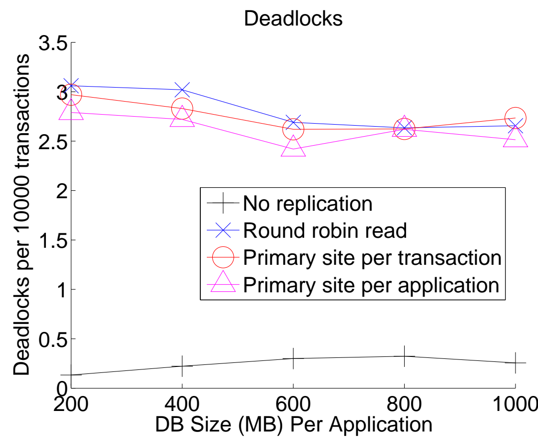

Figure 5.10: Deadlock rate for ordering mix.

5.4.3 Recovery

Next, we will study the system behavior during the recovery phase. We use 200MB as the database size, shopping mix for the workload and 2 replicas per database.

In the experiment, we warm up the system by running it for 5 minutes and then simulate a machine failure by shutting down one MySQL server process on a machine. We keep collecting throughput information until all the databases

that were hosted in this machine have been redistributed to the other 9 machines. In our system, on average it takes 2 minutes to create a new replica of a database with size 200MB. We experiment with both DB level and table level blocking and with different number of concurrent threads to create replicas.

Figure 5.11 shows the system throughput during recovery with different number of concurrent threads. As we can see, the more concurrent threads we use for replication creation, the more overhead it would impose on the system. With only one thread used, the system throughput is decreased by about 10% while with up to four concurrent threads, the system would spend more resources on migrating the database and reduce the throughput by 30%. In the meanwhile, spending more resources on migrating the database would reduce the overall time spent on recovering the system to fault tolerance state as shown in Figure 5.12. The total time is reduced from 16.8 hours to 8.4 hours by allowing more concurrent threads to migration. With 4 concurrent threads, they start to compete for resources and result in longer recovery time than 3 threads.The graph also shows that migrations on DB level and table level result in similar system throughput and the later one has slightly more overhead for bookkeeping and higher chances for transaction abortions.

Figure 5.13 shows the average number of rejected transactions per database due to database migration. The result shows that table level migration would greatly reduced the rejected transactions during recovery. This is because in TPC-W scheme, most updates are on tables with small sizes. They can be copied over much faster than the whole database and thus reduce the time intervals for rejecting updates. The number of rejected transactions is decreased with more recovery threads because the system throughput becomes lower.

Table 5.3: Experiment Settings and Results for SLA Based Placement

Skew Factor	0.4	0.8	1.2	1.6	2.0
Average Size (MB)	531	451	398	361	310
Average Throughput (TPS)	3.75	2.29	1.44	0.59	0.29
# of Machine Used	9	6	5	4	4
Optimal Solution	9	6	4	4	4

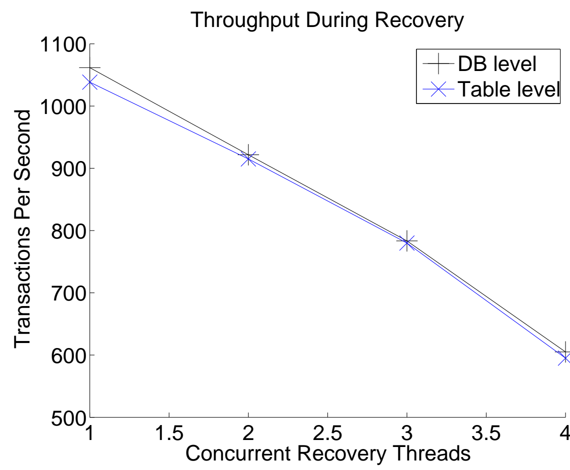

Figure 5.11: Throughput during recovery.

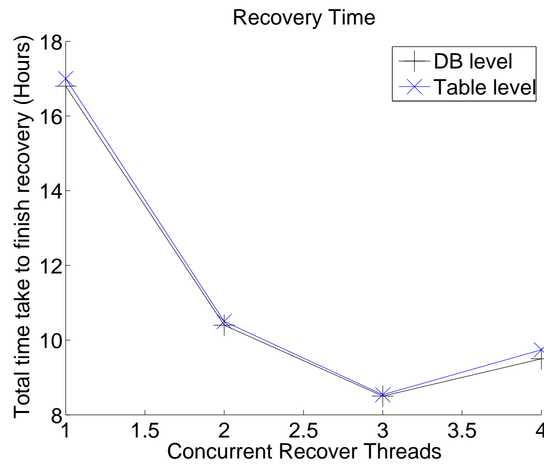

Figure 5.12: Total time to finish recovery.

