Building Structured Web Portals: Research Challenges and General Platforms

By

Xiaoyong Chai

A dissertation submitted in partial fulfillment of

the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN-MADISON

2013

Date of final oral examination: 12/13/12

The dissertation is approved by the following members of the Final Oral Committee:

AnHai Doan, Associate Professor, Computer Sciences
Jeffrey Naughton, Professor, Computer Sciences
Jignesh Patel, Professor, Computer Sciences
Jude Shavlik, Professor, Computer Sciences
Mark Craven, Professor, Biostatistics & Medical Informatics, Computer Sciences

To my wife and parents, who supported me every step of the way.

ACKNOWLEDGMENTS

First of foremost, I would like to express my sincere appreciation to my advisor, Prof. AnHai Doan, for his continuous support and guidance for my Ph.D. study. His enthusiasm, patience, and immense knowledge helped me in all the time of research for and writing of this thesis.

Besides my advisor, I would like to thank the rest of my thesis committee: Prof. Jeffrey Naughton, Prof. Jignesh Patel, Prof. Jude Shavlik, and Prof. Mark Craven, for their time, insightful comments, and suggestions for improving my thesis. I would also like to express my gratitude to Prof. Christopher Re for serving on my preliminary exam committee and giving me valuable feedback at an early stage of this thesis.

I am also blessed to have so many great fellow students in the Database Research Group: Chen Zeng, Yeye He, Chaitanya Gokhale, Ba-Quy Vuong, Akanksha Baid, Sean Crosby, and Warren Shen, for their support and company over the past six years.

In addition, I would like to acknowledge the financial and academic support from the Department and the University. I especially thank Angela Thorp and Tonya Messer for their timely assistance during the final stage of my graduate study.

Last but not the least, I would like to thank my family for their endless support and encouragement. They always believe in me even when I doubted myself. My deepest gratitude goes to my wife, who has stood beside me all these years. Without her patience and sacrifice, I would not have been able to complete this journey.

DISCARD THIS PAGE

TABLE OF CONTENTS

Page

LIST OF TABLES vi

LIST OF FIGURES vii

ABSTRACT x

1 Introduction 1

1.1 Designing and Building a General Platform 3
1.2 Improving the Matchability of Portal Schemas 4
1.3 Supporting User Feedback 4
1.4 Contributions and Outline 5

2 Our Envisioned Platform for Building Structured Web Portals 6

2.1 Desired Capabilities of a General Platform 6
2.2 Data Storage Layer 8
2.3 Data Processing Layer 10
2.4 User Layer 11

3 Improving the Matchability of Portal Schemas 12

3.1 Introduction 13
3.2 Problem Description 15
3.3 Schema Matchability 16
3.3.1 Defining Schema Matchability 16
3.3.2 Estimating Schema Matchability 17
3.4 Analyzing Schema Matchability 19
3.4.1 Common Matching Mistakes 21
3.4.2 Generating a Matchability Report 26
3.5 Improving Schema Matchability 27
3.5.1 SE Transformation Rules 27
3.5.2 Searching for Optimal SE Sequences 28

Page

3.6 Empirical Evaluation . . . 32
3.6.1 Experimental Setup . . . 33
3.6.2 Utility of the Matchability Concept . . . 34
3.6.3 Usefulness of Matchability Reports . . . 35
3.6.4 Automatic Schema Revision . . . 37
3.6.5 Multi-Appearance Representation . . . 39
3.6.6 Sensitivity Analysis . . . 40
3.7 Summaries . . . 42

4 Opening Up Structured Web Portals for User Feedback . . . 43

4.1 Introduction . . . 43
4.2 The Swiki Approach . . . 47
4.3 Creating the Initial Portal . . . 48
4.3.1 Creating a Structured Database G . . . 49
4.3.2 Creating Views over Database G . . . 58
4.3.3 Converting Views to Wiki Pages . . . 60
4.4 Managing User Contributions . . . 66
4.4.1 What Can Users Edit? . . . 66
4.4.2 Infer and Execute Structured Edits . . . 67
4.4.3 Propagate Structured Edits . . . 71
4.5 Managing Multiple Users and Machine . . . 71
4.6 Empirical Evaluation . . . 72
4.7 Summaries . . . 76

5 Efficiently Incorporating User Feedback into IE/II Programs . . . 78

5.1 Introduction . . . 78
5.2 Syntax and Semantics . . . 83
5.2.1 The xlog Language . . . 84
5.2.2 The hlog Language: Syntax . . . 85
5.2.3 The hlog Language: Semantics . . . 86
5.2.4 Extending hlog to Handle II . . . 92
5.3 Executing hlog Programs . . . 92
5.3.1 Storing Provenance Data . . . 93
5.3.2 Storing and Incorporating User Feedback . . . 93
5.4 Optimizing hlog Execution . . . 95
5.4.1 Incremental Execution . . . 96
5.4.2 Incremental ID Maintenance . . . 102
5.4.3 Improved Concurrency Control . . . 103

Page

5.5 Empirical Evaluation . . . 106
5.5.1 Incremental Execution . . . 108
5.5.2 Concurrency Control . . . 109
5.6 Summaries . . . 112

6 Building a General Platform . . . 113

6.1 Building the Data Storage Layer . . . 113
6.2 Building the Data Processing Layer . . . 117
6.3 Building the User Layer . . . 118
6.4 Building Platform Administration Interface . . . 120
6.5 Case Studies . . . 120
6.5.1 Building a Limnology Research Portal . . . 122
6.5.2 Generating Natural Language Profiles for Musical Artists . . . 124
6.6 Observations . . . 127

7 Related Work . . . 130

7.1 Building Structured Web Portals . . . 131
7.2 Matching Portal Schemas . . . 132
7.3 Opening Up Structured Web Portals for User Feedback . . . 132
7.4 Incorporating User Feedback Into IE/II Programs . . . 133
7.5 Observations on Industrial Practices . . . 134

8 Conclusions . . . 137

8.1 Contributions . . . 137
8.2 Future Directions . . . 138
8.3 Closing Remarks . . . 139

Bibliography . . . 140

APPENDIX Basic Relational Actions . . . 146

DISCARD THIS PAGE

LIST OF TABLES

Table Page

3.1 Conditions, reasons, and suggestions used in report generation 25

3.2 Classification of SE transformation rules 28

3.3 Real-world domains in our experiments 33

3.4 Compilation of report snippets generated by mSeer 36

4.1 Basic relational actions on V_d, V_s, G_d and G_s 68

4.2 Basic ER actions 69

5.1 Incremental properties of DBLife operators. 107

6.1 Statistics on Limnology research portal development. 123

6.2 Statistics on musical artist portal development. 127

DISCARD THIS PAGE

LIST OF FIGURES

Figure Page

2.1 High-level architecture of a general platform for building structured Web portals. . . . 8

3.1 An example of schema perturbation 18

3.2 A sample matchability report 20

3.3 A matching scenario in a global system 23

3.4 The procedure that RevSearcher uses to find the best set of rules in each iteration. . . 31

3.5 Matching accuracy of schemas produced by mSeer vs. that of the original schemas . 38

3.6 Improvement achieved vs. ceiling across all domains 39

3.7 Accuracy with and without multi-appearance representation 39

3.8 Change in matching accuracy with respect to (a) size of synthetic workload, and (b) number of data instances 40

3.9 Matching accuracy with a new matching system 41

4.1 An example to illustrate the Swiki approach. 44

4.2 The Swiki architecture 48

4.3 (a) A snapshot of the ER graph G, (b) a sample view schema, (c) a sample data of the above view, and (d) how the above sample data is exported into a wiki page in the s-slot wiki language. 49

4.4 Tables for *person* entities: (a) a single table for all attributes, and (b)-(d) vertical partitions of the single table. 52

4.5 Examples of transaction-time tables for *person_organization*: (a) before entity with id=1 is updated, and (b) after entity with id=1 is updated. 53

Figure Page

4.6 An example of attribute tables for *organization* of entity type *person*: (a) A_m, (b) A_u, and (c) A_p. 55

4.7 Examples of meta tables: (a) $meta\_entity$, (b) $meta\_relation$, and (c) $meta\_attribute$. 57

4.8 Generating wiki page W from view data V_d 64

4.9 Generating $\mathcal{S}_{ER}$ from V_d and V_d' 70

4.10 Time to request a wiki page and distribution of page size. 73

4.11 Time to process user edits on a wiki page. 74

4.12 User performance on several editing tasks. 75

5.1 An illustration of our approach: given the set of data sources in (a), a developer U writes the IE/II program P in (b) to extract titles and abstracts of talks from the data sources; next U writes the user feedback rules in (c) to specify which parts of P users can edit and via which UIs; the system then executes P and exposes the specified data portions for users to edit, as shown in (d). 81

5.2 Execution graph of the hlog program in Figures 5.1.b-c, as reproduced from Figure 5.1.d. 87

5.3 Example provenance tables for a case where operator p takes as input two tables and produces as output a table with three tuples. 92

5.4 An example of incorporating user feedback. 94

5.5 An example of incremental-update property specification. 100

5.6 Generic incremental update algorithm. 101

5.7 Transaction runtime in different DBLife versions. 109

5.8 Comparison of transaction throughput. 110

5.9 Comparison of transaction response time. 111

5.10 Space consumption under different concurrency control policies. Numbers in brackets give the number of active transactions at a certain time. 111

6.1 High-level architecture of the general platform. 113

Figure Page

6.2 An example of retrieving unstructured text data from the Web. 115

6.3 An example of storing and retrieving versioned text data. 116

6.4 An example of creating and querying entities. 117

6.5 An example of entity homepage. 120

6.6 Web-based administration tool. 121

6.7 An example of publishing entity homepages. 121

6.8 Homepage of Lake Mendota. 123

6.9 Homepage of Michael Jackson. 125

6.10 Record interface for editing artist details section. 126

6.11 Wiki interface for editing wiki text section. 126

BUILDING STRUCTURED WEB PORTALS: RESEARCH CHALLENGES AND GENERAL PLATFORMS

Xiaoyong Chai

Under the supervision of Professor AnHai Doan

At the University of Wisconsin-Madison

Over the past decade, the rapid growth of Web communities has led to an increasing number of structured Web portals being built. With their rising popularity, the value of a general platform that developers can use to quickly build portal applications has been recognized. Although tools and systems have been developed in commercial domains, little on their design and development has been published. The goal of this dissertation is thus to explore how to design and build an efficient general platform for developing structured Web portals. Towards this goal, we first examine the capabilities of a general platform and describe its high-level architecture. Next, we study three key research challenges that often arise in building Web portals: (1) how can we analyze and revise a portal schema, which serves as a uniform interface for storing and querying portal data, to make it easier to match, (2) how can we "open up" a portal so that users can easily provide feedback to improve quality of portal data, and (3) how can the portal incorporate user feedback automatically and efficiently? Finally, we developed a general platform and deployed it to build structured Web portals for two real-world applications. We describe our implementation of the platform, conduct case studies of the applications built, and discuss the lessons learned about the utility and limitations of the platform.

AnHai Doan

ABSTRACT

Over the past decade, the rapid growth of Web communities has led to an increasing number of structured Web portals being built. With their rising popularity, the value of a general platform that developers can use to quickly build portal applications has been recognized. Although tools and systems have been developed in commercial domains, little on their design and development has been published. The goal of this dissertation is thus to explore how to design and build an efficient general platform for developing structured Web portals. Towards this goal, we first examine the capabilities of a general platform and describe its high-level architecture. Next, we study three key research challenges that often arise in building Web portals: (1) how can we analyze and revise a portal schema, which serves as a uniform interface for storing and querying portal data, to make it easier to match, (2) how can we "open up" a portal so that users can easily provide feedback to improve quality of portal data, and (3) how can the portal incorporate user feedback automatically and efficiently? Finally, we developed a general platform and deployed it to build structured Web portals for two real-world applications. We describe our implementation of the platform, conduct case studies of the applications built, and discuss the lessons learned about the utility and limitations of the platform.

Chapter 1

Introduction

In recent years we have witnessed a growing number of structured Web portals, which extract and integrate structured data from a multitude of disparate Web sources, being built in both academia and industry. Examples include Citeseer [48], YAGO-NAGA [59], DBLife [36], Freebase [5], to name just a few. These portals are called structured Web portals because their contents are backed up an underlying database, and the database is continuously updated using automatic techniques that gather and filter relevant information from disparate Web data sources. These Web portals are appealing because they offer users powerful capabilities of searching, querying, and aggregating information that is otherwise scattered over the Web. These capabilities make structured Web portals valuable for a wide variety of domains, ranging from scientific data management to end-user communities on the Web.

Today many of such portals are built by applying information extraction and integration techniques (henceforth *IE/II techniques* for short) to unstructured text (e.g., Web pages of news articles), semi-structured data (e.g., Wikipedia infoboxes), as well as structured data (e.g., product feeds from vendors). The data obtained is typically structured as *entities* and *relationships* among the entities, and the entities and relationships are then made available to portal users via a variety of user services. Consider for example DBLife [36], a structured Web portal for the database research community. It continuously crawls Web data sources, such as researcher home pages, conference pages, and extracts structured data about interesting entities in the domain (such as researchers, publications and organizations) and relationships among them (such as who *authored* which publication and who is *affiliated with* which organization). It then constructs an entity-relationship (ER) graph from these extracted entities and relationships, and provides a variety of user services for

portal users to exploit the graph. As an example of its user services, DBLife automatically creates so-called entity *superhomepages* (Web pages that aggregate the extracted and inferred information about an entity and its relationships with other entities). For example, for each researcher, it creates a superhomepage that lists biographical information of the researcher, publications, conference services, and recent talks. DBLife then allows users to search for and browse these superhomepages via its Web interface.

Despite the rising popularity of structured Web portals, today there is a lack of consensus on how to build such portals. Although tools and systems have been developed in commercial domains, little on their development has been published. To address this problem, this dissertation explores how to design and build an efficient general platform that enables developers to build structured Web portals quickly and to maintain them easily over the lifetime of the portals.

Toward this goal, we first study what capabilities a general platform should have, and what its architecture should be to realize these capabilities.

Next, as we will see in Chapter 2, many research challenges arise from building a general platform. In this dissertation, we study three of them in detail. The first problem we study is how to make the portal schema, which serves as a uniform interface for storing and querying portal data, easier to match. The problem arises when we need to integrate data from multiple data sources, which are usually heterogeneous. After acquiring data (e.g., through crawling and querying) from those data sources, we often need to convert the data into a uniform format before the data can be further used. To achieve this, a structured portal often needs a uniform schema (referred to as *portal schema* from here forth) with which the schemas of the disparate data sources are matched so that data obtained from those data sources can be transformed and aggregated accordingly. Different designs of the portal schema may result in different amounts of effort in matching the schemas. Thus how can we assist developers in designing the portal schema to make schema matching easier?

The second and third problems we study are how to allow users to easily provide feedback on the portal data and how to enable the portal to efficiently incorporate the feedback. These problems arise when we want to improve the quality of IE/II results. Automatic IE/II are inherently

difficult and are thus error-prone. One way to improve the quality of the IE/II results is to "crowd-source" the work, for example, by asking a multitude of users to help detect and correct errors. To encourage users to provide feedback on the IE/II results while they are using the portal, the portal should provide the users means to easily edit the data. The questions then are (1) how can we "open up" a portal so that once a user finds an error, she can easily provide feedback (e.g., by submitting the correction via a form user interface), and (2) how can the portal incorporate the feedback automatically and efficiently? In Sections 1.1-1.3 below, we elaborate on these challenges.

1.1 Designing and Building a General Platform

One way to study the capabilities of a general platform is through analyzing how typical structured Web portals – the products of the platform – function.

A typical structured Web portal works as follows. It gathers data from various data sources (e.g., websites, local or remote databases and file systems) in its application domain. If the data from a data source is unstructured, the portal needs to extract structured data from the obtained data. Moreover, the data from a contributing data source most likely follows its own schema which was designed to meet it own application needs. Thus in the next step, the portal matches the schemas of the data sources with its own, transforms the structured data according to the matching results, and aggregates the transformed data so that in the end all the data conforms to the portal schema. With the data integrated, the portal provides users services such as keyword search and browsing to explore and exploit the data. Finally, since the work of extracting and integrating structured data is mainly conducted by automatic programs, and automatic information extraction (IE) and information integration (II) are error-prone, we often want to leverage user feedback to improve the quality of IE/II results. To leverage user feedback, the portal needs to provide means for users to submit edits on the data deemed incorrect.

As we can see, to build such a portal, developers often need to accomplish many tasks, including designing the uniform portal schema, writing IE/II programs to collect and structure the data according to the portal schema, implementing various user services, and empowering the portal with the ability of taking in user feedback and automatically incorporating the feedback. Each of

these tasks is non-trivial, and accomplishing the tasks from scratch is labor-intensive and error-prone. Since these activities are common in building structured Web portals, a platform that allows developers to readily accomplish them is invaluable.

1.2 Improving the Matchability of Portal Schemas

As mentioned above, a structured Web portal acquires and aggregates its data from a multitude of data sources, which are most likely heterogeneous. To integrate the data, the portal usually adopts a uniform schema into which data from the sources is transformed. Before the transformation, the schemas of the data sources and the uniform schema must be matched. In the data integration literature, such a uniform schema is commonly referred to as *mediated schema* [74]. In the context of building structured Web portal, we call the schema *the portal schema*. How much effort developers need to match the schemas is largely decided by how easily the *semantic matches* (e.g., attribute *location* at a data source is equivalent to attribute *address* at the portal) can be found. In Web-scale portal applications, schema matching is often performed repeatedly (e.g., when integrating data from hundreds of data sources), continuously (e.g., when the schema at a data source evolves), and semi-automatically (e.g., matches are first computed by some automatic solutions and then are verified by human analysts). Thus making the portal schema easy to match is highly desirable.