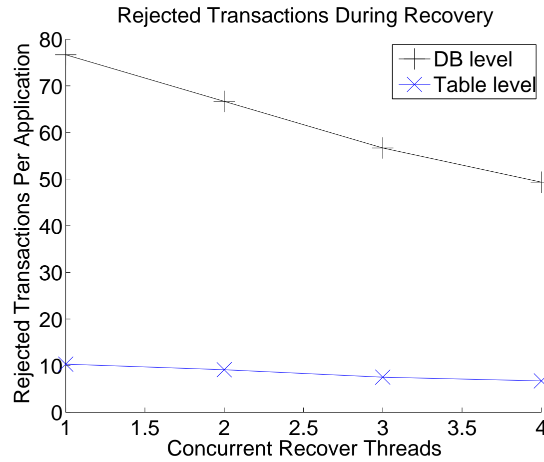

Figure 5.13: Rejected transactions per application during recovery.

5.4.4 SLA Based Placement

We will next study the effectiveness of our database placement algorithm. In the experiment, we randomly generate 300 database instances with database size and throughput randomly picked from 200, 400, 600, 800 and 1000 megabytes and 0.1, 0.2 ... to 10 transactions per second following Zipfian distributions. We use different skew factors for Zipfian distributions from 0.4 to 2 in different run. For each skew factor, we run the experiment three times and pick the best result. We compute the optimal solutions offline using a brute force method. The result is summarized in Table 5.3. As we can see, our algorithm gives a very close estimation to the optimal solutions.

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

In this dissertation, we proposed technologies to address several crucial challenges for developing, optimizing and hosting data-driven web applications.

In Chapter 2, we presented HILDA, a high-level declarative language for developing data-driven web applications. HILDA offers many benefits for application developers, including providing a unified model for all layers of the application, providing a structured programming paradigm for developing websites, and providing support for application conflict detection. Currently, our system provides weaker consistency model than the traditional transaction model. It is a future work to fully explore other alternatives and provide more flexible consistency support.

In Chapter 3, we introduced the HILDA runtime system that automatic partitions applications across client-server and helps in avoiding manual ad hoc and suboptimal decisions. Based on the observed workload, the HIDLA run time system determines a client-server partition of the application, which is close to the optimal partition, using a quantitative method. We also illustrated the benefits of HILDA and automatic client-server partitioning by comparing it with J2EE, using two web applications — a Course Management System with a real workload and an Online Book Store with a benchmark workload. We showed that the performance of the CMS is comparable for both HILDA and J2EE, and that HILDA gains on the amount of data transferred between the client and the server.The TPC-W benchmarked Online Book Store illustrated a 35 percent improvement in response time for HILDA over a J2EE implementation of the same.

The current HILDA optimization model treats each user operation independently, but does not take into account the client side operations performed by the users. Interesting techniques such as asynchronous prefetching and anticipating user actions to prefetch data are not supported. The optimization goal currently focuses only on improving a user's experience and the system's response time. It would be interesting to consider other goals for optimization, such as system throughput by automating load balancing at server side.

Besides performance criteria, there are other concerns, e.g., security, that need to be taken into consideration while partitioning the logic across tiers. Currently our system allow developers to manually annotate each AUnit definition to prevent sensitive information from being shipped to client sides. It is a future work to fully automate the process.

In Chapter 4, we introduced our WYSIWYG system for non-technical users to build data-driven web applications. A growing breed of advanced users are increasingly facing the following dilemma: use a simple graphical tool to build a stripped down version of an application, or go through a steep learning curve and build the more sophisticated application they really want. AppForge tries to provide a solution to this dilemma by expanding the boundary of applications that can be built using a graphical WYSISYG framework. As we have illustrated, AppForge can be used to build fairly sophisticated applications, involving complex schemas and sophisticated page views, without programming or database knowledge.

We have also conducted a small and preliminary user study to evaluate the effectiveness of AppForge. Based on this study, we have identified some concepts that can be confusing to developers, such as multiple levels of user ab-

straction. While we have made some changes based on this feedback, fully addressing and evaluating these aspect is an interesting topic for future work. We are also exploring graphical primitives for capturing more sophisticated application logic such as notifications, workflows, and other forms of information passing between pages (e.g., allowing a user to select an event from a list and navigate to a new page that shows all the events that occur on the same day as the selected event).

In Chapter 5, we have described the design and implementation of a data management platform that can scale to a large number of small applications, i.e., applications that can comfortably fit in a single machine while meeting the desired SLA. In this context, we have developed various techniques for database replication, migration and SLA management that ensure the ACID semantics of transactions while still providing full-featured database features such as complex queries and updates, all using commodity hardware and software components. Our experiments using multiple TPC-W applications show that the proposed techniques are scalable and efficient. As part of future work, we are exploring more sophisticated methods for allocating databases to machines while still preserving SLAs, which can further reduce the hardware cost of the system. We are also exploring extensions to the system architecture that can accommodate "some" applications that are larger than the capacity of a single machine, while the majority of the applications still comfortably fit within a single machine.

CHAPTER 7

RELATED WORK

7.1 Systems and Tools for Building Data-Driven Web Applications

Many tools have been developed to simplify the development of data-driven web applications.

Commercial tools. Sun's Java 2 Platform, Enterprise Edition (J2EE) with Enterprise JavaBeans (EJBs), JSP and Java Servlets, Microsoft's .NET including ASP.NET, and scripting languages like PHP are representative examples of powerful commercial tools for building Web applications. Other tools for designing web sites and HTML pages are surveyed by Fraternali [52]. Such tools typically use a relational database for the database layer, use objects (such as J2EE) and dispatchers to object methods (such as the Model View Controller [37, 32, 39]) for the application logic layer, use a scripting language (such as JavaScript) and HTML links for specifying the web site structure, and use style sheets such as CSS for specifying web site appearance. The main drawback of these approaches is that they do not provide a unified model for all layers of applications, are not declarative, do not use structured programming for web sites and do not provide systematic methods to deal with application-level conflicts.

Declarative approaches. A variety of research prototype systems has been proposed with the common goal of supporting web application development at a higher level of abstraction. Strudel [44] defines the content of web pages in StruQL, a declarative language which can access and integrate semi-structured

data sources and generate web site graphs. However, Strudel only supports read-only operations, but no database updates.

WebML [43, 48] is a powerful and declarative web application development language which shares many common goals with Hilda. WebML provides sophisticated tools for specifying the organization of persistent data, navigational structure, and query/update operations. At its core, WebML extends UML with the concept of "links", which mirror the structure of a web site. The web site is then declaratively specified in this model using a GUI. Hilda differs from WebML in three aspects, which we believe are important especially for large application programs.

First, WebML does not fully separate web site structure from application logic; application logic is embedded as special boxes in the web site graph [43]. Consequently, the control flow of an application is similar to programming with goto statements (links), which makes it difficult to create and maintain large programs. Further, since the application logic is tightly coupled with the web site structure, it is difficult to develop multiple web site structures for the same application logic. In contrast, Hilda supports a more structured programming model, whereby each AUnit instance only communicates with its parent and child AUnit instances. Further, Hilda uses AUnit inheritance to separate application logic from web site structure.

Second, WebML only provides a limited form of code reuse and code abstraction. Specifically, WebML does not provide a declarative way to create complex "functions", which capture complex parts of the application logic and can possibly be reused in multiple places. Instead, all complex application logic is directly embedded in the web site graph as a sequence of simple operation

units, and this sequence has to be replicated if it is used in multiple places (unless the sequence is specified non-declaratively as a simple operation unit, in which case the declarative benefits are lost). Hilda, on the other hand, supports encapsulation and code reuse using AUnits. Specifically, each AUnit is declarative, fully encapsulates its functionality, and can be reused in multiple places.

Finally, WebML does not provide support for declaratively specifying and detecting application-level conflicts. In contrast, Hilda uses the activation tree to capture the allowable operations in the current state, and uses this set of allowable operations to detect application-level conflicts.

Abstract State Machines and relational transducers are powerful approaches for describing and validating computing systems [46, 58] and there has been related work on formally specifying workflows and verifying their properties [42, 107]. Recently proposed new standards for describing various aspects related to Semantic Web Services, including a Web Services Modeling Language (WSML) [47], fit into this context as well. This work is related to Hilda in that it models application execution as a sequence of states, and declaratively specifies actions that are possible in each state. Hilda takes this work a step further by providing a complete programming language (Hilda programs are compiled into executable web applications), which is tailored to building data intensive web applications by providing features such as persistence, AUnits with sophisticated support for application conflict detection and PUnits.

Industrial standards. There is growing interest in the industry to separate business and application logic from the underlying platform technology. A major emerging standard is Model Driven Architecture (MDA) [78]. MDA defines different levels of abstraction and well-defined transformations between them. A

number of major database vendors like IBM and Oracle support MDA and data-driven application development [14, 45]. MDA is a programming methodology rather than an actual programming language. Hilda, on the other hand, is a programming language that can use the MDA programming methodology.

Other Programming Languages. LINK [28], HOP [92] and Volta [100] share very similar purpose as HILDA. They all provide a unified programming language and allow developers to build multi-tier web applications in a single language while hide the underlying multi-tier details. A major difference is that in those languages, developers need to annotate the program manually to partition the applications across tiers. The compilers of those languages generate client and server executable code statically and can not be change automatically based on performance metrics as HILDA.

LINQ [86] and Persistent C++ [13, 70] tries to solve the impedance mismatch problem by hiding the relational model as objects in a general purpose programming languages, e.g., C# and C++. HILDA is designed to be a domain specific language that are suitable for data-driven aplications. It captures all the logic and states using relational models and enable various optimization that are hard to do in a general purpose programming language.