1.3 Supporting User Feedback

Structured Web portal applications increasingly employ IE/II programs to infer structures from unstructured data. Since automatic IE/II are inherently imprecise, such programs often make many mistakes, and thus can significantly benefit from user feedback. Today, however, there is no good way for users to easily provide feedback and for portals to automatically process the feedback. A common practice is to provide users with an email address or a Web form and ask the users to submit the feedback that way. To a great extent, this approach discourages users from providing feedback because it incurs much effort from the users. On the developers' side, they need to

interpret the submitted user feedback, manually examine the program internals to locate and fix the error. This is usually a slow, error-prone, and frustrating process. Besides, there can be a gap of days or even weeks between the time the user feedback is submitted and the time corrections are fully implemented. Therefore, a platform that enables portals to support entering user feedback easily and incorporating it automatically would greatly improve user experience and reduce developers' work.

1.4 Contributions and Outline

In addressing the challenges above, we make the following contributions in this dissertation:

- We study the capabilities of a general platform, and describe its high-level architecture.
- We describe the problem of analyzing and revising portal schemas to make them easier to match, and present a promising solution.
- We present a framework for supporting user feedback, discuss various challenges in realizing the framework, and describe our solutions.
- We prototype a general platform, and evaluate it in the context of real-world applications.

The rest of the dissertation is organized as follows. Chapter 2 describes our envisioned platform for building structured Web portals. Chapter 3 presents our solution to analyzing and revising portal schemas to improve their matchability. Chapter 4 illustrates how we can enable user feedback to a portal and manage it in the underlying database. Chapter 5 describes our solution for developers to easily write IE/II programs amenable to user feedback and for IE/II programs to automatically process such feedback. Chapter 6 describes our implementation of the platform, the choices we made along the way, and the case studies we conducted to evaluate the platform. Chapter 7 reviews existing approaches to building structured Web portals and discusses related work in addressing the problems studied in Chapters 3-6. Finally, Chapter 8 summarizes and discusses directions for future research.

Chapter 2

Our Envisioned Platform for Building Structured Web Portals

In this chapter we briefly describe our envisioned platform for building structured Web portals, to set the stage for discussing the research challenges in the next three chapters. In Chapter 6 we return again to the issue of general platform and discuss in detail how we have implemented and evaluated such a platform.

2.1 Desired Capabilities of a General Platform

As we briefly described in Section 1.1, a general platform for building structured Web portals should possess a few capabilities that enable developers to quickly build and deploy a portal application. Here we elaborate on these capabilities:

Support for writing declarative IE/II programs: A structured portal collects, transforms, and aggregates structured data from a multitude of data sources. Since data at those data sources may be unstructured, IE/II programs are heavily used in the backend of the portal to process the data. In developing these programs, developers often use scripting languages such as Perl and Python to implement IE/II operations, then wire them together into programs. In non-trivial applications, developers often spend a lot of time tuning the implementation, e.g., by building index structures to speed up the IE/II processes. Developing IE/II programs this way is tedious and time-consuming, and the resulting programs are hard to interpret, maintain and improve. A better solution would be to allow developers to declaratively specify the IE/II steps, and leverage platform-provided IE/II operators and index facilities to implement these steps. This way the developers can focus more on IE/II logic instead of mundane implementation details. To realize this solution, the platform

should offer a declarative language that developers can use to write IE/II programs and a library of generic operators that they can leverage to accomplish common IE/II tasks.

Support for managing program execution: Another benefit of providing a declarative language for writing IE/II programs is that the developers can leave the task of figuring out how to efficiently execute the programs to the platform. In other words, the platform should be able to analyze and optimize program execution. Furthermore, IE/II programs are usually long running processes, which may take hours or even days to execute. To ensure timely data processing, multiple IE/II programs may be scheduled to run concurrently. Thus it is important for the platform to support concurrent execution of multiple programs, and to ensure the consistency and integrity of the data in the case of execution failure or system crashes.

Support for implementing common user services: Like other Web services, a structured Web portal makes its data available to users via user services such as browsing and keyword search. Although a variety of software tools are available to bootstrap implementation, developers still need to spend much time adapting and integrating the tools. Since domain concepts are typically represented as entities and relationships in portal applications, the platform can automate much work on building user services assuming data takes those forms. Take keyword search for example. By knowing what the entities are and how their data is stored, the platform can automatically index the entities and support keyword search over them right away. The entity superhomepage we described in Chapter 1 is another example where a great amount of automation can be achieved by the platform and only a little customization is required from the developers. Therefore, the platform should provide modules that implement common user services and make the modules easy for developers to customize for their portal applications.

Support for leveraging user feedback: Supporting user feedback as we described earlier requires a significant amount of effort from developers. First, the portal should provide users easy-to-use user interfaces so that they can immediately and directly make edits. Second, once a user enters an edit, the portal should be able to interpret the edit, and incorporate it as an update to the underlying structured database. In some cases, the portal may need to propagate the edit to update the data that was computed from the data user edited. Furthermore, to prevent unauthorized and

malicious users from modifying the data, the portal needs the abilities of authenticating users and enforcing access control. Finally in the case of incorrect user feedback, the portal should be able to recover from the error by undoing the changes. As we can see, the platform should provide all these functionalities so as to support development of "user-feedback-aware" portals.

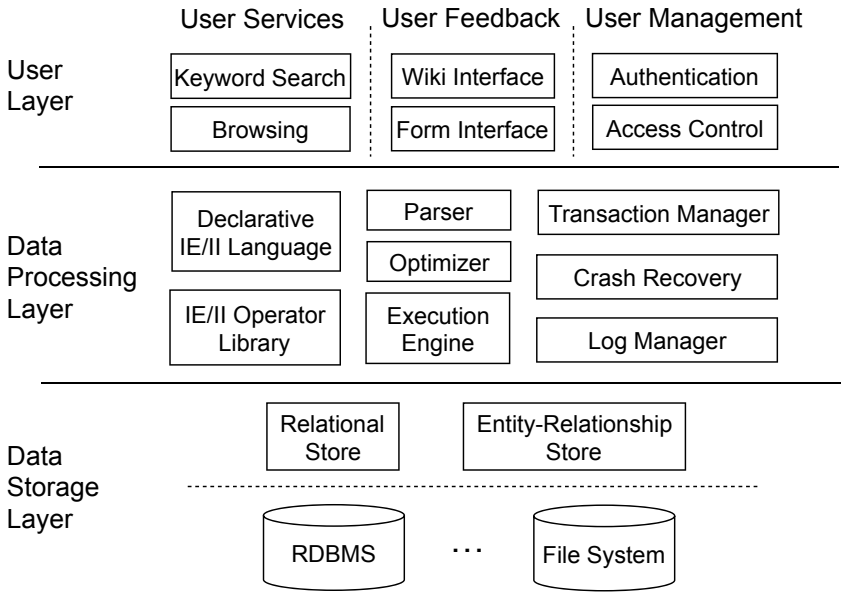


Figure 2.1: High-level architecture of a general platform for building structured Web portals.

Figure 2.1 shows the architecture of the platform we envisioned. Bottom-up, it consists of three layers: data storage layer, data processing layer, and user layer. Below we describe each layer in detail.

2.2 Data Storage Layer

This layer stores portal data. Two data models are provided: a relational model and an ER model, as we observed that both of them play important roles in structuring portal data.

The relational model represents data objects as relations; this simple and uniform representation enables efficient data storage and query processing, and allows us to leverage existing RDBMS techniques and systems. In many real-world IE/II applications, we found that the relational model is also suitable for modeling the inputs and outputs of IE/II operators. For example, suppose we

have a collection of HTML pages and we want to extract titles from these pages. To do that, we can implement an IE operator *extractTitle*, whose input can be modeled as a relation *HTML-Pages*(*pageId*, *content*), and whose output can be modeled as a relation *titles*(*pageId*, *title*). Modeling inputs and outputs of IE/II operators as relations allows developers to compose the operators with relational operators in the programs. This also makes it easier for us to provide automatic optimization of program execution in the platform as in traditional RDBMS.

As we discussed in Section 1, in many portal applications, entities and relationships are often explicitly modeled and represented. As such developers often want the ability of creating and manipulating them directly without casting them back and forth from relations. To facilitate these activities, we support an ER model in the platform. The benefits are three-fold. First, the ER model has the semantics of entities and relationships built in. It enables developers to declaratively model domain concepts as entities and relationships, and to leave the work of creating, storing and retrieving them to the platform. In contrast, using a relational model, developers have to structure the entities and relationships into relational tables, and bear in mind how the data is structured and stored in the tables when writing programs to manipulate the entities and relationships. Second, by delegating the task of managing entities and relationships to the portal, we can provide generic and automatic user services as an integrated part of the platform, as we discussed in Section 2.1. Third, in supporting user feedback, it is important to keep track of the before and after values of a data object a user edited so that we can recover the data if the edit is later deemed incorrect. To achieve this, we need a data model that is able to support storing, updating and querying historical values of a data object, in other words, a temporal data model. Creating and managing temporal information correctly, however, is a time-consuming and error-prone task for developers. With an ER model, the platform can extend entities and relationships with temporal attributes and manage them automatically (see Chapter 4 for details). Thus we support an ER model in addition to the relational model in the platform. In Figure 2.1, the relational store and the ER store are two logical stores that adopt these two data models, respectively.

Beneath the two logical stores are physical storage systems. RDBMS and file system are the two examples shown in Figure 2.1. The reason to support multiple storage systems is as follows.

The data flowing through a portal application may take different forms: it may be unstructured text (e.g., Web pages crawled from Web sources), or structured data derived from it. Different storage systems are suitable for storing different forms of data. For example, a portal application may crawl Web pages every day and save all snapshots of the crawled pages to answer historical queries. Since consecutive snapshots of a page may differ by only a little (e.g., a departmental seminar page is only updated once in a while to list new talks), a storage system that can incrementally store text data is preferable. Thus in this layer, we assume multiple storage systems may be used.

2.3 Data Processing Layer

This layer is responsible for data processing, in particular, IE/II program execution. A declarative language with a built-in libary of IE/II operators is needed to make it easy for developers to write IE/II programs. We observed from building DBLife [36] that relational operations, such as select, project and join, are frequently carried out in IE/II programs. Thus the language should also make those relational operators available to developers. One way to provide such a declarative language is by extending SQL [76], which is a query language that has relational operators built in and that developers are most likely familiar with.

The various components on the data processing layer in Figure 2.1 play similar roles to their counterparts in an RDBMS. For example, the parser parses an IE/II program into a tree of relational and IE/II operators, which is then optimized by the optimizer and evaluated by the execution engine.

The crash recovery module is worth noting. As we mentioned earlier, an IE/II program is usually long-running, possibly taking hours to execute. If the system crashes in the middle of program execution, re-executing the program from scratch would be time consuming. To reduce the overhead of re-execution, we allow developers to partition the program into blocks, where each block is executed atomically (equivalent to a database transaction). In the case of system crash, the recovery module, together with the log manager, enables the system to resume execution from where it left off (at the granularity of transaction) after the system recovers from the crash.

2.4 User Layer

This layer provides support for users to exploit portal data as well as support for users to provide feedback into the portal. While different portal applications may require different user services, some services are needed by most applications. Examples include keyword search, structured querying, and browsing. Thus we include these generic user service modules on this layer.

The user feedback module enables the developers to quickly deploy user feedback functionalities. In the frontend, it should support common user interfaces, such as widely used form interfaces and increasingly popular Wiki interfaces, to allow users to enter their feedback easily. In the backend, the module should be able to translate user feedback and incorporate it into the underlying structured database automatically.

Finally, the authentication and access control component enables the portal to verify the identity of a user, and allows portal administrators to carry out administration activities such as specifying what portion of the portal data the user can see and what portion of data the user can edit.

Chapter 3

Improving the Matchability of Portal Schemas

A structured Web portal extracts and integrates data from a multitude of autonomous data sources, which may be structurally and semantically heterogeneous. To address these heterogeneity problems, portal developers often need to manually match the schemas used by the data sources with the schema used by the portal. This is a tedious, time-consuming, and not scalable process. Hence much work has focused on developing semi-automatic techniques to find the matches accurately and efficiently. In this chapter we consider the complementary problem of *improving the portal schema*, to make finding such matches easier. Throughout the lifetime of a portal, the portal schema may be matched with many data source schemas. Thus the specific question we want to answer is *can the developer analyze and revise the portal schema in a way that preserves its semantics, and yet makes it easier to match with in the future.*

In our work [22], we provided an affirmative answer to the above question, and outlined a promising solution direction, called mSeer. Given a portal schema S and a matching tool M, mSeer first computes a matchability score that quantifies how well S can be matched against using M. Next, mSeer uses this score to generate a matchability report that identifies the problems in matching S. Finally, mSeer addresses these problems by automatically suggesting changes to S (e.g., renaming an attribute, reformatting data values) that it believes will preserve the semantics of S and yet make it more amenable to matching. In this chapter, we present the details of the work, including extensive experiments over several real-world domains that demonstrate the promise of the proposed approach.

3.1 Introduction

Building structured Web portals essentially is a Web data-integration process. Solving structural and semantic heterogeneity problems accurately and efficiently during the process has been a long-standing challenge in the database and AI communities. The main integration approaches (whether they employ virtual integration, data warehouses, or information exchange via messaging) rely on development of a uniform schema and mappings between the uniform schema and the schemas of individual data sources [74]. This methodology also applies to building structured portal applications. In the following, we describe our work in the boarder data-integration context. We refer to this uniform schema used by the portal as *mediated schema*, and the schemas used by the data sources as *source schemas*.

To create the required mappings, a data-integration system uses a set of *semantic matches* (e.g., *location* = *address*) between the mediated schema and the source schemas. Creating such matches is well-known to be laborious and error-prone. Consequently, many semi-automatic schema matching solutions have been proposed. Much progress has been made (see [44, 74] for recent surveys), and schema matching has become a research area on its own. No satisfactory solution, however, has yet been found, and the high cost of finding the correct semantic matches continues to pose a bottleneck for the widespread deployment of data-integration systems.

To address this problem, we propose to open another attack direction, by considering the complementary problem of *revising the mediated schema to improve its matchability*. Specifically, when creating the mediated schema S, can a developer P analyze and revise S in such a way that preserves S's semantics, and yet makes it easier to match with in the future? The ability to do this can prove quite helpful in many common integration scenarios, such as those given below.

Example 3.1 A developer P often must add new data sources to an existing data integration system I. To do so, P must match the schemas of the new sources with S, the mediated schema of I, using a matching tool M.

As another example, following recent trends of providing Web-based services, many integration systems (especially those in scientific domains) are being "opened up", so that members of the

user community can easily add new data sources via a GUI (e.g., [72]). To add a source T, a user U must eventually invoke a matching tool M (provided at the system site) to match T's schema with the mediated schema S, then sift through the results to fix the incorrect matches.

As yet another example, developers often "compose" integration systems, i.e., take an integration system I, treat it as a single source, then integrate it with a set of other sources to build a higher-level integration system. In such cases, the mediated schema S of I will often be matched with other mediated schemas.

In all of the above cases, if the target mediated schema can be designed to be more amenable to matching, then it can be matched with new schemas more accurately and quickly. The problem of improving the matchability of mediated schemas is therefore appealing. But it is unclear exactly how this problem should be attacked.

A key contribution of this chapter is that we provide such an attack plan. Specifically, we decomposed the above problem into three well-defined subproblems. For each subproblem we then identified the main challenges and provided initial solutions. Finally, we demonstrated the promise of the approach, using extensive experiments on real-world data sets. Our work therefore can help to motivate further research in this novel approach to schema matching.

In our work, we focus on improving 1-1 matching (e.g., *location* = *address*) for relational mediated schemas, a common scenario in practice [74]. Besides its conceptual simplicity, 1-1 matching allows us to focus on analyzing the fundamental reasons for matching errors and thus provides a good starting point. We believe the direction we open and the solution we propose can be extended to tackle more complex matches and data representations.

The first subproblem we consider is how to define the notion of *matchability score*, which quantifies how well the mediated schema S matches future schemas using a given matching tool. Such a score has not been proposed before, and estimating it is a difficult challenge. To address this challenge, we propose to employ a synthetic workload W that approximates the set of future schemas and is generated automatically from S.

Using the above notion of matchability score, we then define and address the second subproblem: analyze different types of matching mistakes, and show how to produce a report that identifies

potential matching mistakes of S. Given this report, a developer P can already revise S to address the mistakes.

Manually finding good revisions, however, is difficult and tedious. Hence, in the final subproblem, we consider how to automatically discover a good set of revisions, which can then be presented to P in form of a revised schema S^*. Developer P is free to accept, reject, or modify further these suggested revisions.

In summary, we make the following contributions:

- *Introduce the novel problem of analyzing and revising mediated schemas to improve their matchability.*
- *Describe a clear decomposition of the above problem into three well-defined subproblems:* estimating "matchability" of a mediated schema, producing a report that identifies potential matching mistakes of a mediated schema, and automatically discovering a good set of schema revisions (to improve matchability).
- *Identify the key challenges underlying these subproblems, and provide initial solutions.* These include a way to approximate the set of future schemas, an analysis of reasons for incorrect matching, a method to identify these reasons, and an algorithm to efficiently search for the most effective schema revisions.
- *Establish the promise of the approach via extensive experiments over four real-world domains with several matching systems.* The results show that we can reveal fundamental reasons for incorrect matches and can revise mediated schemas to substantially improve their matchability.

3.2 Problem Description

Suppose developer P has created an initial version of the mediated schema S. Then to help P revise S, we envision providing three services: computing a matchability score, generating a

report of potential matching mistakes, and suggesting schema revisions. For simplicity, we will call a system that provides these services mSeer (shorthand for "match seer").

As a start, P can simply ask mSeer to compute a score that quantifies how well S can be matched in the future, using M. This requires relatively little effort from P (just supplying S and M), and yet can already prove quite useful. For example, if the matchability score is low, then P may consider replacing matching tool M, or allotting more time for matching activities (in anticipation of having to correct more matching mistakes than initially expected).

Next, P can ask mSeer to generate a report that describes the potential matching mistakes and makes high-level suggestions for fixing them. P can then use the report to revise S. At the minimum, the report can alert P of "obvious problems" (e.g., two attributes with almost identical names and very similar data values) that are hard to spot in a large mediated schema, thus allowing P to quickly fix them. But it can do much more. Section 3.6 shows how such reports can also identify non-obvious, yet important potential problems for matching.