7.2 Client-Server Partitioning for Distributed Applications

The most related area of research is Mobile Code, which aims at transforming a centralized program into a distributed architecture and utilizing resources in distributed systems [4, 62, 97, 104]. The system in [62] takes the binary code of a program and distributes the components and the procedures among a cluster

in order to optimize the communication cost. Wang et al. address the problem of partitioning programs in the context of mobile devices [104]. They represent a program in the form of a Task Control Flow Graph (TCFG), i.e., a directed graph, where each node represents a task, and each edge represents data transfer between the tasks. Their cost model includes computation time, communication time, scheduling time, and data registration time. They formulate the optimization problem as a parameterized min-cut/max-flow problem, where common parameters include buffer size, input size, command-line options, etc. The Abacus system [4] consists of a programming model and a run-time system. The proposed programming model encourages the programmer to develop data-intensive applications using small, functionally independent components or objects. The run time system automates the placement of the objects in data-intensive applications and file systems among the nodes of a cluster. The J-Orchestra [97] system partitions Java applications into distributed ones using Java RMI. By rewriting the code using Java RMI, their system can distribute components which share data in memory and thus result in finer granularity for partitioning. However, none of this work consider the concepts of consistency and conflicts for the cached table data between client and server sides. Another drawback is that the language model used by all of this work is not declarative, and therefore the efficacy of the system is limited by how programmers code the components and the procedures.

Caching data and query results at clients is a concept that has been studied in relational and object-oriented database systems. Work in this area has focused on Transactional Client-Server Cache Consistency [51, 108, 80], a technique that evaluates part of a transaction at the client by shipping it the required data. This work is concerned with guaranteeing the ACID properties of a transaction, and

proposes many different approaches such as the Avoidance Based Approach (Adaptive CallBack Locking) and the Detection Based Approach (Adaptive Optimistic Concurrency Control). However, the work assumes a predefined partition of transactions across the server and the client, and thus is complementary to what Hilda achieves.

Hybrid Shipping Architectures have been proposed to run queries in a distributed setting [101, 102]. The motivation behind these systems is that data shipping (query execution at clients) and query shipping (query execution at servers) can be done together. However, these architectures only consider the partitioning of a single read-only query. They decompose each query into operators such as join, scan and display etc. and then distribute these operations across different sites, taking into account the parallelism and communication costs. They use standard optimization techniques to achieve this. Our goal, on the other hand, is to partition queries in one transaction across the server and the client and to cache data for multiple queries.

7.3 Graphical Tools For Building Data-Driven Web Applications

Many commercial website creation tools, e.g., Dreamweaver [34], Frontpage [54] provide a WYSIWYG interface for creating web pages. Users specify page contents graphically and can see the resulting page instantaneously. However, they are mainly used for creating webpages, and the backend application server and database is developed separately not in a WYSIWYG manner. AppForge takes these systems a step further by providing WYSIWYG not only for

webpages, but also for application logic and backend database development.

Zoho Creator [30], CogHead [25], App2You [5] DabbleDB [31], and Wyaworks [109] provide developers with a form-oriented, drag-and-drop interface to build data-driven Web applications. Salesforce [90], QuickBase [87] and Instant Application Platform (IAP) [63] provides extensive solution libraries so developers can customize the applications to fit their own business requirement. While a few of these systems provide a WYSIWYG environment and most of them do not need developers to edit the database schema directly, they do not provide an abstraction for complex schemas, including n-way relationships and aggregation, as complex views includes joins, aggregations and nesting.

Ning [77] is a website that allows developers to create and customize their own social network portal. While simple customization can be performed using templates, more sophisticated customization involving new entities and relationships requires explicit programming. JotSpot [66] is a related website that extends Wiki [106] with rich structured content, forms [8], and a WYSIWYG interface. However, it is not designed for general Web applications with multiple entities and complex relationships. There are also many other enterprise tools designed to improve developer productivity, e.g., SAP Visual Composer [27] and Oracle Forms [50]. While these tools are more powerful, they are mostly targeted towards professional developers.

CASE tools such as UML [16] and WebML [19] have been developed over the past few decades to help developers to build applications. WebML extends UML with links and operations abstractions tailored specially for web applications. It provides a graphical way of specifying database schema, application logic and navigational structure of web applications and automatically generate

websites based on the graphical components specified by developers. The main difference between WebML and AppForge is that WebML separates the phases for designing the database schema and designing Web page content. Further, WebML separates the query specification from the output and hence does not provide WYSIWYG interface for creating web pages. In contrast, in Appforge, the database schema is generated implicitly, and changing the queries that populate the page contents will result in instantaneous changes in web pages, that allows users to continuously refine the query as they are constructing it.