Finally, even if P recognizes potential matching problems, it is often still far from obvious how best to revise S, given the large number of potential revisions, and the complex interaction among them. To address this problem, P can ask mSeer to suggest a revision of S. mSeer then searches a space of schemas judged to be semantically equivalent to S, to produce a schema S^* that has higher matchability than S. P can then accept, reject, or revise S^*.

We now describe the three mSeer services in detail.

3.3 Schema Matchability

In this section we introduce *schema matchability* and show how to estimate it.

3.3.1 Defining Schema Matchability

Recall from Section 3.2 that our goal is to improve the matchability of the mediated schema S with respect to a matching tool M. A reasonable way to interpret this notion of matchability is to say it measures *on average* how well S can be matched with future schemas, using M.

Specifically, let $\mathcal{T} = \{T_1, \ldots, T_n\}$ be the set of all the future schemas that will be matched against S (of course, we often do not know $\mathcal{T}$), and $m(S, \mathcal{T}, M)$ be the matchability score of S w.r.t. $\mathcal{T}$ and M. Then we can write

$$m(S, \mathcal{T}, M) = \big[\sum_{T_i \in \mathcal{T}} accuracy(S, T_i, M) \big] / n \tag{3.1}$$

where $accuracy(S, T_i, M)$ is the accuracy of matching S with T_i using M.

While in principle any measure of matching accuracy can be used, we will use F_1, a popular measure [39, 74], to define $accuracy(S, T_i, M)$. Specifically, suppose that applying M to schemas S and T_i produces a set of matches O. Then the accuracy of matching S and T_i using M is $accuracy(S, T_i, M) = 2PR/(P + R)$, where precision P is the fraction of matches in O that are correct, and recall R is the fraction of correct matches that are in O.

In addition to matchability, we also define the notion of *matching variance*, denoted as $v(S, \mathcal{T}, M)$, to capture the variance in the accuracy of matching S with future schemas:

$$v(S, \mathcal{T}, M) = \big[\sum_{T_i \in \mathcal{T}} (m(S, \mathcal{T}, M) - accuracy(S, T_i, M))^2 \big] / n.$$

Our goal will be to revise S to maximize its matchability (breaking ties among revisions by selecting the one that produces the lowest variance). However, computing schema matchability and variance as defined above requires knowing the future schemas T_i as well as the *correct* matches between these schemas and S (without which we cannot compute precision P and recall R). This is rarely possible. Hence, we show how to estimate schema matchability and variance using synthetic matching scenarios.

3.3.2 Estimating Schema Matchability

We estimate schema matchability by adapting a technique proposed in the recent eTuner work [78]. eTuner attacks a very different goal, namely how to tune a matching system to maximize accuracy. It however also faces the problem of finding $\mathcal{T} = \{T_1, \ldots, T_n\}$, the future schemas that will be matched with S. eTuner solves this problem by applying a set of common *transformation*

EMPLOYEES

id	first-name	last-name	salary
1	Mike	Brown	42,000
2	Jean	Laup	64,000
3	Bill	Jones	73,000
4	Kevin	Bush	36,000

(a)

EMP

id	name	salary
1	Mike Brown	42K
2	Jean Laup	64K
3	Bill Jones	73K
4	Kevin Bush	36K

(b)

Figure 3.1: An example of schema perturbation

rules to the schema and data of S, in essence randomly "perturbing" S to generate a collection of synthetic schemas $V = \{V_1, \ldots, V_m\}$.

For example, suppose that S consists of the sole table *EMPLOYEES* in Figure 3.1.a. Then eTuner can apply the rule "abbreviating a name to the first three letters" to change the table name *EMPLOYEES* to *EMP*, then the rule "merging two neighboring attributes that share a suffix, and renaming it with their common suffix" to merge the *first-name* and *last-name* attributes, and the rule "replacing ,000 with K" to the data values of column *salary* of the table. The resulting table is shown in Figure 3.1.b. The paper [78] describes an extensive set of such rules, including those that perturb (a) the set of tables (e.g., joining two tables, splitting a table), (b) the structure of a table (e.g., merging two columns, removing a column, and swapping two columns), (c) the names (e.g., abbreviating names, adding prefixes), and (d) the data (e.g., changing formats, perturbing values). We note that these rules are created only once by eTuner, independently of any schema S.

Since eTuner generates schemas $V = \{V_1, \ldots, V_m\}$ from S, clearly it can trace the generation process to infer the correct matches $\Omega = \{\Omega_1, \ldots, \Omega_m\}$ between these schemas and S. Hence, the set V, together with the correct matches, form a *synthetic matching workload* $W = \{(V_i, \Omega_i)\}_{1..m}$ that is an approximation of the true future workload $\mathcal{T}$.

The synthetic workload idea can be adapted directly to our current context. Given a schema S, we first perturb S to generate a synthetic workload $W = \{(V_i, \Omega_i)\}_{1..m}$ (see [78] for the detailed algorithm). Next, we use M to match S with each schema V_i in W. Since we know Ω_i, the correct matches between S and V_i, we can compute $accuracy(S, V_i, M)$. We then return the average of $accuracy(S, V_i, M)$ over all schemas in W as our estimate of the true matchability score of S. We estimate the matching variance of S in a similar fashion.

Discussion: At a first glance, the idea of deriving a set of synthetic schemas from schema S might seem counterintuitive. One may question if it can effectively approximate the future schemas.

We believe that in the absence of *any* additional information, this provides a reasonable way to do it. (It is unclear what other alternatives we can consider.) While synthetic workloads differ from real future workloads, they do capture common variations in schema design. Moreover, although matchability scores estimated with synthetic workloads vs. real future workloads will differ, we only need the matchability rankings to be similar (and they often are, see Section 3.6), in order to revise S effectively. Of course, if developer P has additional knowledge about the future workload, then P can create additional transformation rules to capture those.

One may also question if the so-derived schema pairs would be easy to match. The answer is no, as it turns out that "reverse engineering" the process is quite difficult, given that the rules are *randomly* applied. We note that synthetic scenarios like these have recently also been used in competitions on ontology matching (oaei.ontologymatching.org).

Perhaps a useful perspective we can take on the use of synthetic schemas is that they provide a reasonable set of "test cases" to estimate how good our solutions are. In other words, if our solutions cannot even handle synthetic schemas, then how much confidence we would have that they can handle the real ones?

Finally, we note that the above matchability estimation process requires data instances for schema S. To maximize accuracy, schema matching systems increasingly make use of such data instances [74]. Hence, we want to analyze *both the schema and data* of S and propose changes to both. To do so, mSeer requires developer P to supply several sample data instances for S (as a part of the input). Section 3.6.6 shows that mSeer works well with only a few (3-5) instances, thus not imposing an excessive burden on developer P.

3.4 Analyzing Schema Matchability

We now describe the report generator, the second mSeer service. Given an internal mediated schema S, the generator produces a report that lists the matchability and variance of S and the main reasons for matching mistakes.

Schema: Product1, Matching System: iCOMA
Product1 has matchability 0.76 and variance 0.09 (synthetic workload: 20 schemas)

(1) Attribute "discount", data values = 0.00, 0.15, 0.20, ...
Correctly matched 11 out of 20 times, matchability 0.47

Reason R1: (3 times) "discount" has no match, but iCOMA predicts a match t
Example: t = "discontinued" of schema S2, data values = 0, 1, ...
Suggestion: revise the name or the data format of "discount" to move "discount" away from "discontinued"

Reason R3: (6 times) "discount" matches t, but iCOMA predicts a match t'
Example: t = "disc_price" of schema S3, data values = 0.00, 15.00, 20.00, ...
and t'= "discontinued" of schema S3, data values = 0, 1, ...
Suggestion: revise the name or the data format of "discount" to move "discount" closer to "disc_price" and away from "discontinued"

(2) Attribute "ship_via", data values = 1, 3, 7, ...

Figure 3.2: A sample matchability report

Figure 3.2 shows such a report. The report first describes schema S and the matching system M (e.g., Product1 and iCOMA in this case, see Section 3.6). Next, the report shows that S has a matchability 0.76 and variance 0.09 (over a synthetic workload of 20 schemas).

Next, the report tries to explain why S obtains a somewhat low matchability of 0.76. A reasonable way to explain this is to list the attributes of S, together with their matchability scores (so that developer P can get a sense about which attributes of S are difficult to match).

The *matchability score of an attribute* can be defined in a similar fashion to that of a schema (see Section 3.3.1). Then it can be estimated as follows. Let s be an attribute of S. Suppose that when matching S with schemas $V_1, \ldots, V_n$ of a synthetic workload W using a matching system M we obtain a set K of matches that involve s (i.e., matches of the form $s = t$, $t \in V_i$, $i \in [1, n]$). Then s's estimated matchability with respect to W and M is $m(s, W, M) = 2PR/(P + R)$, where P is the fraction of matches in K that are correct, and R is the fraction of correct matches involving s (and between S and the V_i's) found in K.

The report shows the most-difficult-to-match attributes first, to help the developer P quickly identify those. For example, the report in Figure 3.2 shows that attribute *discount* is the most difficult to match, with matchability 0.47.

Still, just showing that *discount* is difficult to match is not very informative for P. Hence, the report goes one step further, trying to explain the common matching mistakes involving *discount* and make suggestions on how to fix them. In Figure 3.2, the report lists two reasons R_1 and R_3 for *discount*. Reason R_1 for example states that iCOMA predicted spurious matches for *discount*, such as *discount* = *discontinued*. To fix this mistake, the report suggests to pick a more distinctive name for *discount*. Section 3.6 provides examples of mistakes identified and suggestions made by the report on real-world schemas.

In the rest of this section we will first identify a set of common matching mistakes. Then we describe how to generate a report such as the above one.

3.4.1 Common Matching Mistakes

In what follows, we use the term *appearance* to refer to the name and the data format of an attribute. We divide matching systems into *local* and *global* ones, and start our analysis with the local ones.

3.4.1.1 Mistakes with Local Matching Systems

A *local* matching system M matches two attributes s and t by analyzing their appearances to compute a similarity score $sim(s,t)$, then declaring $s = t$, if $sim(s,t) \geq \epsilon$ for a pre-specified ϵ. M is local in that it decides if s matches t based solely on $sim(s,t)$, not on any other matches (as global systems that we describe later do). Examples of such systems include many of those from the COMA++ matching library [13], the LSD basic system (without the constraint handler) [41], and Semint [64].

Now consider applying M to schemas S and V, where V is a synthetic schema, and consider attribute $s \in S$. Matching mistakes involving s fall into three cases:

Case 1. Predict a Spurious Match: $s = none$, i.e., it has no match, but M predicts $s = t$, where $t \in V$. This implies that $sim(s,t) \geq \epsilon$. The fundamental reason is that

R_1: the appearances of two non-matching attributes s and t are too similar.

To solve this problem, we should change the appearance of s to "move it away" from t. This can reduce $sim(s,t)$, thereby reducing the chance that M matches s with t. For example, if s has name "elec." (shorthand for "elective") with values "yes" and "no", and t has name "electricity" also with values "yes" and "no", then their appearances are too similar. To address this, we can expand s's name to "elective" and use values "1" and "0". As another example, if s has name "salary" with values "53000", "65500", etc., it can be easily confused with "zip code" (with values "53211", "60500", etc.), if in computing similarity scores M gives data value similarities a large weight. To address this, we can insert into the data values of s characters that never occur in zip codes (e.g., change "53000" into "53,000") to "pry" these two attributes apart.

Case 2. Miss a Match: $s = t$, but M predicts $s = none$. This implies $sim(s,t) < \epsilon$. The fundamental reason is that

R_2: the appearances of two matching attributes s and t are very different.

Examples include "yes/no" vs. "1/0", and "02.07.07" vs. "Feb 07, 2007". This is the reverse of Case 1. To solve this, we can change s's appearance to "bring it closer" to t. In many cases, however, this will not completely solve the problem. To see why, consider the following example.

Example 3.2 Suppose the synthetic workload W contains 100 attributes that match s: 60 attributes with data values "yes/no", and 40 with data values "1/0". Then no matter how we change s's data format, to "yes/no" or to "1/0", M will fail to match s in at least 40% of the cases.

Fundamentally, the problem is that in the future schemas, the attributes that match s *can appear in many different formats*. Hence if s appears in just a single format, it may fail to match many such attributes. To address this problem, we propose a multi-appearance representation, which we will discuss shortly.

Case 3. Predict a Wrong Match: $s = t$, but M predicts $s = t'$. The mistake in this case is two-fold. First, M fails to predict the correct match $s = t$, which implies $sim(s,t) < \epsilon$. Second, M predicts instead a wrong match $s = t'$, which implies $sim(s,t') \geq \epsilon$. Thus the reason is that

R_3: s is more similar to a non-matching attribute t', and less so to matching attribute t.

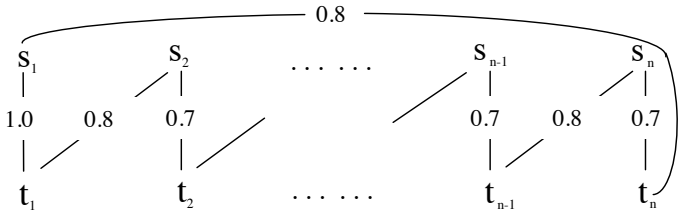


Figure 3.3: A matching scenario in a global system

To avoid this, we should change the appearance of s such that it moves "closer" to t, to increase $sim(s,t)$, and "away" from t', to reduce $sim(s,t')$. This case thus in a sense combines Case 1 and Case 2.

Changing the appearance of s is relatively easy when t and t' are quite different. The more similar t and t' are, the more difficult this task becomes. In the extreme case, when t and t' are "almost identical" in their appearances, such changing may be impossible. For example, let s be "stime" (shorthand for "start time"). Suppose t and t' are "time1" and "time2", respectively, and suppose that all three attributes s, t and t' have very similar values (e.g., "3:05am", "4:00pm", etc.). Then t and t' are so similar that it is virtually impossible to change s so that it would have a higher chance of matching correctly. Fundamentally, this is because the future schema T is ill-designed, by having two almost identical attributes. In this case, there is not much we can do on schema S.

3.4.1.2 Mistakes with Global Matching Systems

A *global* system M matches two attributes s and t by examining not just their appearances, but also *external* information, such as domain constraints [41] and special filters [69]. M exploits such information to revise similarity scores and match selections.

With a global system M, matching mistakes involving s still fall into Cases 1-3 described earlier. However, the underlying reason for a mistake may be quite different. Consider for example Case 2: $s = t$, but M predicts $s = none$. If M is local, then by the definition of local systems, we know that $sim(s,t) < \epsilon$ and that this is the fundamental reason why M misses match $s = t$.

However, if M is global, the reason for missing $s = t$ may be rather involved. It may even be the case that $sim(s,t) \geq \epsilon$ and yet M suppresses $s = t$, perhaps because t has been matched with

another attribute s' and hence can no longer be matched with s, due to some constraint. In general, matches in a global system can influence one another in a rather complex fashion, as the following example illustrates:

Example 3.3 Figure 3.3 shows a matching scenario with attributes $s_1, \ldots, s_n$ and $t_1, \ldots, t_n$ of S and V, respectively. Here an edge $s_i - 0.7 - t_j$ denotes that $sim(s_i, t_j) = 0.7$; there is no edge between s_i and t_j if $sim(s_i, t_j) = 0$.

Suppose that a global matching system M imposes the constraint that each attribute participates in at most one single match (e.g., [69]). Suppose further that M starts by selecting as a match the edge with the maximum score, and hence predicts $s_1 = t_1$. Since t_1 is already involved in this match, M has no choice for s_2 but to predict $s_2 = t_2$, and so on, until it predicts $s_n = t_n$. Now suppose that the correct matches are $s_n = t_{n-1}$, $s_{n-1} = t_{n-2}$, ..., $s_2 = t_1$, and $s_1 = t_n$. Then clearly the incorrect decision to match s_1 and t_1 has caused a chain of cascading matching errors, all the way to s_n and t_n.

Because of such cascading errors, pinpointing the exact reasons for matching mistakes of global systems can be very difficult. Consequently, in this work we focus on identifying *some common reasons* for mistakes, rather than conducting a comprehensive mistake analysis for global systems.

Specifically, when Case 2 or Case 3 happens (i.e., $s = t$, but M predicts $s = none$ or $s = t'$), and $sim(s, t) \geq \epsilon$, clearly Reasons $R_1 - R_3$ do not apply. In this scenario, we have observed that a very common reason is that

R_4: s is dominated by an attribute $s' \in S$.

By "dominating", we mean that $sim(s', t) \geq sim(s, t)$ (recall that t is the correct matching attribute for s). In this case, M often incorrectly matches s' with t. Then, due to a constraint such as "each attribute can participate in a single match", M can no longer match s with t. Consequently, it either declares $s = none$, leading to a Case 2 mistake, or $s = t'$, leading to a Case 3 mistake.

An extreme example of the domination scenario is when s and s' are "almost identical" (e.g., "time1" and "time2", with very similar data values "3:05am", "4:00pm", etc.). In this case, $s = t$ and $s' = t$ often have identical similarity scores, and M ends up guessing wrong 50% of the time.

Conditions		Likely Reasons	Suggestions
s = none, predicts s = t	*sim (s, t) ≥ ε*	R_1	(a) move s away from t
s = t, predicts s = none	*sim (s, t) < ε*	R_2	(a) bring s closer to t (consider multi-appearance representation (MAR) for s)
	sim (s, t) ≥ ε , *∃s′ s.t. sim (s′, t) ≥ sim (s, t)*	R_4	(a) move s closer to t (consider MAR) (b) move s′ away from t (c) check if ∃s′ such that s and s′ are highly similar
s = t, predicts s = t′	*sim (s, t) < ε ≤ sim (s, t′)*	R_3	(a) move s closer to t (consider MAR), and away from t′ (b) but, check if t and t′ are highly similar
	else if ∃ s′ s.t. *sim (s′, t) ≥ ε > sim (s, t)*	R_4	(a) move s closer to t (consider MAR) (b) move s′ away from t (c) check if ∃s′ such that s and s′ are highly similar

Table 3.1: Conditions, reasons, and suggestions used in report generation

To address the domination problem, we should change the appearances of s' and s so that s is "moved closer" to t and s' is "moved away" from t.

Summary: Table 3.1 briefly lists the conditions, likely reasons, and suggestions we have discussed so far, for both local and global systems. The first row of this table, for example, states that if $s = none$, but M predicts $s = t$, and $sim(s, t) \geq \epsilon$, then R_1 is a likely reason, and developer P should consider changing the appearance of s to "move it away" from t. The report generator uses this table to identify likely matching mistakes (see Section 3.4.2).

3.4.1.3 Multi-Appearance Representation

We have seen from the discussion in Case 2 that in the future schemas the attributes that match $s \in S$ can appear in many different formats. Hence if s appears in just a single format (as is the case today), it may fail to match many such attributes. To address this problem, we experimented with a *multi-appearance representation (MAR)* for such an attribute s, by creating different *relational views* over s, and enforcing the constraint that any attribute that matches one of these views must also match s.

To illustrate, consider again Example 3.2. Suppose s is "waterfront" with values "1/0". Then we can create a view v_1 over s, with name $waterfront_1$ and data values "yes/no", then treat v_1 as another attribute of schema S. Next we enforce the constraint that any attribute t that matches v_1 must also match s, and vice versa. This ensures that no matter whether t takes values "yes/no" or "1/0", we can match t with s.

Creating such views in relational schemas should incur a moderate effort from developer P, and the views do not have to be kept up-to-date by the minute, for matching purposes. It is important to note that instead of creating views, P can also simply record in a text document that "s can also take 'yes/no' values". However, no matching systems can exploit such *textual* information effectively today. Instead, virtually all of them have focused on exploiting the schema and data of attributes. Hence, we feel that capturing such information in views makes it more "understandable" to matching systems.

In theory, for an attribute s, we can create as many views as necessary, to capture all of s's possible future appearances. However, doing so can often make s "confusable" with other attributes, and hence can quickly decrease matching accuracy (e.g., by causing Case 1 or Case 3). Hence, developer P can propose such views for s, but P should let mSeer decide which views to keep. The experiment section shows that the use of such views as decided by mSeer can significantly improve matching accuracy.

3.4.2 Generating a Matchability Report

We are now in a position to describe the end-to-end working of report generation. Given a schema S, mSeer first generates a synthetic workload W. Next, mSeer applies the matching tool M to match S and schemas in W, then computes S's matchability and variance for the report.

Next, mSeer analyzes the above matching results to compute matchability scores for all attributes in S, and then displays these attributes in increasing order of their scores. For each attribute s, mSeer then generates an analysis as follows.

Let I be the set of all incorrect matches involving s (from workload W). mSeer finds the reason for each of these incorrect matches. Currently these reasons are $R_1 - R_4$ in Table 3.1 (or $OTHER$ if none applies). mSeer then groups matches in I based on their reasons, producing at most five groups. Next, mSeer reports each group as a triple (R, E, S): R is the reason, E is a concrete example to illustrate the reason, and S is a suggestion (to be described below). mSeer lists triples (R, E, S) in decreasing order of the corresponding group size (i.e., the number of matches in the group).

Within each group, mSeer selects as example E the incorrect match m that can be fixed most easily, since developer P seems likely to attempt to fix m first. Specifically, for group R_1, mSeer picks m with the lowest similarity score. For R_2, it picks m with the highest similarity score. For R_3, where $s = t$, but M predicts $s = t'$, it picks m that minimizes $[sim(s, t') - sim(s, t)]$. For R_4, where s is dominated by s', it picks m that minimizes $[sim(s', t) - sim(s, t)]$. mSeer then generates suggestion S by replacing variables in suggestion template with those in example E.

3.5 Improving Schema Matchability

Given a schema S, developer P can employ the report generator as described earlier to identify potential matching mistakes of S, then revise S to address these mistakes. Manually finding good revisions, however, is difficult and tedious. The revision advisor, the third mSeer service, addresses this problem. It automatically discovers a good set of revisions, then presents them to P, in form of a revised schema S^*. P is free to accept, reject, or modify further these suggested revisions.

We now describe the revision advisor. Clearly, the advisor can only suggest revisions that retain the *semantics* of S (e.g., it cannot suggest P to drop an attribute). Hence, we start by defining the notion of *semantically equivalent transformation rules* (or *SE rules* for short). Later we describe how the revision advisor finds a good set of SE rules to apply to S.

3.5.1 SE Transformation Rules

Let r be a transformation rule and $r(S)$ be the schema obtained by applying r to a schema S. Intuitively, we say that r is a *semantically equivalent (SE)* rule if for any schema S, S and $r(S)$ are semantically equivalent, i.e., creator P can use $r(S)$ instead of S in his or her application.

SE rules fall into two categories: *domain-independent* and *domain-dependent*. Examples of domain-independent rules are "replacing data values 'yes' with '1' and 'no' with '0"', and "abbreviating a table name to its first three letters". Other examples include rules that cover special data types, such as "if s is a date attribute, then reformat s's values as '06/03/07"', and "if s is a price, then insert '$' to front of data values". To use such rules, we must recognize the type of an attribute

(e.g., date, price, etc.). To do so, we employ a set of type recognizers, as described in [37]. Finally, an example of domain-dependent rules is "replacing attribute name 'gName' with 'gene-name"'.

We have created a large set E of domain-independent rules, to be used in the current mSeer implementation and for our experiments. These rules are created only *once, when building* mSeer, not once per schema S. We omit a detailed description of E for space reasons, but show a high-level description in Table 3.2.

Categories	Sub-categories	Rules	Descriptions
Structure	*Schema-level*	merge-two-tables	Merges two tables based on their join path to create a new table.
		split-table	Splits a table into two, and duplicates key attributes in both tables.
	Table-level	merge-attributes	Merges multiple attributes into one (e.g., merging Day, Month, Year into Date).
Name	*Syntactic*	prefix-table-name	Adds the table name to the attribute name as prefix.
		drop-prefix	Drops the first token of the name (e.g., ContactName → Name).
		append-data-type	Appends the data type of the attribute to its name (e.g., appending Phone to attribute Office).
	Dictionary-based	expand-abbreviation	Expands common abbreviations (e.g., Qty → Quantity).
		expand-acronym	Expands acronyms (e.g., SSN → SocialSecurityNumber).
		use-synonym	Uses synonyms (e.g., PostalCode → Zip).
Data	*Numeric*	convert-unit	Converts the unit of the numbers (e.g., Price = "14,500" → Price = "14.5 K").
	Categorical	change-category-values	Converts categorical value representation (e.g., Fireplace = "yes/no" → Fireplace = "1/0").
	Special-type	change-data-format	Changes data formats of special data types (e.g., Date = "12/4" → Date = "Dec. 4").

Table 3.2: Classification of SE transformation rules

It is important to emphasize that this set of rules is not meant to be comprehensive. New rules can easily be added to the set, including domain-dependent ones supplied by P. However, the current set of rules is adequate as a starting point for us to examine the proposed mSeer approach, and to demonstrate mSeer's feasibility, a major goal of this work.

3.5.2 Searching for Optimal SE Sequences

Let $E = \{r_1, \ldots, r_m\}$ be the set of SE rules fed into mSeer, as defined above. Abusing notation slightly, we will also use the term "rule r_i" to refer to a particular *application* of r_i to a schema S (i.e., r_i captures both the rule itself and an instance of applying it to an attribute of S), when there is no ambiguity.

Then given a schema S, we use $seq(S)$ to refer to the schema that results from sequentially applying rules $seq = (r_1, \ldots, r_n)$, where $r_i \in E$ for $i \in [1, n]$, to S. Note that SE rules are "transitive", in that $seq(S)$ is also semantically equivalent to S.

Intuitively, then, the goal of mSeer is to find a sequence seq^* that when applied to S yields a schema S^* with maximum matchability. Formally, $seq^* = \arg\max_{seq \in \mathcal{S}} m(seq(S))$, where $\mathcal{S}$ is the set of all sequences of SE rules in E and $m(seq(S))$ is the matchability of schema $seq(S)$. mSeer then faces two key challenges: how to estimate $m(seq(S))$ and how to find seq^* efficiently.

To address the first challenge, mSeer employs synthetic workloads, in the spirit of computing matchability that we have described so far. Recall from Section 3.3.2 that to estimate the matchability of S, mSeer employs a synthetic workload $W = \{(V_i, \Omega_i)\}_{1..m}$, where V_i is a synthetic schema obtained by perturbing S, and Ω_i is the set of correct matches between S and V_i. To estimate $m(seq(S))$, however, mSeer cannot simply employ W again, since the matching scenarios (S, V_i, Ω_i) do not involve S'. Instead, mSeer needs a new workload that approximates matching scenarios involving $seq(S)$. We show how to generate such a workload in Section 3.5.2.1.

To address the second challenge, mSeer employs look-ahead heuristics to cope with the exponential search space. The result is the algorithm RevSearcher, which approximates seq^*, and which we describe in detail in Section 3.5.2.2.

3.5.2.1 Estimating Matchability of Revised Schemas

After applying rules seq to schema S, we obtain a different but semantically equivalent schema $S' = seq(S)$. To determine whether applying seq is worthwhile, we need to estimate the matchability $m(S')$ of S'.

As discussed above, to compute $m(S')$, mSeer needs a workload W' that approximates matching scenarios involving S'. To achieve this, we augment mSeer as follows. When deriving the schemas in W, mSeer logs the applied SE rules R. These rules are then used to generate workload W' for S'. Specifically, mSeer generates W' by applying R to S', in the same way it generates W. After that, mSeer employs W' to compute $m(S')$. Applying the same rules R to generate W' ensures that W' is closest to W, compared with the workloads generated by randomly perturbing S'. This way, mSeer can compare the matchability scores of S' and S based on similar matching scenarios.

3.5.2.2 Algorithm RevSearcher

A simple algorithm H to approximate seq^* is to use the hill-climbing (i.e., greedy) heuristic to find the best rule to apply at each step. First, H generates a synthetic workload W from S, and uses W to compute the matchability $m(S)$ of S. Then H generates all schemas $S_1, \ldots, S_m$ that can be obtained from S by applying a single SE rule in E.

Next, for each schema S_i, H computes its matchability $m(S_i)$ as described in Section 3.5.2.1. Let S_k be the schema with the highest matchability, i.e., $m(S_k) = \max_{i=1}^{m} m(S_i)$. If $[m(S_k) - m(S)] < \theta$ (currently set to 0.005), then H terminates, returning the schema S^* with the highest matchability it has found so far, together with the rule sequence that creates S^* from S. Otherwise, H sets S to S_k, sets S^* to S_k, and transforms the workload W to W_k. It then repeats the search, starting with S_k.

In each search iteration, algorithm H finds and applies a *single* SE rule. Hence, it explores the search space rather "slowly", and at the same time is myopic. To address both problems, we develop algorithm RevSearcher. This algorithm works exactly like H, except that in each iteration it finds and applies *a set of SE rules*, instead of a single one (see the pseudocode in Figure 3.4). We now describe how RevSearcher finds this set of rules.

Compatible Rules: Let U be a set of SE rules. The result of applying U to S, denoted as $U(S)$, is meaningful only if the rules in U are *compatible*, in the sense that applying them *in any order* still results in the same schema $U(S)$. We say that two SE rules are *compatible* if they either apply to different attributes, or to different aspects of the same attribute (e.g., one applies to its name, and the other applies to its data values). Then we say that U is a *compatible set* if any two rules in U are compatible.

Finding the Best Set of Compatible Rules: In each search iteration, RevSearcher finds and applies a compatible set U^* of SE rules that maximizes matchability. Unlike H which enumerates all rules, RevSearcher cannot enumerate and evaluate all compatible sets, because there are often too many of them (if there are n SE rules, there may be up to 2^n such sets). Consequently, RevSearcher finds U^* greedily as follows.

Consider the first iteration, where RevSearcher starts with S. First, RevSearcher applies all SE rules to S and computes the matchability of all resulting schemas, just like H does, adding those rules that produce schemas with higher matchability than S to a set U. Next, RevSearcher computes the gain of each rule in U (defined below), adds the rule with maximum gain to U^* (which is initially empty), recomputes the gain of each remaining rule, then adds the rule that has maximum gain and that is *compatible* with all rules already in U^*, and so on. The iteration stops when U is empty or contains only rules that are either incompatible with some rules in U^* or of zero gain. This is the set of SE rules U^* that RevSearcher uses for the first iteration. Finding U^* for subsequent iterations is carried out in a similar fashion (see pseudocode in Figure 3.4).

Input: Schema S, set of SE rules $U = \{r_1, r_2, \cdots, r_n\}$
Output: maximal set of compatible rules U^*

1. Compute the matchability $m(S)$ of schema S;
2. For each r_i in U do
   2.1 Compute the matchability $m(r_i(S))$ of schema $r_i(S)$;
   2.2 If $m(r_i(S)) < m(S)$ then
       Remove r_i from U;
3. Compute the matchability $m(a_j, S)$ for each attribute a_j in S;
4. Let $m^*(a_j) = m(a_j, S)$, for each a_j in S;
5. $U^* = \phi$;
6. For each r_i in U do
   6.1 If r_i is compatible with all rules in U^* then
       Compute the matchability $m(a_j, r_i(S))$ for each a_j in $r_i(S)$;
       gain$(r_i) = \sum_j \max\{[m(a_j, r_i(S)) - m^*(a_j)], 0\}$;
7. $k = \arg\max_i(gain(r_i))$;
8. If *gain*$(r_k) > 0$ then
   8.1 Remove r_k from U, and add r_k to U^*;
   8.3 $m^*(a_j) = \max[m(a_j, r_k(S)), m^*(a_j)]$, for each a_j in S;
   8.4 Goto Step 6;
9 Return U^*;

Figure 3.4: The procedure that RevSearcher uses to find the best set of rules in each iteration.

Computing Gain of a Rule: All that remains is to describe computing the gain of a rule r, which measures the potential increase in matchability that applying r can bring. At first glance, it appears that this gain can be computed as $gain(r) = m(r(S)) - m(S)$, that is, the increase in matchability between S and the schema $r(S)$ obtained by applying r to S.

However, we found that applying this gain definition is not effective. For example, one might have two compatible rules, r_1 and r_2, that apply to the same attribute a of S (e.g., one to a's data

values and one to a's name). Suppose they both increase the matchability of S. Then with the above gain definition, RevSearcher will add both of them to U^*. However, it may be the case that when applied together, they cancel the effects of each other. Consider a matching scenario where attribute $a = none$, but the matching system predicts $a = b$ (reason R_1 in Table 3.1). Both r_1 and r_2 reduce the errors in matching a by moving a away from b. In the meantime, however, they undesirably move a closer to some attribute c. Although applying either rule in isolation does not incur the incorrect match $a = c$, applying them both might. This suggests that RevSearcher should select only one rule, which gives a higher matchability.

To alleviate this problem, we explore a different gain definition. Specifically, we define the gain of a rule r to be the total increase in matchability of the attributes $a_1, \ldots, a_n$ of S:

$$gain(r) = \sum_{i=1}^{n} \max\{[m(a_i, r(S)) - m^*(a_i)], 0\},$$

where $m(a_i, r(S))$ is the matchability of attribute a_i in schema $r(S)$ (if a_i does not exist in $r(S)$, then we set $m(a_i, r(S))$ to 0, indicating that r does not contribute to any gain on matchability of a_i). Furthermore, $m^*(a_i)$ is the maximal matchability that a_i has achieved so far. $m^*(a_i)$ is initially set to be $m(a_i, S)$, the matchability of attribute a_i in S. It is set to be $m(a_i, r(S))$ every time RevSearcher adds a rule r to U^* and $m(a_i, r(S))$ is higher than $m^*(a_i)$ at that point.

Note that $gain(r)$ is "conservative" in the sense that it "discourages" applying multiples rules to one attribute when subsequent changes to the attribute do not increase its matchability further. Also, it is "optimistic" in the sense that whenever $m(a_i, r(S))$ is lower than $m^*(a_i)$, this definition does not "punish" r; it simply sets the contribution of r to a_i to 0. Otherwise, it tends to underestimate the actual benefit of r, and RevSearcher ends up selecting fewer rules than it could.

3.6 Empirical Evaluation

We now describe experiments with mSeer. First, we ranked a set of schemas according to matchability with (a) synthesized schemas, and (b) real schemas. The rankings strongly agree with one another. We thus conclude that for estimating matchability, synthesized schemas provide a promising proxy for using difficult-to-obtain real schemas.

Domain	# schemas	# tables per schema	# attributes per schema
Course	5	3	13-16
Inventory	10	4	9-11
Real Estate	5	2	26-35
Product	5	2	46-50

Table 3.3: Real-world domains in our experiments

Second, we provide anecdotal evidence that matchability reports produced by mSeer can help a schema designer identify likely matching problems.

Third, once we had revised a schema using the revisions suggested by mSeer, we matched it against a set of *real schemas*, and showed that we could improve matching accuracy by 1.3-15.2% for 17 out of 20 schemas across four domains (while obtaining no improvement or minimally reducing the accuracy by 0.2-0.3% on the remaining 3). The results thus suggest that mSeer is robust and can revise schemas to improve their matchability across a range of domains.

Finally, we showed that (a) the multi-appearance representation could further improve matching accuracy by 7.1% on average, (b) mSeer is robust for small changes in the size of the synthetic workload, and (c) it requires only a few data instances to do well. We now describe the experiments in detail.

3.6.1 Experimental Setup

Domains: For research purposes, obtaining domains with a large number of realistic schemas is well-known to be difficult[1]. For our experiments, we obtained publicly available schemas [37, 41, 78] in four real-world domains, as shown in Table 3.3. "Course" contains university time schedules. "Inventory" describes business product inventories. "Real Estate" lists houses for sale, and "Product" stores product description of groceries.