There has been a lot of work on graphically creating SQL queries such as Query-By-Example [110] ,Visual Query Builder [15], Visual Query Language [11, 75]. While these approaches hide the SQL syntax from users, they still expose the full schema in terms of relational tables. This is especially confusing when relationships are normalized into tables as well where users are required to use joins to ”stitch” information back together [64]. In contrast, AppForge hide the complexity of the E-R and the relational models, and instead exposes a simple hierarchal Schema Navigation Menu. Another major difference is that AppForge provides a WYSIWYG experience and tightly integrate with schema generation which is not considered by these approaches.

Forms-based approaches [21, 35, 73] for query interface design have been proposed to provide users with visual tools to frame queries and to perform tasks such as database design and view definition. However, like Query-By-Example based methods, they require the users to deal with joins across multiple normalized tables, and are not truly WYSIWYG, which reduces their usability for the audience we are targeting.

In [76], an instantaneous-response interface is proposed to allow users to

allows the user to continuously refine the query as they are typing the initial query. By the time the user has typed out the entire query, the query has been correctly formulated and the results have returned. We share the same philosophy to make database more usable. AppForge extends the same WYSIWYG methodology from query formation (View Creation) to other aspects of creating web applications, e.g., schema creation, forms creation.

7.4 Shared Data Hosting Systems

Commercial relational database systems, e.g., Oracle [23], DB2 [9] and SQL Server [93], mainly target at large enterprise applications and require a sizable upfront investment, bring more complexity than is needed for ”small” web applications, and often require DBAs to maintain and administer. Our system focuses on providing a scalable data platform by using commodity hardware and software components and automating tasks for replication, failure recovery and load balancing and adding and making use of new resources without reconfiguration which could otherwise renders the system unmanageable for a large amount of applications.

Other commercial systems, e.g., BigTable [40], SimpleDB [94] and PNUTS [57], share a similar purpose as our system. The BigTable [40], is a fast and extremely large-scale column-oriented database system that is designed to scale into the petabyte data with thousands of machines. The Amazon SimpleDB [94] is a web service which provides the ability to store, process and query structured data sets using a cloud of machines at the backend. PNUTS [57] is a data storage platform for web and community applications that

combines massive scalability, support for multiple access (selection) patterns and fault tolerance. Peer-to-peer (P2P) technologies such as Distributed Hash Tables (DHTs) [88, 89, 95] and ordered tables [1, 29, 65] are designed to scale using commodity hardware and software and achieve excellent scalability and throughput performance. All the systems mentioned provide either reduced functionality or weaker consistency guarantees compared with a full fledged DBMS.

There are many previous work on building middle-ware solutions for DB clusters [18, 17, 91, 84]. The replication strategy used in our system is very similar as the RAIDb-1 proposed in [18]. They proposed replications using a redundant array of databases with commodity hardware which is analog to RAID for disks. The Sequoia project [91] is a continuation for C-JDBC [17] and provide a transparent middleware solution offering clustering, load balancing and fail-over management for any databases. None of them consider automatically creating new database replicas to restore fault tolerance after failure and they do not provide support for SLA. In [84], they designed a multi-instance database cluster solution that can handle hundreds of client databases concurrently based on a light weight adapter in middleware. Other systems, such as [72, 83], focus on replicating single database instance for load balancing and fault-tolerant. Most of such system provides weaker consistency model (e.g., snapshot isolations) than the one-copy serializability as implemented in our system.

The problem for providing QoS guarantees in shared hosting platforms has been studied in [6, 7, 71, 98]. All of those systems provide comprehensive frameworks for resource management in web servers in order to deliver predictable QoS and differentiated services. They profile applications on dedicated nodes

and use these profiles to guide the placement of the application onto shared nodes. In their system, they rely on OS resource allocations mechanisms to provide application isolation and performance guarantees at a fine grain level. In our system, DBMS is treated as a black box, the resource consumption can only be measured and controlled indirectly. We enforce the SLA for each database by admitting request based on it's SLA and reject requests if the throughput in SLA is exceeded for a database. Those system generally do not consider fault tolerance and recovery as part of the SLA as in our system.

The problem for dynamic applications placement in a cluster is studied in [68, 96]. In their work, they focused on how to dynamically place application servers among a cluster of machines such that (1) the total satisfied application demand is maximized (2) the total number of application starts and stops is minimized and (3) load is balanced across machines. In our system, we are facing a harder problem since location of a database instance, once determined, can not be changed as easy as instances of application servers. They also do not consider recovery cost as part of the SLA.

BIBLIOGRAPHY

[1] Karl Aberer. P-Grid: A self-organizing access structure for P2P information systems. *Sixth International Conference on Cooperative Information Systems (CoopIS 2001), Lecture Notes in Computer Science*, 2172:179–194, 2001.

[2] Serge Abiteboul and Nicole Bidoit. Non first normal form relations: An algebra allowing data restructuring. *J. Comput. Syst. Sci.*, 33(3):361–393, 1986.