Matching Systems: For experiments described in Sections 3.6.2-3.6.5, we employed a matching system called iCOMA, which consists of a name matcher, a decision-tree matcher, and a combiner.

[1]The largest domain that we are aware of is Thalia at www.cise.ufl.edu/research/dbintegrate/thalia. But its schemas are in XML, hence are not suitable for the current experiments.

The name matcher compares names based on edit distance. The decision-tree matcher compares attributes based on their values, and the combiner combines the similarity scores of the matchers by taking their average. The name matcher and the combiner are taken from COMA++, a state-of-the-art matching library [13], and the decision-tree matcher is added to iCOMA from LSD [41], so that iCOMA can exploit data instances. For sensitivity analysis in Section 3.6.6, we also evaluated mSeer using a revised version of iCOMA, taken from COMA++.

Experimental Methodologies: We briefly describe the methodology employed for the main experiments (Section 3.6.4). In those experiments, for each domain in Table 3.3, we first randomly selected a schema to be the internal mediated schema S, then computed its average matching accuracy m with respect to the remaining schemas in the domain (treated as future schemas). Next, we applied mSeer to revise S into S^*. Then we computed the average matching accuracy m^* of S^*, again with respect to the remaining schemas in the domain. Finally, we report the difference $m^* - m$ as an estimate of the matchability improvement of S (using mSeer) in real-world scenarios.

3.6.2 Utility of the Matchability Concept

We first examine what matchability scores can tell us. Since we estimate such scores using synthetic workloads, it is unlikely that they will be roughly the same as the true scores (that can be computed if we know the true set of target schemas). However, we hoped that they would help us *rank* schemas, given that such ranking lies at the heart of schema revision.

Consequently, we want to know that if we rank a set of schemas using (a) synthetic workloads, and (b) real schemas, how strongly would such rankings agree. Toward this end, in each domain, say Course, we first selected a schema S, then perturbed it using SE rules one rule at a time, to obtain a set of schemas $\mathcal{S} = \{S_1, \ldots, S_{10}\}$.

Next, we ranked the schemas in $\mathcal{S}$ in decreasing order of their matchability scores, computed using a synthetic workload. Since matchability scores vary depending on the particularities of a workload, we ranked a schema $S_i \in \mathcal{S}$ higher than a schema $S_j \in \mathcal{S}$ only if their scores differ by at least ϵ (currently set to 0.005). We call the resulting ranked list SynList.

We then created TarList, a similar ranked list of the schemas in $\mathcal{S}$, except now we computed their matchability scores using *all schemas in Course other than* S as a real-world target workload.

Finally, we computed the distance between SynList and TarList as the ratio between the number of disagreeing pairs (with respect to their rankings) and the total number of pairs. This is the *Kendall distance*, a popular IR measure of the distance between two rankings [38], adapted to our context.

We repeated the above process for all other schemas S in Course, then computed the average Kendall distance. These distances, for Course, Inventory, Real Estate, and Product, are 0.28, 0.22, 0.19, and 0.27, respectively. For comparison purposes, the average Kendall distance between TarList (the ranking produced using real-world schemas) and a randomly generated list, again for the above four domains, are 0.43, 0.44, 0.39, and 0.45, respectively, roughly twice the distances produced using the synthetic schemas. This suggests that matchability scores computed by mSeer are indeed useful in helping rank schemas with respect to their matchability. The schema revision results in Section 3.6.4 further quantify this degree of "usefulness", in showing that by using such rankings (produced with synthetic workloads), mSeer was able to revise schemas to improve their matchability across all four domains.

3.6.3 Usefulness of Matchability Reports

We now provide anecdotal evidence that matchability reports produced by mSeer can help the schema creator identify likely matching problems. Table 3.4 shows snippets of matchability reports produced by mSeer (condensed and compiled in English, for exposition and space reasons). The reports cover two schemas: Product1 comes from Product domain, and TPCH is the publicly available schema of the well-known TPC-H benchmark (see www.tpc.org/tpch), which we also experimented with to broaden our range of experience with mSeer. (We did not include TPCH in our other experiments because we could not obtain a set of schemas comparable to TPCH.)

Part 1 of Table 3.4 reports that iCOMA failed to match *discount* and *discounted*, and incorrectly matched *discontinued* and *discounted*. It is clear from examining this part that attributes *discount* and *discontinued* of schema Product1 are "too similar" (Case 3, see Section 3.4.1.1). In particular,

(1) iCOMA failed to match "discount" of schema Product1 and "discounted" of schema Product1_S1 iCOMA incorrectly matched "discontinued" of schema Product1 and "discounted" of schema Product1_S1 Suggestion: revise the name or the data format of "discontinued" to move "discontinued" away from "discounted"
(2) iCOMA failed to match "P_MFGR" of schema TPCH and "P_MANUFACTURER_GROUP" of schema TPCH_S1 iCOMA predicted no match for "P_MFGR" of schema TPCH Suggestion: revise the name or the data format of "P_MFGR" to move "P_MFGR" closer to "P_MANUFACTURER_GROUP"
(3) iCOMA incorrectly matched "C_COMMENT" of schema TPCH and "P_COMMENT" of schema TPCH_S1 iCOMA incorrectly matched "C_COMMENT" of schema TPCH and "P_NOTES" of schema TPCH_S2 Suggestion: revise the name or the data format of "C_COMMENT" to move "C_COMMENT" away from "P_COMMENT" and "P_NOTES"

Table 3.4: Compilation of report snippets generated by mSeer

their names share the string "disco", which can confuse a name matcher (e.g., one using q-grams [13]). Given this, developer P can change the name, e.g., from "discontinued" to "terminated", then rerun mSeer, to see if the problem has been addressed.

Similarly, Part 2 of Table 3.4 reports that *P_MFGR* failed to match *P_MANUFACTURER_GROUP*. Here, the abbreviation "MFGR" may have caused the names not to match. Note that the knowledge "MFGR" is an abbreviation of "MANUFACTURER_GROUP" is highly domain specific. Since we simply cannot know if a particular matching system will possess such domain specific knowledge, it is better to revise the TPC-H schema to make it match aware by expanding such abbreviations.

Part 3 of Table 3.4 reveals a different problem. This part first reports that *C_COMMENT* matched *P_COMMENT* incorrectly. Given that both names share "COMMENT", this is not surprising. But then mSeer reports that *C_COMMENT* also incorrectly matched *P_NOTES*, despite the fact that their data values are quite different (one attribute records customer comments, while the other records product comments). A likely explanation for this is that the matching system knows "COMMENTS" is a synonym of "NOTES", and thus makes the latter incorrect match. To address this problem, it is important that the strings "C" and "P" in the names must be fully expanded (e.g., to "CUSTOMER" and "PRODUCT") to "push" the attributes away from each other as much as possible.

Other likely matching problems for the TPCH schema (that we found from the mSeer report) includes abbreviations such as "MK", the use of very short names for ID attributes (making all of them "confusable" with one another), and the merging of words without some separation characters, such as "RETAILPRICE" (instead of "RETAIL_PRICE") and "ORDERSTATUS".

From working with several mSeer reports, we found that a promising future work direction would be to produce aids in designing easily matched mediated schemas. Some can be "best practice" rules for humans, e.g., "avoiding short prefixes that carry crucial information (such as *P_COMMENTS*)", "avoiding very short names for ID attributes", etc. A richer direction would be to provide a library of idioms, to be used in constructing attribute names.

3.6.4 Automatic Schema Revision

Next, we examine how well mSeer can revise a schema to improve its matchability. Figure 3.5 shows the results for all four domains, five schemas in each domain (Inventory has 10 schemas, from which we randomly selected five). Consider the very first schema, Homeseekers of Real Estate (at the topmost left corner of the figure). Here, the two bars show the average matching accuracy of the original schema and that of the schema produced by RevSearcher, respectively. This average accuracy is computed by matching against the target workload of Homeseekers, i.e., the set of all remaining schemas in Real Estate. Note that this target workload consists of *real-world schemas*; it approximates the true set of target schemas that Homeseekers will be matched against in the future. We generated the bars for other schemas similarly.

The results show that RevSearcher was effective in improving matching accuracy. It was able to revise schemas to achieve higher accuracy in 17 out of 20 cases, by 1.3-15.2%. It did not improve accuracy in one case (on Rice), and reduced accuracy minimally in two cases (on Product2 and Product5), by 0.2-0.3%. The results thus suggest that mSeer is robust and can revise schemas to improve their matchability across a range of domains. As an aside, the current unoptimized version of mSeer took 96-814 seconds to produce a revised schema in the experiments, spending most of time in searching for compatible rule sets (Section 3.5.2.2). To reduce runtime, we are exploring

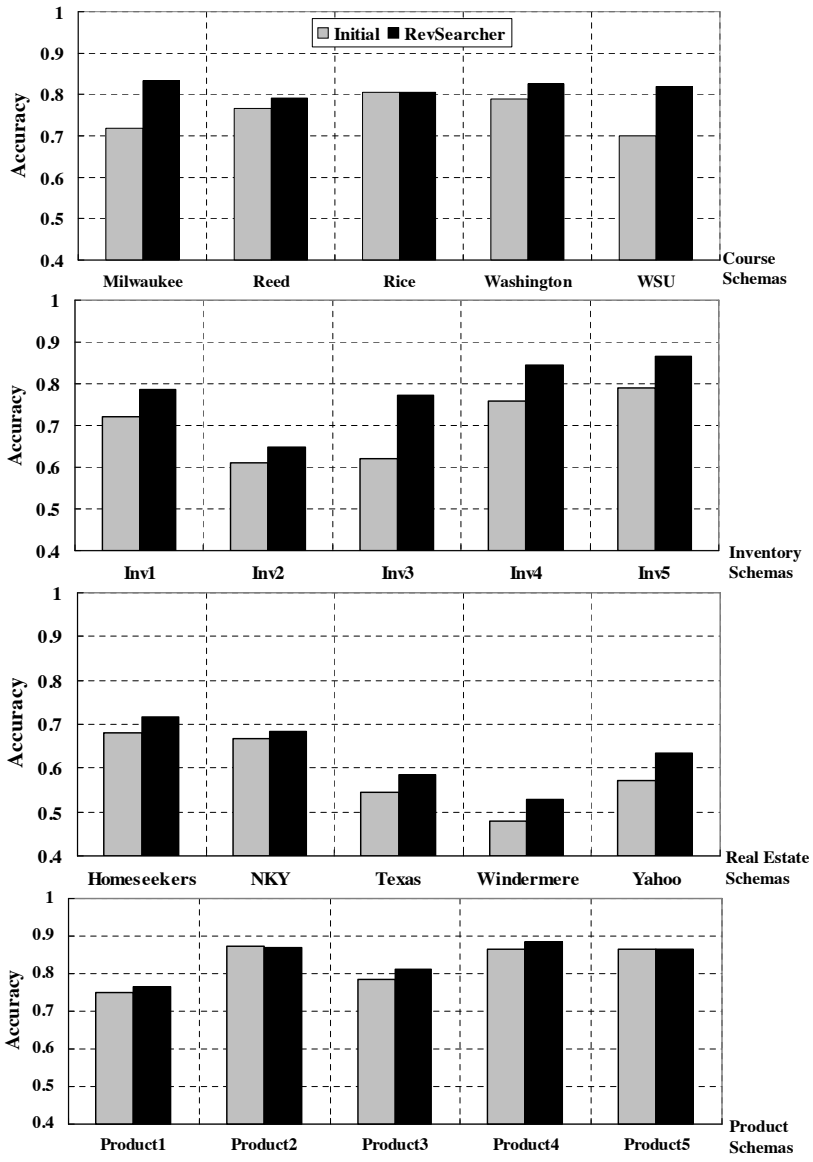


Figure 3.5: Matching accuracy of schemas produced by mSeer vs. that of the original schemas

better search techniques and ways to reuse results across matching steps to compute matchability scores incrementally.

Room for Improvement: How much better could mSeer revise a schema S if RevSearcher guided the search process using S's target workload, instead of a synthetic one? Figure 3.6 provides the answers for all four domains. In each domain, the three bars show the accuracies of the original

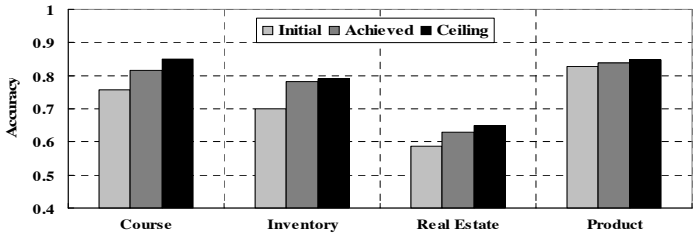


Figure 3.6: Improvement achieved vs. ceiling across all domains

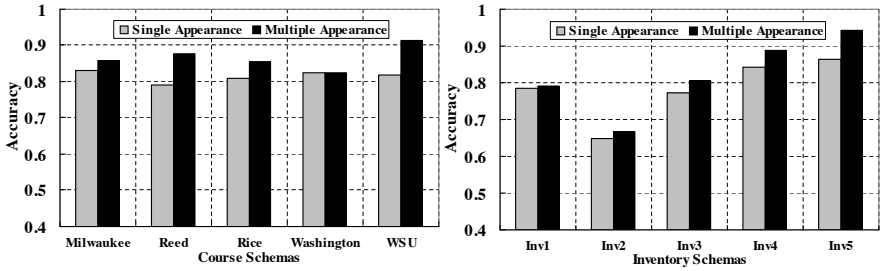


Figure 3.7: Accuracy with and without multi-appearance representation

schema, the schema produced by RevSearcher (using a synthetic workload), and the schema produced by a version of RevSearcher that uses the target workload, respectively. The accuracies are averaged over all sources in the domain.

The difference between the first and the third bar is the room for accuracy improvement. The results show that RevSearcher has done quite well. It achieves on average 69.7% of the improvement achievable with full knowledge (i.e., knowing the actual target workloads), demonstrating that its search strategy selects SE rules effectively. By expanding its set of SE rules, RevSearcher is likely to make inroads into the remaining 30%; and by pursuing an even better search strategy, RevSearcher can make further improvements, possibly beyond what is shown in the third bars.

3.6.5 Multi-Appearance Representation

Next, we examine the utility of multi-appearance representation (MAR) in schema revision (see Section 3.4.1.3). Figure 3.7 shows the results for the real-world schemas in Course and Inventory (experiments on other domains show similar results). For each schema, the two bars show the

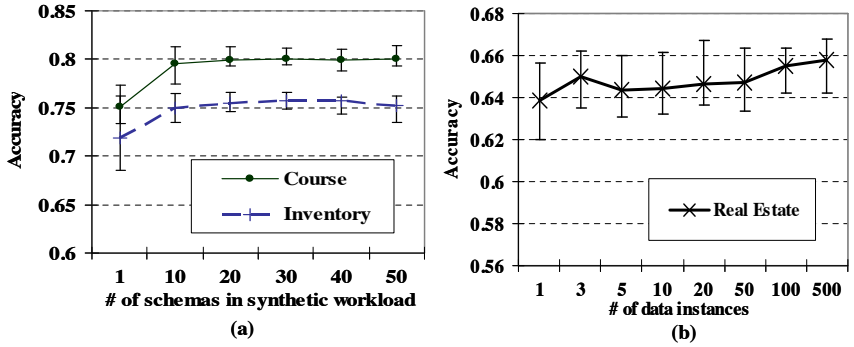


Figure 3.8: Change in matching accuracy with respect to (a) size of synthetic workload, and (b) number of data instances

accuracy of mSeer in single-appearance and multi-appearance settings, respectively, measured using the real schemas as the target workloads.

The results show that using MAR significantly improves the matchability of schemas, increasing accuracy in 9 out of 10 cases, on average by 5% in Course, and 3% in Inventory. MAR failed to improve accuracy in only one case (on Washington). This suggests that MAR is quite promising as a way to revise a schema with modest effort and yet making it more match aware.

3.6.6 Sensitivity Analysis

Size of Synthetic Workload: Figure 3.8.a shows the accuracy of the revised schema that mSeer produces, as we vary the number of schemas in the synthetic workload W. In the figure the lines show average accuracies and the vertical bars show the maximum-minimum accuracy ranges. The results show that as W's size increases from 1 to 20, W captures the results of more transformation rules, thus better representing true target workload. Consequently, matching accuracy increases and the maximum-minimum fluctuations decrease. After size 30-35, however, all transformation rules have been captured in W, and as the size increases further, W's "distance" to the real workload increases, and its performance starts to decrease. This result is consistent with the observations in [78], for tuning matching systems. Overall, the results suggest an optimal workload size in the

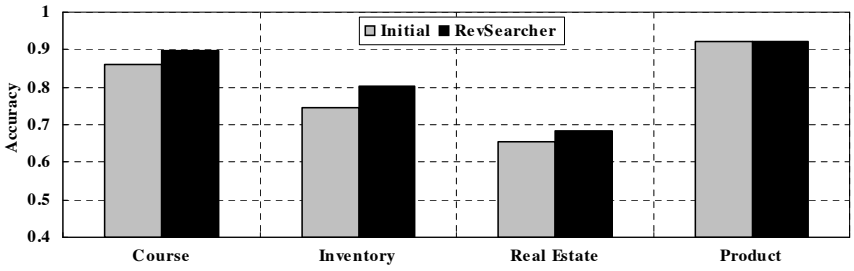


Figure 3.9: Matching accuracy with a new matching system

range of 20-30. The results also show no abrupt degradation of accuracy, thus demonstrating that mSeer is robust for small changes in the workload size.

Number of Data Instances: Figure 3.8.b plots the accuracy averaged over all sources in Real Estate, as we vary the number of data instances available to mSeer (i.e., to the decision-tree matcher). We chose Real Estate because it has the most of data instances available. The results show that more data instances led to a slow steady climb in accuracy. However, the accuracy is already quite high (within 2% of the maximum accuracy achieved) for 3-5 data instances. This suggests that mSeer requires only a few data instances to do well, and thus does not impose an unduly heavy burden on the schema creator.

New Matching System for mSeer: Next, we examine the performance of mSeer with respect to a different matching system. Instead of using system iCOMA described earlier (in Section 3.6), we employed a new system where the name matcher compares names using TF/IDF instead of edit distance, and the combiner takes the maximum of the similarity scores instead of the average (see COMA++ [13]).

Figure 3.9 summarizes the results with this new matching system, over all four domains. The results show that RevSearcher was able to revise schemas to improve accuracy in all four domains. RevSearcher for instance increased the average accuracy in Inventory by 5.7%. The results thus suggest that mSeer can be effective with more than one matching system.

3.7 Summaries

We have described the problem of analyzing and revising mediated schemas to improve their matchability, and presented a promising initial solution. Our work can help to motivate further research in this novel direction to schema matching. A sample of important issues follow. In this work, we have proposed a reasonable way to generate synthetic schemas. But what is the optimal way? What should be an optimal set of rules? Our analysis of matching mistakes and related experiments have achieved the goal of demonstrating that such a report of matching mistakes can be very useful to the schema creator. But can we improve the analysis further? In particular can we analyze better matching mistakes of global matching systems? Likewise, can we develop a better set of SE rules and a better search technique? Many more interesting challenges remain, such as developing an interactive environment (in which a creator can accept or revise a suggested schema revision on the fly, and can in general interact with the system in real time to revise the schema) and generalizing the work here to other data representations (e.g., XML) or problem contexts (e.g., revising schemas to facilitate record matching).

From a broader perspective, perhaps the most important conclusion drawn from this work, as well as the eTuner one, is that synthetic schemas can be very helpful for schema matching, and thus deserves more studies into their generation and usage.

Chapter 4

Opening Up Structured Web Portals for User Feedback

Structured Web portals are typically built by applying information extraction and integration (IE/II) techniques to unstructured text (e.g., Web pages of news articles), semi-structured data (e.g., Wikipedia infoboxes), as well as structured data (e.g., product feeds from vendors). Automatic IE/II are inherently difficult and are thus error-prone. As we discussed in Chapter 1, soliciting and incorporating user feedback is a promising solution to improve the quality of the IE/II results. In this chapter, we outline our ideas on "opening up" a structured Web portal for user feedback, describe its challenges and opportunities, and provide our solutions [34]. In the end of the chapter, we describe a real-world implementation of the solution and preliminary experiments that demonstrate the utility of our approach.

4.1 Introduction

Creating structured Web portal using automatic IE/II approaches has many benefits: it incurs relatively little human effort, often generates a reasonable initial portal, keeps the portal fresh with automatic updates, and enables structured queries over the portal. However, it usually suffers from inaccuracy, caused by imperfect extraction and integration methods, and limited coverage, because it can only infer whichever information is available in the data sources. While it is difficult or even impossible to find automatic solutions to solve these problems, human users (developers and ordinary portal users) often can easily spot and correct the errors.

In this chapter, we consider how to combine automatic IE/II approaches with user feedback to build structured Web portals. Specifically, we use automatic IE/II approaches to deploy an initial

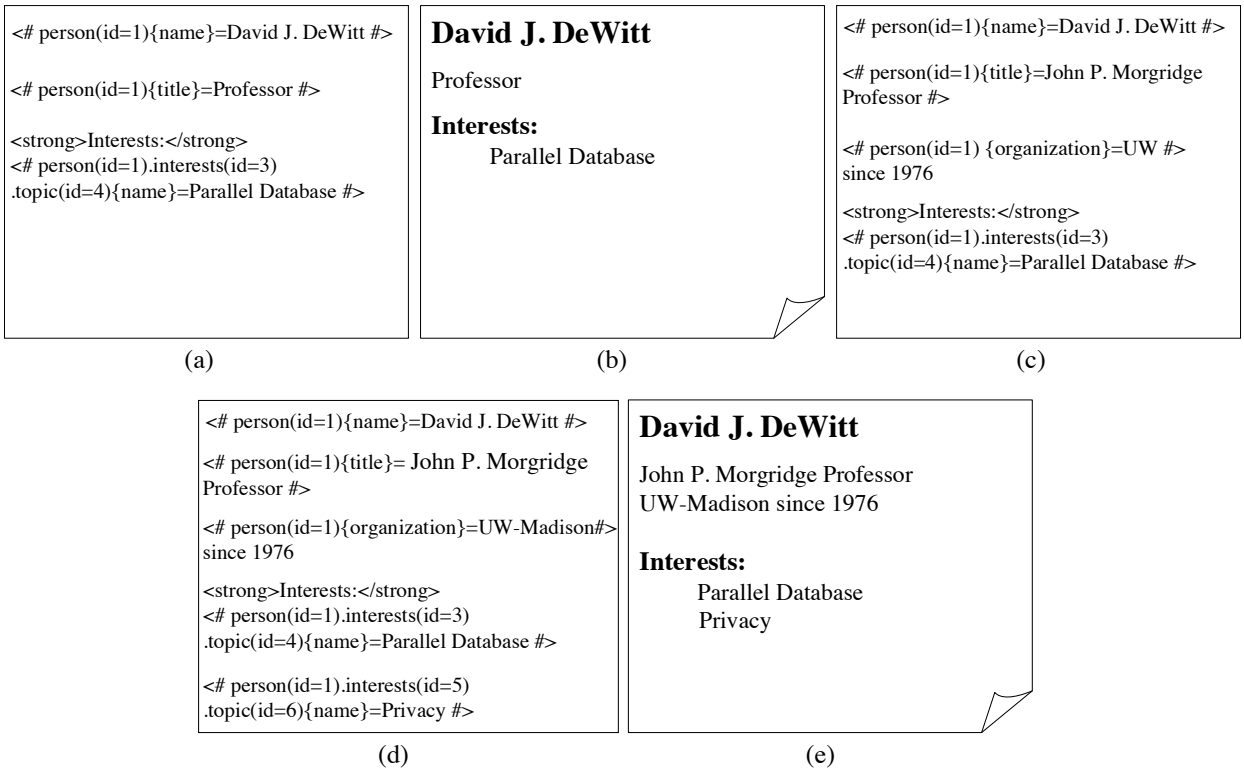


Figure 4.1: An example to illustrate the Swiki approach.

portal, then allow *both* machines and human users to revise and add material. Machines can add structured information to keep the portal updated automatically, while users can provide feedback on the material or improve them (e.g., by adding and editing both text and structured information). Furthermore, machines and humans can also correct and augment each other's contributions, in a synergistic fashion.

Exposing data from the underlying database directly, e.g., in the form of relational tables as the data is stored in an RDBMS, to users for their feedback is not only hard to use but also dangerous. Users may very likely have difficulty understanding how the data pieces in a table correspond to what they have seen on the portal. Besides, in the case of error or malicious users, changes they made may be catastrophic. As a common practice, developers build user interfaces via which they can present the data in a way easy for users to understand and edit and via which they can restrict which data users can edit. As wiki interface becomes popular and widely adopted among a growing

number of Web communities, in this chapter we study how to allow user to provide feedback via a wiki interface and how to enable the portal to interpret the feedback and incorporate it into the portal. Although the solution we lay out is specific to wiki interface, we believe the solution can be easily adapted and employed to other user interfaces. We refer our solution as Swiki (shorthand for "structured wiki"). The following example illustrates the approach.

Example 4.1 Suppose we apply Swiki to build a portal for the database community. We can start by applying a semi-automatic approach (i.e., "machines") to extract structured data from the Web, then use the data to create and deploy wiki pages, such as page W in Figure 4.1.a. Page W contains "structured data pieces" mixed with ordinary wiki text, and will display as the HTML page in Figure 4.1.b. In effect, W describes a person entity who has three attributes: id = 1, name = "David J. DeWitt", and title = "Professor". This person also participates in a relationship called "interests" with an entity of type "topic", whose name is "Parallel Database".

Once W has been deployed, a user U may come in and edit page W, e.g., by correcting the value of attribute *title* from "Professor", which was generated by machines, to "John P. Morgridge Professor". U may also contribute a structured data piece "$<\#$ person(id=1)$\{$organization$\}$= UW $\#>$", to state that this person is working for an organization called "UW". Finally, U adds free text "since 1976" after this data piece. The edited page W' is shown in Figure 4.1.c.

Later a machine M may discover from data sources that the above person also participates in "interests" relationship with topic "Privacy". M can then add this piece of information to the page, as "$<\#$ person(id=1).interests (id=5).topic(id=6)$\{$name$\}$=Privacy $\#>$". With high confidence, M may also correct the value of attribute *organization* from "UW", which was contributed by U, to "UW-Madison". The resulting wiki page W'' is in Figure 4.1.d, and it will display as the HTML page in Figure 4.1.e. Thus, page W has been evolved over time, with both machines and users' contributing and correcting each other's contributions.

The approach can bring significant benefits. First, it can achieve broader and deeper coverage, because it exploits both machines and human users. Second, it can provide more incentives for users to contribute, because the initial portal built by machines can already be reasonably useful

and comprehensive, thus motivating users to further improving it. Third, it can keep the portal more up to date, with less user effort, because machines can continuously monitor data sources and update certain parts of the portal. Finally, the structured data in the wiki pages of the portal is also stored in an underlying structured database, thus enabling a variety of structured queries over the portal.

In the rest of this chapter we elaborate on the above approach. First, we consider how to build an initial wiki-based portal, using machines. We cast this as a *view creation* problem: store the data generated by machines in a structured database G, create structured views over G, then export the views in wiki pages. The key questions are then: How to model and implement the structured database G? What should be the view language? And how to export the structured data of the views into wiki pages? As parts of our solution, we represent the machine-generated data using an entity-relationship (ER) model, define a path-based view language over this model, extend the standard wiki language[1] with *s-slots* – constructs to embed structured data into the natural text of wiki pages, then show how to export the views in wiki pages, using s-slots (Section 4.3.3).

Next, we consider how to manage user contributions to the portal. If a user U has edited a wiki page W, then we want to extract the "structured" part of U's edits, and "push" it all the way into the underlying database G. The key questions here are: What is it that U is conceptually allowed to edit? And how to efficiently infer such edits based on what U has done to a wiki page W? To answer these questions, we cast the problem of processing user contributions as a problem of mapping U's edits over the wiki page into edits over the corresponding view, then from this view into edits over G. This is a *view update* problem. But it is complicated (compared to RDBMS view update) by the facts that here (a) U can also edit the *schema*, not just the data, of the view, and (b) U's edits, being limited to the wiki interface, are often ambiguous. Furthermore, after we have updated database G with edits from W, we must decide how to propagate this update to other views and corresponding wiki pages. In Section 4.4 we elaborate on these issues, then provide a solution.

[1] http://en.wikipedia.org/

Finally, for the sake of completeness, in Section 4.5 we briefly touch upon the problem of managing *multiple* users, where we extend current solutions employed in Wikipedia (namely, optimistic concurrency control and access rights based on a user hierarchy) to handle concurrent editing and malicious users. We also consider how to let machines join users in updating the portal. The key challenge is the following: once a user has entered an edit, can machines be allowed to overwrite the edit, and when?

We have been applying the above solution to build a user-feedback-enabled portal for the database research community (see the live system[2]). In Section 4.6 we report our experience and preliminary experiments that demonstrate the potentials of this approach, and suggest opportunities for future research.

To summarize, in this chapter we make the following contributions:

- Introduce a new hybrid approach that employs both machines and human users to build structured Web portals, backed up by an underlying database. As far as we know, ours is the first work that studies this direction in depth.

- Provide solutions to modeling the underlying structure database, representing views over this database with a path-based language, and exporting these views in wiki pages.

- Provide an efficient solution to process user edits in wiki pages and "push" these edits into the underlying database. The solution recasts this problem as translating edits across different user interfaces.

- Provide empirical results over a real-world implementation that demonstrate the promise of the approach and suggest opportunities for future research.

4.2 The Swiki Approach

In the rest of the chapter, we describe the Swiki approach. Figure 4.2 illustrates how Swiki works. It starts by applying M, a machine-based solution, to extract and integrate data from a

[2]http://dblife-labs.cs.wisc.edu/wiki-test/index.php/Main_Page

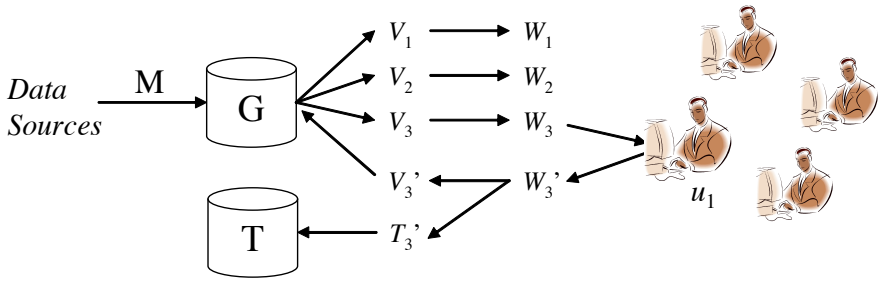


Figure 4.2: The Swiki architecture

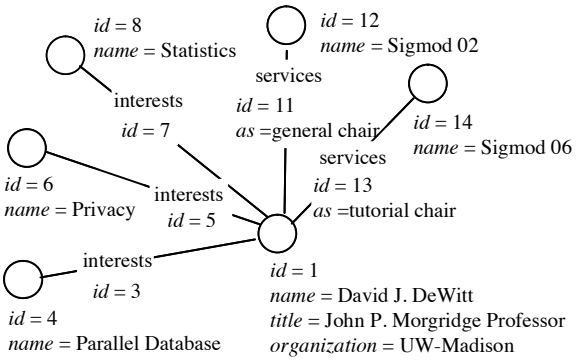
set of data sources, then loads this data into a structured database G. Next, it initializes an empty text database T, which will be used in the future to store text generated by users. Then Swiki generates structured views over G (e.g., $V_1 - V_3$ in Figure 4.2), and exports them in wiki pages (e.g., $W_1 - W_3$). The initial portal $\mathcal{W}$ then consists of all such wiki pages.

Portal users and machine M then revise and add material to $\mathcal{W}$. Suppose a user u_1 has revised wiki page W_3 into page W_3' (Figure 4.2). Then Swiki extracts the structured data portion V_3' from W_3' and uses it to update the structured database G. Next, Swiki extracts the text portion T_3' from W_3' and stores it in the text database T. Swiki also reruns machine M at regular intervals (to obtain the latest information from the data sources), updates G based on the output of M, then updates the views and wiki pages accordingly. Updating a wiki page W_i for example means creating a new version of W_i that combines the latest versions of its structured data portion from G and text portion from T. In addition to revising existing wiki pages, as described above, both users and machines can add new pages or delete existing ones.

The next two sections describe the key contributions of this work: how to build the initial portal and to manage user contributions. Section 4.5 briefly touches upon the issue of managing multiple users and machine.

4.3 Creating the Initial Portal

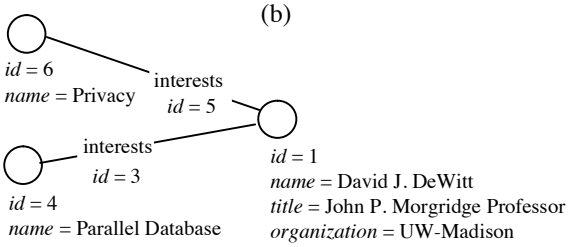
To create the initial portal, we proceed in three steps: employ a machine M to create a structured database G, create structured views V_i over G, then convert each view V_i into a wiki page W_i.



(a)

```
person(id=1){name,title}
person(id=1).interests.topic{name}
person(id=1).interests.topic(name=Statistics){name}
```

(b)



(c)

```
<# person(id=1){name}=David J. DeWitt #>
<# person(id=1){title}=John P. Morgridge
                                    Professor #>

<# person(id=1) {organization}=UW-Madison #>

<strong>Interests:</strong>
<# person(id=1).interests(id=3)
        .topic(id=4){name}=Parallel Database #>

<# person(id=1).interests(id=5)
                .topic(id=6){name}=Privacy #>
```

(d)

Figure 4.3: (a) A snapshot of the ER graph G, (b) a sample view schema, (c) a sample data of the above view, and (d) how the above sample data is exported into a wiki page in the s-slot wiki language.

4.3.1 Creating a Structured Database G

Here we describe in detail the language we use to model database G, how we extend a conventional RDBMS to capture temporal aspect of G, and how we initialize G using the Cimple solution [35].

4.3.1.1 Modeling Database G

To model G, we can choose from a wide variety of data languages. Since the data from G will eventually appear in wiki pages as structured constructs (see Section 4.3.3 for a motivation

for this), we had to select a data language that *ordinary, database-illiterate* users are familiar with, and can quickly understand and edit. Since most users are already familiar with the concepts of entity and relationship, as commonly employed by current portals, we choose an ER language to represent the data in G.

Specifically, we define the schema G_s of G to consist of a set of entity types $E_1, \ldots, E_n$ and a set of relation types $R_1, \ldots, R_m$. Each entity/relation type is specified using a set of attributes. Attributes are either atomic, taking string or numeric values, or set-valued.

Next, we define the data G_d of G to be a temporal ER data graph. This graph contains (a) a set of nodes that specify entity instances (or entities for short when there is no ambiguity), (b) a set of edges that specify relation instances (or relations for short when there is no ambiguity), (c) temporal information regarding attributes, entities, and relations, e.g., when an attribute/entity/relation was created, by which user, when it was deleted, by whom, when it was reinstated, etc. This information will be used in managing users (Section 4.5). We view machine M as a special user M.

We require G to be a temporal database that captures all changes so far, so that later we can develop undo facilities. Note also that even if G_s specifies that a person entity has an attribute email, this attribute can be missing from a particular person instance.

Figure 4.3.a shows for example the snapshot of a tiny G_d at time 1. On this snapshot the nodes are entities and the edges are relations (labeled with relation names). The attributes are described next to the nodes and edges.