[3] AJAX. http://en.wikipedia.org/wiki/Ajax_(programming).

[4] Khalil Amiri et al. Dynamic function placement for data-intensive cluster computing. In *USENIX 2000 Annual Technical Conference, San Diego, CA, June 2000.*, pages 307–322, 2000.

[5] App2You. http://app2you.com/site/.

[6] M. Aron. Differentiated and predictable quality of service in web server systems. In *PhD thesis, Department of Computer Science, Rice University, October 2000.*

[7] Mohit Aron, Peter Druschel, and Willy Zwaenepoel. Cluster reserves: a mechanism for resource management in cluster-based network servers. In *Measurement and Modeling of Computer Systems*, pages 90–101, 2000.

[8] Form Assembly. http://www.formassembly.com/.

[9] Chaitanya K. Baru, Gilles Fecteau, Ambuj Goyal, Hui-I Hsao, Anant Jhingran, Sriram Padmanabhan, George P. Copeland, and Walter G. Wilson. Db2 parallel edition. *IBM Systems Journal*, 34(2):292–322, 1995.

[10] Visual Basic. http://msdn2.microsoft.com/en-us/vbasic/default.aspx/.

[11] Francesca Benzi, Dario Maio, and Stefano Rizzi. Visionary: a viewpoint-based visual language for querying relational databases. *Journal of Visual Languages and Computing*, 1999.

[12] P.A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 3 edition, 1987.

[13] Alexandros Biliris, Shaul Dar, and Narain H. Gehani. Making c++ objects persistent: the hidden pointers. *Software - Practice and Experience*, 23(12):1285–1303, 1993.

[14] A. W. Brown. An introduction to model driven architecture. *The Rational Edge*, February 2004. e-zine for the Rational community.

[15] Active Query Builder. http://www.activequerybuilder.com/.

[16] Rainer Burkhardt. *UML: Unified Modeling Language*. Addison-Wesley, 1997.

[17] E. Cecchet, J. Marguerite, and W. Zwaenepoel. Cjdbc: Flexible database clustering middleware. In *In Proceedings of USENIX Annual Technical Conference, Freenix track*, 2004.

[18] Emmanuel Cecchet. Raidb: Redundant array of inexpensive databases. *Parallel and Distributed Processing and Applications*, 2005.

[19] Stefano Ceri, Piero Fraternali, and Aldo Bongio. Web modeling language (webml): a modeling language for designing web sites. In *WWW'00*, 2000.

[20] Peter P. Chen, editor. *Entity-Relationship Approach to Information Modeling and Analysis, Proceedings of the Second International Conference on the Entity-Relationship Approach (ER'81)*. North-Holland, 1983.

[21] J. Choobineh, M. V. Mannino, and V. P. Tseng. A form-based approach for database analysis and design. In *CACM, 35(2)*, 1992.

[22] Panos K. Chrysanthis and Krithi Ramamritham. Synthesis of extended transaction models using acta. *ACM TODS*, 19(3):450–491, 1994.

[23] Oracle Real Application Cluster. http://www.oracle.com/.

[24] E. F. Codd. A relational model of data for large shared data banks. *CACM*, 13(6):377–387, 1970.

[25] CogHead. http://www.coghead.com.

[26] Tm Web Commerce. Tpc benchmark http://www.tpc.org/tpcw/.

[27] SAP NetWeaver Visual Composer. https://www.sdn.sap.com/.

[28] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: Web programming without tiers. In *Submitted to ESOP 2007*.

[29] Adina Crainiceanu, Prakash Linga, Ashwin Machanavajjhala, Johannes Gehrke, and Jayavel Shanmugasundaram. P-ring: an efficient and robust p2p range index structure. In *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 223–234, New York, NY, USA, 2007. ACM.

[30] Zoho Creator. http://creator.zoho.com/index.jsp?serviceurl=

[31] DabbleDB. http://dabbledb.com/.

[32] M. Davis. Struts, an open-source mvc implementation. February 2001, http://www- 106.ibm.com/developerworks/library/j-struts/?n-j- 2151.

[33] E. W. Dijkstra. Letters to the editor: go to statement considered harmful. *CACM*, 11(3):147–148, 1968.

[34] Adobe Dreamweaver. http://www.adobe.com/products/dreamweaver/.

[35] D. W. Embley. Nfql: The natural forms query language. In *ACM Trans. Database Syst.*, 1989.

[36] C. Botev et al. Supporting workflow in a course management system. In *In Proc. SIGCSE*, 2005.

[37] D Alur et al. *Core J2EE Patterns: Best Practices and Design Strategies*. Prentice Hall, 2001.

[38] D. R. Karger et al. Haystack: A general-purpose information management tool for end users based on semistructured data. In *Proc. CIDR*, pages 13–26, 2005.

[39] E. Gamma et al. *Design Patterns - Elements of Reusable Object Oriented Software*. Addison Wesley, 1995.