4.3.1.2 Storing G using RDBMS

We want to query G efficiently and may want to implement a variety of concurrency control schemes later (to manage concurrent user edits), including lock-based schemes. Consequently, we decided to store G_s and G_d using an RDBMS. The key questions are then: (1) How to convert G_s, essentially an ER graph, into a set of relational tables? (2) How to extend a conventional RDBMS to store temporal data? (3) How to manage data from multiple users and machine? In what follows,

we first elaborate on and propose an initial solution to each question. Then we present a complete solution to storing G using an RDBMS.

Converting G_s into Relational Tables: As described above, schema G_s consists of a set of entity types and relation types. A standard approach to translating G_s into a relational database schema is to convert each entity (or relation) type into a table. And each attribute of the entity (or relation) type becomes an attribute of the table. An example is shown in Figure 4.4.a. Table *person* store person entity instances. In the table, column *id* gives the ID of a person entity, and *name*, *title* and *organization* are the three attributes describing each person. Such a design, however, is not space-optimized in our scenario for the following reasons:

- A table may be sparse. Take a person table for example. Most title values may be missing. This happens when title values are obtained from data sources, but the extractors are not powerful enough to extract them, or many title values are simply not available in the sources.

- Users may create new attributes for an entity type (e.g., creating attributes *homepage* and *country* for *person*). In this case, we need to enlarge the schema of the entity table, and entries for these new attributes are empty.

- Last but not the least, as we will see later, space utilization gets worse when we extend an RDBMS to store temporal data. When an attribute is updated, instead of updating the value in place, we *logically* delete the record with the old value and insert a new record with the new value. Other attribute values in the old record are copied to the new record. Consequently, we waste space in duplicating other attributes. Waste is significant when the table is wide (i.e., contains many attributes) and updates are frequent.

To address these problems, we chose to vertically partition an entity or relation table along each attribute. Consider an entity type E. Let $A_1, \ldots, A_n$ be the set of attributes E has. We convert E into n attribute tables. Each attribute table T_i ($1 \leq i \leq n$) is defined as $T_i(\underline{id}, value)$, where id stores the ID of an entity e, and $value$ stores the A_i value of e. In table T_i, we only store those entities that have an A_i value. A partitioning of the person table in Figure 4.4.a is given in

person

id	name	title	organization
1	David J. DeWitt	Professor	UW-Madison
2	Mike Brown	NULL	NULL
3	Chris Clifton	NULL	Purdue

(a)

person_name

id	value
1	David J. DeWitt
2	Mike Brown
3	Chris Clifton

(b)

person_title

id	value
1	Professor

(c)

person_organization

id	value
1	UW-Madison
3	Purdue

(d)

Figure 4.4: Tables for *person* entities: (a) a single table for all attributes, and (b)-(d) vertical partitions of the single table.

Figure 4.4.b-d. Note that table *person_title* has only one record since title values for the other two persons are missing. Similarly, we can convert a relation type into a set of attribute tables. For a relation type, in addition to the attribute tables, we need one more table to store the IDs of the entities that each relation relates, as we will see later.

Supporting Temporal Data: A user may enter an incorrect data value into the database, either unintentionally or intentionally. Once detected, we need to be able to rollback the data item to its previous correct value, which may be a long time ago. To provide such undo facilities, we require G to be a temporal database. Extending a conventional database to support temporal data has been well-studied [81, 51]. In this work we use the transaction-time table solution which is described in detail in [81].

Specifically, to convert a non-temporal attribute table $T(\underline{id}, value)$ into a temporal table T', we append T with two columns, denoted as *start* and *stop*. Thus we obtain $T'(\underline{id}, value, start, \underline{stop})$. Attributes *start* and *stop* are two timestamps: *start* indicates when a value was first inserted into the database, and *stop* indicates when the value was updated or deleted. Note that the primary key of T' consists of *id* and *stop*. This is because an entity (or relation) attribute may take different values at different times. In our design, all these values are stored in the same table with the same entity (or relation) ID but different *stop* values.

Figure 4.5.a gives an example of a temporal table for *person_organization* (Figure 4.4.d). In the example, at time 2007-04-01 08:01:20, a user entered organization "UW-Madison" for the person entity with $id = 1$ (person1 for short). Attribute *start* was set to "2007-04-01 08:01:20" to indicate

id	value	start	stop
1	UW-Madison	2007-04-01 08:01:20	9999-12-31 23:59:59
3	Purdue	2007-05-02 11:40:35	9999-12-31 23:59:59

(a)

id	value	start	stop
1	UW-Madison	2007-04-01 08:01:20	2007-05-27 09:50:10
3	Purdue	2007-05-02 11:40:35	9999-12-31 23:59:59
1	UW	2007-05-27 09:50:10	9999-12-31 23:59:59

(b)

Figure 4.5: Examples of transaction-time tables for *person_organization*: (a) before entity with id=1 is updated, and (b) after entity with id=1 is updated.

that the value "UW-Madison" started to be current at the time of insertion. Attribute *stop* was set to "9999-12-31 23:59:59", which is the largest timestamp, to indicate that the value would be current forever.

Moreover, when an attribute value is updated or deleted, we first *logically* delete its record, then insert a new record with the new value (for update) or a NULL value (for deletion). Consider again the table in Figure 4.5.a. Suppose that at time 2007-05-27 09:50:10, another user modified the organization value of person1 from "UW-Madison" to "UW". To reflect the modification in the table, first we located the record with the value "UW-Madison", and changed the *stop* time of the record to the current time, denoting that the value of "UW-Madison" stopped to exist at 2007-05-27 09:50:10. Next, we inserted a new record for value "UW". We set *start* and *stop* of the new record to "2007-05-27 09:50:10" and "9999-12-31 23:59:59", respectively, denoting that "UW" would be the current value from 2007-05-27 09:50:10 on. The temporal table after the modification is shown in Figure 4.5.b.

By adding *start* and *stop* to an attribute table and by doing logical deletions and updates, we keep track of all values that an attribute has taken, and for each value, the time period during which it was current. This way, we are able to recover an attribute value of any time in the past. Besides tracking attribute values, we also need to maintain temporal information regarding entities and relations themselves, e.g., when an entity was created, and when a relation was deleted. To store such temporal information, for each entity and relation type, we first create a special attribute

exists, then create a temporal table for *exists* the same as we do for other attributes. Attribute *exists* can take one of the two values, 1, denoting that the corresponding instance was created or reinstated, or 0, denoting that the instance was deleted. Creating an entity or relation instance can thus be implemented as inserting a record into an *exists* table with a value of 1, and deleting an instance can thus be implemented as logically updating the value to 0. This way, we are able to tell from an *exists* table whether an instance existed at a given time.

Managing Data from Multiple Users: Multiple users may contribute data into the database. For user management (Section 4.5), we need to know which user inserted, updated or deleted a data item. Moreover, two users may disagree on the value for one data item. And we need to decide whose value to use in generating V_d for a wiki page (Section 4.3.2).

To track the source of each data item, we further extend a temporal table T' by appending a column *who*, which stores the ID of the user who entered that item. The resulting table T'' is defined as $T''(\underline{id}, value, start, \underline{stop}, who)$. Note that the primary key does not change, since we only allow one value of each attribute to be current at any time, regardless of by whom.

Among all users, machine M is a special one. It automatically extracts and integrates data from a set of data sources. Thus it supplies data into the database much more frequently than any particular human user. On the other hand, M's data suffers from inaccuracies due to the capacity of the extraction and integration methods M uses. Consequently, M's data has lower credibility than other users' data. Therefore, we need to distinguish M from the rest of users. As a solution, for each attribute A, we create two temporal tables, $A\_m(\underline{id}, value, start, \underline{stop}, who)$[3] and $A\_u(\underline{id}, value, start, \underline{stop}, who)$. Table A_m stores attribute values entered by M, and table A_u stores values entered by human users.

An attribute may have different values in tables A_m and A_u. To decide which value to use in generating V_d, we need to resolve conflicts between the two tables. As a solution, we define a view table A_p over A_m and A_u. Table A_p has the same schema as A_m and A_u, and it stores the reconciled value of each attribute. Table A_p is updated when A_m or A_u is updated. Thus

[3] In an A_m table, $who \equiv$"M" since M is the only machine involved. We keeps attribute who in the table so that our design is easily extensible to multiple machines.

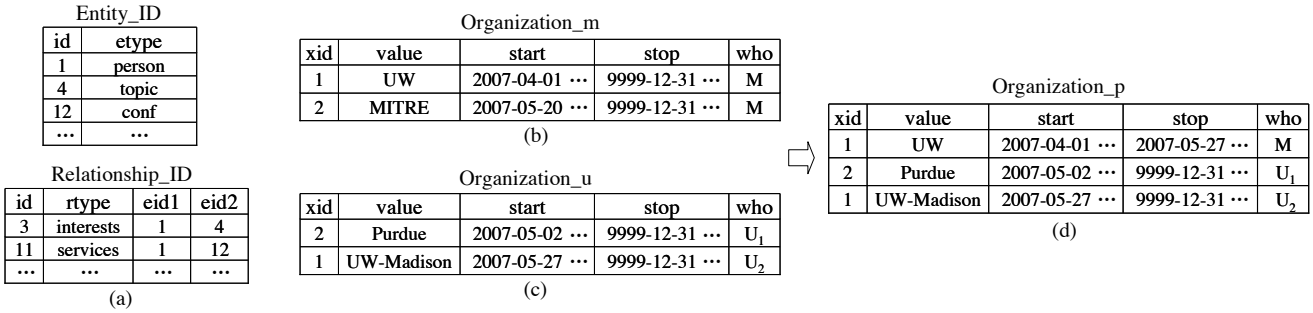


Figure 4.6: An example of attribute tables for *organization* of entity type *person*: (a) A_m, (b) A_u, and (c) A_p.

we can embed in its update procedure how we resolve conflicts in attribute values. Specifically, when an attribute a is updated in either of A_m and A_u, we first check whether a is already in A_p. If not, we simply insert a with its value into A_p. (Values for *start*, *stop* and *who* are assigned accordingly.) Otherwise, we need to decide whether we should overwrite a's value in A_p. A reasonable approach is to allow a user U to overwrite data entered by M or another user. We also allow M to overwrite its own data, but only allow it to overwrite U's data in certain situations, for example, when M is sufficiently confident in its data.

An example of A_m, A_u and A_p for table *person_organization* is shown in Figures 4.6. For simplicity of illustration, we assume that a user U can overwrite machine M's data but M cannot overwrite U's data. Based on this assumption, when user U_2 entered value "UW-Madison" for person1, we first inserted the value into table *person_organization_u*, then logically updated the existing value "UW" in table *person_organization_p*. Value "UW" was entered by M and thus we overwrote it with U_2's value. In contrast, when M entered "MITRE" for person3 into *person_organization_m*, we did not update value "Purdue" in *person_organization_p* since "Purdue" was entered by a human user.

Finally, table *A_p* can be explicitly stored in the database or computed as needed. In our design, we chose to materialize A_p for efficiency.

A Complete Solution: With all the problems addressed, we now present a complete solution to storing G in an RDBMS.

Formally, let G_s and G_d be the schema and the data of G. Let $E_1, \ldots, E_n$ be the set of entity types in G_s, and $R_1, \ldots, R_m$ be the set of relation types in G_s. Suppose for simplicity that each relation type is binary. We create the following relational tables to store G:

- An entity ID table $Entity\_ID(\underline{id}, ename)$, where id and $ename$ store the ID and the type of an entity.

- For each entity type E, we create a special attribute *exists*, whose value can be either 1 or 0. Denote *exists* as A_0. Let $A_1, \ldots, A_k$ be the attributes of E in G_s. For each attribute $A \in \{A_0, \ldots, A_k\}$, we create three temporal tables, A_m, A_u and A_p. Each table $T \in \{A\_m, A\_u, A\_p\}$ is defined as follows:

  $$T(\underline{id}, value, start, \underline{stop}, who),$$

  where *id* is the ID of an entity, and *value* is the value of attribute A of that entity. Timestamps *start* and *stop* specifies a time period during which the value was current. And finally, *who* gives the ID of the user who entered that value.

- For each relation type R, we create a relation ID table R_ID. Let E_1 and E_2 be the two entity types that R relates in G_s, table R_ID is defined as follows:

  $$R\_ID(\underline{id}, eid1, eid2),$$

  where id is the ID of a relation, and $eid1$ and $eid2$ are the IDs of the two related entities. Similar to converting an entity type, we first create attribute *exists* for R, then create tables A_m, A_u and A_p for attribute *exists* and each attribute of R in G_s.

A user may create an entity type (same for a relation type and an attribute), delete an entity type, or reinstate a deleted entity type. To enrich catalog data with temporal information, we also create three meta tables:

- Table $meta\_entity(\underline{ename}, start, \underline{stop}, who)$, which stores the names of the entity types that have been created. Attributes *start*, *stop* and *who* (same for those attributes in tables

meta_entity

ename	start	stop	who
person	2007-03-12 05:10:00	9999-12-31 23:59:59	M
pub	2007-03-12 05:10:10	9999-12-31 23:59:59	M
topic	2007-03-12 05:10:20	9999-12-31 23:59:59	M
...	...	...	...

(a)

meta_relation

rname	ename1	ename2	start	stop	who
interests	person	topic	2007-03-12 05:10:30	9999-12-31 23:59:59	M
write-pub	person	pub	2007-03-12 05:10:40	9999-12-31 23:59:59	M
advise	person	person	2007-05-24 16:04:27	9999-12-31 23:59:59	U_1
...	...	...	...	...	...

(b)

meta_attribute

tname	aname	category	type	start	stop	who
person	title	atomic	CHAR(100)	2007-03-12 05:10:50	9999-12-31 23:59:59	M
person	age	atomic	INT	2007-04-18 12:40:19	2007-06-10 10:30:25	U_2
person	age	NULL	NULL	2007-06-10 10:30:25	9999-12-31 23:59:59	U_3
...	...	...	...	...	...	...

(c)

Figure 4.7: Examples of meta tables: (a) $meta\_entity$, (b) $meta\_relation$, and (c) $meta\_attribute$.

$meta\_relation$ and $meta\_attribute$ below) have the same semantics as they do in an attribute table.

- Table $meta\_relation(\underline{rname}, ename1, ename2, start, \underline{stop}, who)$, which stores the names of the relation types that have been created. For each relation type R, the table also store the names of the two entity types that R relates.

- Table $meta\_attribute(\underline{tname}, \underline{aname}, category, type, start, \underline{stop}, who)$, which stores the name of each attribute (*aname*) that each entity or relation type (*tname*) has. For each attribute, the table also gives its category (atomic or set-valued) and data type (string or numeric) specifications in *category* and *type*, respectively.

Examples of the meta tables are shown in Figure 4.7.

4.3.1.3 Initializing G

To initialize G, we employ a machine-based solution M. Many such solutions exist [35]. In this work, we use the Cimple solution which is described in detail in [35]. The solution works in two steps: (1) creating an entity-relationship (ER) graph, and (2) importing the ER graph into database G.

Creating an Entity-Relationship Graph: First, a domain expert provides Cimple with a set of relevant data source. Use the community of database researchers as an example. Data sources can be home pages of database researchers, DBLP, conference pages, etc.. The expect also provides domain knowledge about entities and relations of interest. For example, *person* and *conference* are two entity types, and between them exists a relation type *give-talk*.

Then Cimple uses simple but focused automatic methods to create an ER graph of the application. Specifically, Cimple first crawls the sources at regular intervals to obtain data pages, then marks up mentions of relevant entities. Examples of mentions include people names (e.g., "D. DeWitt", "David J. DeWitt"), conference names, and paper titles. Next, Cimple matches mentions and groups them into entities (e.g., mentions "D. DeWitt" and "David J. DeWitt" refer to the same person entity). Cimple then discovers relations among the entities. As a result, Cimple creates an ER graph from the raw data sources.

DBLife is an example portal built using such a semi-automatic solution.

Importing the ER Graph into Database G: Cimple stores the ER graph in a set of XML files. To initialize database G, we first convert G_s into a set of relational tables, as described in Section 4.3.1.2. Then we use an import module to bulk load the XML data into G.

4.3.2 Creating Views over Database G

View Language Requirements: To create views over G, we must define a view language $\mathcal{L}$. We now discuss the requirements for $\mathcal{L}$. First, we note that a primary goal of structured portals is to describe interesting entities and relations in the application domain. Toward this goal, we use each wiki page W to describe an entity e or a relation r. A popular way to describe an entity e, say, is to describe a "neighborhood" of e on the ER data graph G, e.g., all or most nodes within two hops from e. Consequently, language $\mathcal{L}$ must be such that we can easily write and modify views that describe such "neighborhoods".

Second, when a user requests a wiki page W, we materialize it on the fly, to ensure the page contain the latest updates. This in turn requires materializing the view V underlying W (see

Section 4.4.3). Consequently, $\mathcal{L}$ must be such that its views can be materialized quickly, to ensure real-time user interaction.

Finally, when a user U edits a wiki page W, we assume that U may also edit the schema of view V underlying W, e.g., by removing all papers from W, U may be modifying V's schema to exclude all papers (Section 4.4.1 discusses this assumption in depth). Hence, language $\mathcal{L}$ must be such that we can modify a view schema quickly, based on user edits, to ensure real-time user editing.

A Path-based View Language: The above requirements led us to design a path-based view language $\mathcal{L}_p$. To define $\mathcal{L}_p$, first we define *data and schema paths*. Intuitively, a *data path* is a path on the ER graph G that (a) starts with an entity node e_1 and ends at an entity node e_n, and (b) retains only certain attributes for each node/edge along the path.

A *schema path* $p = ep_1.rp_2.ep_3.\ldots.rp_{n-1}.ep_n$ then specifies a set of data paths, which start with node ep_1, follow edge rp_2, etc., then end with node ep_n. To further constrain these data paths, we express each ep_i as $T_i(C_i)\{A_i\}$, meaning that (a) ep_i must have type T_i and satisfy condition C_i (which is a conjunction of conditions over the attributes), and (b) we keep only those attributes of ep_i that appear in A_i (which is a set of attribute names). T_i is required, but (C_i) and $\{A_i\}$ are optional. A missing $\{A_i\}$ means that we retain all attributes. We express each rp_i in an analogous fashion.