[40] Fay Chang et. al. Bigtable: A distributed storage system for structured data. In *OSDI'06: Seventh Symposium on Operating System Design and Implementation*.

[41] G. Booch et al. *The Unified Modeling Language User Guide,The Addison-Wesley Object Technology Series*. Addison Wesley, 1998.

[42] H. Davulcu et al. Logic based modeling and analysis of workflows. In *Proc. PODS*, pages 25–33, 1998.

[43] M. Brambilla et al. Declarative specification of web applications exploiting web services and workflows. In *Proc. SIGMOD*, pages 909–910, 2004.

[44] M. F. Fernandez et al. Declarative specification of web sites with strudel. *The VLDB Journal*, 9(1):38–55, 2000.

[45] P. M. Deshpande et al. Model driven development of content management applications. In *Proc. COMAD*, pages 112–121, 2005.

[46] S. Abiteboul et al. Relational transducers for electronic commerce. In *Proc. PODS*, pages 179–187, 1998.

[47] S. Arroyo et al. *Web Service Modeling Ontology Primer*, June 2005. W3C submission.

[48] S. Ceri et al. Architectural issues and solutions in the development of data-intensive web applications. In *Proc. CIDR*, 2003.

[49] Evite. http://www.evite.com/.

[50] Oracle Forms. http://www.oracle.com/.

[51] Michael J. Franklin, Michael J. Carey, and Miron Livny. Transactional client-server cache consistency: alternatives and performance. *ACM Trans. Database Syst.*, 22(3), 1997.

[52] P. Fraternali. Tools and approaches for developing data-intensive web applications: A survey. *ACM Computing Surveys*, 31(3):227–263, 1999.

[53] P. Fraternali and P. Paolini. Model-driven development of web applications: The AutoWeb system. *ACM TOIS*, 18(4):323–382, 2000.

[54] Microsoft Office FrontPage. http://msdn2.microsoft.com/en-us/office/aa905421.aspx.

[55] Google Gadgets. http://www.google.com/apis/gadgets/.

[56] H. Garcia-Molina and K. Salem. Sagas. In *Proc. SIGMOD*, 1997.

[57] Community Systems Group. Community systems research at yahoo! *SIGMOD Rec.*, 36(3):47–54, 2007.

[58] Y. Gurevich. Abstract state machines: An overview of the project. In *Proc. FoIKS*, pages 6–13, 2004.

[59] A. Hayrapetyan, D. Kempe, M. Pál, and Z. Svitkina. Unbalanced graph cuts. In *European Symposium on Algorithms (ESA), Mallorca, Spain*, 2005.

[60] http://java.sun.com/j2se/1.4.2/docs /guide/plugin/developer_guide/applet_caching.html.

[61] R. Huebsch, B. N. Chun, J. M. Hellerstein, B. T. Loo, P. Maniatis, T. Roscoe, S. Shenker, I. Stoica, and A. R. Yumerefendi. The architecture of pier: an internet-scale query processor. In *CIDR*, 2005.

[62] Galen C. Hunt and Michael L. Scott. The coign automatic distributed partitioning system. In *Operating Systems Design and Implementation*, pages 187–200, 1999.

[63] Interneer. http://www.interneer.com/.

[64] H. V. Jagadish, Adriane Chapman, Aaron Elkiss, Magesh Jayapandian, Yunyao Li, Arnab Nandi, and Cong Yu. Making database systems usable. In *SIGMOD '07*, 2007.

[65] H. V. Jagadish, Beng Chin Ooi, and Quang Hieu Vu. Baton: a balanced tree structure for peer-to-peer networks. In *VLDB '05: Proceedings of the 31st international conference on Very large data bases*, pages 661–672. VLDB Endowment, 2005.

[66] JotSpot/Google. http://www.jot.com.

[67] E. G. Coffman Jr., M. R. Garey, and D. S. Johnson. Approximation algorithms for bin packing: a survey. In *Approximation algorithms for NP-hard problems*, 1997.

[68] A. Karve, T. Kimbrel, G. Pacifici, M. Spreitzer, M. Steinder, M. Sviridenko, and A. Tantawi. Dynamic placement for clustered web applications. In

WWW '06: Proceedings of the 15th international conference on World Wide Web, pages 595–604, New York, NY, USA, 2006. ACM.

[69] L. T. Kou and G. Markowsky. Multidimensional bin packing algorithms. In *IBM Journal of Research and Development. Volume 21, Number 5, 1977*.

[70] Charles Lamb, Gordon Landis, Jack Orenstein, and Dan Weinreb. The objectstore database system. *Communications of the ACM*, 34(10):50–63, 1991.

[71] C. Li, G. Peng, K. Gopalan, and T. Chiueh. Performance guarantee for cluster-based internet services. In *Performance guarantee for cluster-based internet services. In Proc. of ICDCS*, 2003.

[72] Yi Lin, Bettina Kemme, no-Martínez Marta Pati and Ricardo Jiménez-Peris. Middleware based data replication providing snapshot isolation. In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 419–430, 2005.

[73] K. Mitchell and J. Kennedy. Drive: An environment for the organized construction of user-interfaces to databases. In *Interfaces to Databases (IDS-3)*, 1996.

[74] J. Moss. Log-based recovery for nested transactions. In *Proc. VLDB*, 1987.

[75] N. Murray, N. Paton, and C. Goble. Kaleidoquery. A visual query language for object databases. In *Advanced Visual Interfaces*, 1998.

[76] A. Nandi and H. V. Jagadish. Assisted querying using instant-response interfaces. In *SIGMOD*, 2007.

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

[78] Object Management Group. *MDA Guide Version 1.0.1*, 2003. Available at http://www.omg.org/docs/omg/03-06-01.pdf.

[79] Ruby on Rails. http://www.rubyonrails.org/.

[80] M. Ozsu, K. Voruganti, and R. Unrau. An asynchronous avoidance-based cache consistency algorithm for client caching dbmss, 1998.

[81] Yahoo! Pipe. http://pipes.yahoo.com/pipes/.

[82] Facebook Platform. http://developers.facebook.com/.

[83] Christian Plattner and Gustavo Alonso. Ganymed: scalable replication for transactional web applications. In *Middleware '04: Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware*, pages 155–174, 2004.

[84] Christian Plattner, Gustavo Alonso, and M. Tamer zsu. Dbfarm: A scalable cluster for multiple databases. In *In: Proc. of the 7th ACM/IFIP/USENIX International Middleware Conference Melbourne, Australia*, 2006.

[85] Microsoft Popfly. http://www.popfly.ms/.

[86] The LINQ Project. http://msdn.microsoft.com/en-us/netframework/aa904594.aspx/.

[87] QuickBase. http://www.quickbase.com/p/home.asp.

[88] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proceedings of ACM SIGCOMM*, Aug. 2001.

[89] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware), Heidelberg, Germany*, pages 329–350, November, 2001.

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

[91] Continuent Sequoia Project. http://sequoia.continuent.org/homepage.

[92] Manuel Serrano, Erick Gallesio, and Florian Loitsch. Hop: a language for programming the web 2.0. In *OOPSLA '06: Companion to the 21st ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 975–985, New York, NY, USA, 2006. ACM.

[93] Microsoft SQL Server. http://www.microsoft.com/sql/.

[94] Amazon SimpleDB. http://www.amazon.com/.

[95] Ion Stoica, Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. In *Proceedings of the 2001 ACM SIGCOMM Conference*, pages 149–160, 2001.

[96] Chunqiang Tang, Malgorzata Steinder, Michael Spreitzer, and Giovanni Pacifici. A scalable application placement controller for enterprise data centers. In *Proceedings of the 16th international conference on World Wide Web*, 2007.

[97] E. Tilevich and Y. Smaragdakis. J-orchestra: Automatic java application partitioning. *European Conference on Object-Oriented Programming (ECOOP), Malaga, June 2002.*

[98] B. Urgaonkar, P. Shenoy, and T. Roscoe. Resource overbooking and application profiling in shared hosting platforms. In *in Proceedings of OSDI, 2002.*, 2002.

[99] V. V. Vazirani. *Approximation Algorithms*. Springer-Verlag, Berlin, 2001.

[100] Volta. http://labs.live.com/volta/.

[101] Kaladhar Voruganti, M. Tamer Ozsu, and Ronald C. Unrau. An adaptive hybrid server architecture for client caching ODBMSs. In *The VLDB Journal*, pages 150–161, 1999.

[102] Kaladhar Voruganti, M. Tamer Özsu, and Ronald C. Unrau. An adaptive data-shipping architecture for client caching data management systems. *Distrib. Parallel Databases*, 15(2):137–177, 2004.

[103] H. Wachter and A. Reuter. The contract model. *Database Transaction Models for Advanced Applications*, 1992.

[104] Cheng Wang and Zhiyuan Li. Parametric analysis for adaptive computation offloading. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, 2004.

[105] Yahoo! Widgets. http://widgets.yahoo.com/workshop/.

[106] Wiki. http://www.wiki.org/.

[107] D. Wodtke and G. Weikum. A formal foundation for distributed workflow execution based on state charts. In *Proc. ICDT*, pages 230–246, 1997.

[108] Keqiang Wu, Peng fei Chuang, and David J. Lilja. An active data-aware cache consistency protocol for highly-scalable data-shipping dbms architectures. In *CF '04: Proceedings of the 1st conference on Computing frontiers*, 2004.

[109] WyaWorks. http://www.wyaworks.com/.

[110] M. M. Zloof. Query-by-example: the invocation and definition of tables and forms. In *VLDB*, 1975.