Example 4.2 The schema path $person(id = 1)\{name, title\}$ specifies a single data path that corresponds to person entity with id=1 and that contains only attributes name and title of this entity. The schema path, person(id=1).give-tutorial.conf{name}, specifies a set of data paths, each of which starts with a person node whose id is 1, follows an edge give-tutorial, then ends with a conf node. For each path, we retain all attributes of person node and give-tutorial edge, but retain only the name attribute of conf node.

We can now define ER views considered in this work as follows:

Definition 4.3 (Path-based ER views) A path-based ER view (or view for short when there is no ambiguity) V has a schema $V_s = (In, Ex)$, where In and Ex are disjoint sets of schema paths

over G. Evaluating V_s over G yields the view data V_d. V_d is a subgraph of G that contains only data paths that are (a) specified by some path schema in In and (b) not specified by some path schema in Ex. We refer to schema paths in In and Ex as inclusive and exclusive paths, respectively.

Example 4.4 Figure 4.3.b shows a sample V_s that has two inclusive paths and one exclusive path. This view schema selects a person e with $id = 1$, retains name and title of e, then selects all interests of e except those named "Statistics". Evaluating this view schema over the ER graph G of Figure 4.3.a produces the view data V_d in Figure 4.3.c.

We now discuss how language $\mathcal{L}_p$ satisfies the requirements outlined earlier. First, most "neighborhoods" of an entity e (e.g., all nodes within two hops of e on ER graph G) can be expressed with a set of inclusive and exclusive data paths. Hence, $\mathcal{L}_p$ allows us to quickly write views that capture such neighborhood, in an intuitive manner. Second, evaluating schema paths amounts to performing selection operations over G. Hence, views in $\mathcal{L}_p$ can be materialized quickly. Finally, if a user edits a view schema (using a wiki page), then such edits can be quickly mapped into a set of inclusive and exclusive schema paths, allowing us to modify the view schema quickly and easily.

Creating Views over ER Graph G: Now that we have defined the view language $\mathcal{L}_p$, we can discuss how Swiki uses $\mathcal{L}_p$ to create views over G. First, Swiki decides on the set of entities and relations to be "wikified". In this work, for simplicity we consider all entities, but no relations. Next, for each entity e of a particular type (e.g., person), Swiki specifies a default view schema V_s that specifies a "neighborhood" of e. Swiki thus specifies as many default view schemas as the number of entity types to be "wikified". These default view schemas are application specific. The data of the views is not stored, but will be materialized on the fly, when creating and refreshing wiki pages, which we discuss next.

4.3.3 Converting Views to Wiki Pages

Given a view V with schema V_s and data V_d as defined above, we now discuss converting V_d into a wiki page W. In the following, we introduce our novel s-slot solution. We also discuss some

other non-trivial design issues, such as the ordering of entities, the formation of URLs and the use of schema pages.

A Spectrum of Solutions: Since most current wiki data (e.g., Wikipedia) is natural text, the straightforward solution is to convert V_d into a set of natural-language sentences. For example, suppose V_d specifies that person X works for organization Y. Then we can convert this into sentence "X works for Y" in wiki page W. Knowing this template, if a user later modifies the sentence to be "X works for Y'", we can still parse it back, realize that Y has been modified to be Y', then update the underlying database G accordingly.

This was indeed the first solution we tried. It is very easy for users to edit natural-language wiki pages generated by this solution. But after extensive experiments, we found that it is difficult to extract and update structured data. The set of templates that we can use in natural language settings is somewhat limited; hence, they get reused in multiple contexts, causing many ambiguities for the extractor. Furthermore, suppose G has been updated so that X is now working for Y'. To update W with this information, we must be able to pinpoint the location of Y. This is equivalent to being able to extract Y, a difficult task, as discussed earlier.

For these reasons, we wanted a solution where *it is trivial to pinpoint pieces of structured data* contributed by V_d. A wiki page then contains multiple "islands" of structured data from V_d, in a "sea" of natural text contributed by users. We refer to these "islands" as *s-slots* (shorthand for *structured slot*). Below we describe this *s-slot solution*. In Section 4.6 we discuss how the natural-language and s-slot solutions lie at two ends of a spectrum of solutions that trade off (a) ease of user edit, (b) ease of extracting and updating structured data, and (c) ease of moving data around on wiki pages.

The S-Slot Solution: We first define the notion of attribute path. Recall that a schema path p has the form $T_1(C_1)\{A_1\}.\ldots.T_n(C_n)\{A_n\}$. We say that p is an *attribute path* iff $A_1 - A_{n-1}$ are empty sets and A_n identifies a single attribute a. Thus, p uniquely identifies attribute a. Examples of attribute paths are $person(id = 1)\{title\}$ and

$$person(id = 1).\text{write-pub}(id = 5).pub(id = 14)\{name\}.$$

An s-slot s then has the form $<\# \; p = v \; \#>$, which specifies that the attribute a uniquely identified by the attribute path p takes value v. An example of wiki text including an s-slot is

```
<# person(id=1){name}=David DeWitt #> works for
<# person(id=1).work-org.org(id=13){name}=UW #>
since 1976.
```

When a wiki page is rendered into an HTML page, only the value v of an s-slot $<\# \; p = v \; \#>$ is presented while other parts, as meta data, are suppressed. Thus the HTML presentation of the above example wiki text will display "David DeWitt works for UW since 1976".

An s-slot of $<\# \; p = v \; \#>$ can be marked with a "nodisplay" attribute as in $<\# \; p = v$ nodisplay$\#>$. In this case, the whole s-slot will be suppressed and even the value v will not be presented in the HTML page. Such s-slots are useful when the values are confidence scores used for entity ordering, as to be discussed shortly.

An s-slot of $<\# \; p = v \; \#>$ can also be marked with an "invalid" attribute as in $<\# \; p = v$ invalid$\#>$, indicating that the path p is broken and unsupported by the underlying database, and thus, the validity of the value v expired. This situation is generally caused by deletion of structured data from other related wiki pages. When a page containing invalid structured data is requested, the "invalid" attributes will be added by machine for the corresponding s-slots. $<\# \; p = v$ invalid$\#>$ will be presented in the HTML page as "v(invalid)", reminding the user of the fact and leaving him/her the right to delete the s-slot or fix the broken path.

Now let V be a view with schema V_s that Swiki has defined over database G (see Section 4.3.2). Then Swiki generates the default wiki page W for V in two steps: (a) evaluates V_s over G to obtain the view data V_d, which is a subgraph of the ER graph G, then (b) convert V_d into a wiki page W using s-slots interleaved with English text.

Step (a) is relatively straightforward. Step (b) can be executed in many different ways. In this work, we adopt a default solution. Suppose we know that view V (and thus wiki page W) describes entity e, e.g., David DeWitt. Then our default solution first generates the line $<\#person(id = 1)\{name\} = David \; DeWitt \; \#>$ as the title of the wiki page. Next, it displays the attributes

of e, then the relationships. Figure 4.3.d shows how the data graph V_d in Figure 4.3.c may have been displayed in a wiki page. In the following, we explain the algorithmic details about how to generate a wiki page W from a view data graph V_d.

The Algorithm Generating W from V_d: The algorithm presented in Figure 4.8 generates a wiki page W from a given view data graph V_d for entity e. In line 1, W is initialized to be empty. In line 2, the s-slot corresponding to the name attribute of e is made title of W. In lines 3–4, a section is created in W for other selected attributes of e, that provides the basic attributional information describing e. In lines 5–13, a section is created in W for each relationship type.

Each section is labeled properly with a uniform default look. This can be done since in building an initial portal $\mathcal{W}$, the only participating user is the portal builder, for whom the semantics of each attribute type, entity type and relationship type are transparent to him/her since he/she was the one who created the initial view schema V_s. The selection of view V delivers the builder's intention and the look of the wiki page represents his/her preferences. For the same reason, in line 7, the data paths in V_d for relationship type r can be extracted properly. Notice that V_d itself does not embed such information that how it should be decomposed and presented. Rather, the extraction mechanisms are hard-coded for each relationship section. For example, for the "writes" relationship, the paths of type *person.write-pub.pub.write-pub.person* starting at entity e are extracted from V_d.

In line 8, the extracted paths are possibly sorted if the ordering information is provided in the paths. In lines 9–13, the extracted paths are grouped such that each group corresponds to a unique instance of the relationship type r. Then, the s-slots for the selected attributes of each group form an item and the item is inserted into the section.

The portal builder has every reason to capture the preferences of the majority of users. The above hard-coded interpretation mechanism translates V_d into a default wiki page W, so that the initial portal $\mathcal{W}$ features HTML pages with a uniform look that is easy to the eyes of the majority of users. Later, W would be edited by different individuals, and this default interpretation mechanism will not be used in updating W by machine, in order not to intervene users' intentions and interpretations. Instead, all the fresh structured contents will be inserted into a special section

Input: View data graph V_d describing entity e.
Output: Wiki page W.

1. initialize W to be empty;
2. make title of W the s-slot corresponding to the name attribute of e;
3. create a section S in W for the attributes of e;
4. FOR each selected attribute type a of e other than name in V_d DO
5. insert the s-slot corresponding to a into S;
6. FOR each relationship type r of e in V_d DO
7. create a section S_r in W for r;
8. identify a set P of data paths from V_d corresponding to r;
9. IF P is sortable THEN sort P;
10. FOR each edge (e, f) corresponding to an instance of r DO
11. create an item I in S_r;
12. identify a subset $P_f \subseteq P$ of paths that share (e, f);
13. insert into I the s-slots corresponding to all selected attributes in P_f;

Figure 4.8: Generating wiki page W from view data V_d

called "New", from which users can pick up items and move them around according to their own preferences.

Ordering of Entities: Handling the ordering of entities is a non-trivial design issue. In many cases, entities have a natural ordering depending on how much they relate to a common entity. For example, the related people of a person can be ordered by the closeness of their relationships to that person. The related topics of a person can be ordered by the degree of interest and involvement of that person in those topics. As another obvious example, all the authors of a publication must be ordered by how they appear in the publication. To capture the ordering information, we assign each involving relationship a confidence score as attribute.

In order for applicable entities to appear ordered in the HTML page, the confidence score attribute needs to be selected in V_s. Then, the corresponding data paths in V_d will present this ordering information and be ordered properly by the algorithm (line 9) converting V_d to W. The corresponding s-slots in W will be marked with "nodisplay" and thus those actual confidence score values will not be displayed in the HTML page. This handling of entity ordering is not meant to be systematic and sophisticated to cover arbitrary ordering needs; rather, it focuses on simplicity and adequacy in terms of fulfilling the basic ordering functionality.

Formation of URLs: The formation of URLs raises another non-trivial design issue. For the HTML page specified by a URL of http://dblife-labs.cs.wisc.edu/wiki-test/index.php/David_DeWitt, the corresponding wiki page will have a URL of http://dblife-labs.cs.wisc.edu/wiki-test/index.php?title=David_DeWitt&action=edit. "David DeWitt" is the *page title* for the HTML page as well as the wiki page. As page title is the only replaceable element in a URL, the formation of URLs comes down to the formation of page titles.

Within the same namespace, each entity e must have a unique page title. A natural solution to achieve this uniqueness is to use entity ID's as titles. However, such page titles are neither informative to users nor cooperative with search engines. Entity names seem to be the most informative titles; however, they cannot guarantee the uniqueness since multiple entities may share the same name. In our design, a mapping table is maintained to map each entity ID to a unique page title. In general cases, entity names are used as page titles. In cases a title is used by another entity, a concatenation of entity name and ID will be used.

In particular, we create a mapping table with three fields *eid*, *title*, and *type*, storing entity ID's, page titles and entity types respectively. Both *eid* and *title* are keys. When a new entity e is inserted into the database, a default wiki page will be created for e and the mapping table is used to generate the page title. First, the entity ID of e, say 15, is checked against existing ones in the mapping table. If no duplicates, the name of e, say "David DeWitt", is then checked against existing page titles in the table. If no duplicates, 15 and "David_DeWitt" will form a tuple and be inserted into the table. Otherwise, a concatenation of "David_DeWitt" and 15, i.e., "David_DeWitt15", will be used instead as the page title. Obviously, the page titles thus-generated are guaranteed to be unique.

Use of Schema Page: As to be discussed in Section 4.4.1, we expose view schemas in wiki pages to allow user editing. Thus, a default schema page W_s will be created for each default wiki page W. W_s will have the same page title as W but under the namespace of "Schema". For example, the URL for the schema page of the wiki page of David DeWitt will be http://dblife-labs.cs.wisc.edu/wiki-test/index.php/Schema:David_DeWitt.

Since all the default schema pages of the same type differ only in entity ID, they can be automatically generated and bulk-loaded into the system when building the initial portal. In particular,

all the entities are first registered in the mapping table. Then, a default schema page is generated for each tuple in the table according to the entity type stored in the table. Next, all these schema pages are written in a single file, which is then bulk-loaded into the database supporting the wiki system, without utilizing the interface of the system.

The set of all wiki pages generated as above constitutes the initial portal $\mathcal{W}$. The next section discusses how users can contribute to this portal.

4.4 Managing User Contributions

In this section we discuss what users can edit and how to process those edits.

4.4.1 What Can Users Edit?

Consider a user U editing a wiki page W. We allow U to edit both text and structured data of W. Editing text is trivial. Editing structured data of W means U can modify or delete s-slots, or insert new ones.

In modifying an s-slot $s =<\#p = v\#>$, U can modify the attribute path p as well as value v, but is not allowed to modify the formatting characters (e.g., $<\#$, $=$, and $\#>$). If U were to do so, then the parser would fail to recognize the s-slot, and hence would interpret the modified s-slot as text, not structured data.

Let V be the underlying view of W. Conceptually, editing structured data of W means editing one or a combination of the following components: the data of V, the schema of V, the data of G, and the schema of G (denoted V_d, V_s, G_d, G_s, respectively).

In traditional settings such as RDBMS, ordinary users can only edit view data and thus also the underlying relational database data. This maps to editing V_d and G_d in our case. Should we also allow users to edit V_s and G_s? We decided to allow these actions, because there is often a natural need to do so. For example, a user U may naturally want to modify W so that it no longer *displays* emails. To do this, U must modify V_s. U cannot modify V_d because this would mean *removing* certain emails from G, not the desired effect. As another example, a user U may naturally want to

add to an entity e (described in W) a new attribute a that has not existed so far in the portal. To do this, U must modify both G_s and V_s.

The next question then is: what is the best way to allow users to modify V_s and G_s? A possible option is to expose these schemas in wiki pages, for users to edit. For example, we can expose V_s in a wiki page W_s. Then when U edits W, we interpret such edits as editing V_d, and when U edits W_s, we interpret such edits as editing V_s.

The above option would greatly reduce the ambiguity in interpreting user edits. However, we decided against it, because we found from experimentation that it is difficult for *ordinary, database-illiterate* users to remember this option. In fact, users often are not even aware of the distinction between data and schema edits. Instead, they appear to prefer to edit only the wiki page W, then rely on Swiki to assist them in executing the right kind of edit actions.

For these reasons, we allow U to edit only wiki page W, then ask U (in English) to clarify if he or she intends to edit the data or the schema. In what follows we discuss this process in detail.

4.4.2 Infer and Execute Structured Edits

Suppose user U has edited wiki page W into W'. Then we can parse W' to extract a text portion T' and a structured data portion D'. The text portion can immediately be stored in a text database T (see Figure 4.2). The structured data portion D' consists of all s-slots in W.

Next, we can merge all s-slots in D' together to obtain an ER graph that we will refer to as V'_d. Given that each s-slot maps uniquely into an attribute in the ER graph G, the merging process is relatively straightforward, and hence will not be discussed further, for lack of space. Our problem now is: given V'_d, infer what actions user U intends to execute on V_d, V_s, G_d, G_s, then execute those actions.

Basic Relational and ER Actions: To solve the above problem, we first define a set of basic actions that U can execute over V_d, V_s, G_d, G_s. For example, basic actions on V_d include modifying the value of an entity or relation attribute, and deleting an entity. Basic actions on V_s include inserting a new entity and deleting an attribute of a relationship. We have implemented each basic action as a program over the temporal relational database that stores G. The complete sets of

	Actions on V_d	**Actions on V_s**	**Actions on G_d**	**Actions on G_s**
a_1	Modify an entity attribute value	Insert an entity attribute	Modify an entity attribute value	Create an entity attribute
a_2	Modify a relation attribute value	Insert a relation attribute	Modify a relation attribute value	Create a relation attribute
a_3	Insert an entity attribute	Insert an entity	Insert an entity attribute	Create an entity type
a_4	Insert a relation attribute	Insert a relation	Insert a relation attribute	Create a relation type
a_5	Insert an entity	Delete an entity attribute	Insert an entity	Drop an entity attribute
a_6	Insert a relation	Delete a relation attribute	Insert a relation	Drop a relation attribute
a_7	Delete an entity attribute	Delete an entity	Delete an entity attribute	Delete an entity type
a_8	Delete a relation attribute	Delete a relation	Delete a relation attribute	Delete a relation type
a_9	Delete an entity		Delete an entity	
a_{10}	Delete a relation		Delete a relation	

Table 4.1: Basic relational actions on V_d, V_s, G_d and G_s

basic actions on V_d, V_s, G_d and G_s are given in Table 4.1. Appendix A give their implementations. Abusing notation, we will refer to these basic actions as *basic relational actions*, to distinguish them from the basic ER actions that we will introduce soon below.

Now given V'_d, we must infer the sequence of basic relational actions that we believe user U intends to execute. To do this in a manageable fashion, we introduce an intermediate user interface: the *ER interface*. This interface would display an ER data graph (e.g., V_d) in a graphical fashion, and allow users to execute a number of *basic ER actions*, such as modifying a node or an edge, deleting a node, etc.

The first column of Table 4.2 lists the ten basic ER actions we have defined. We have implemented each ER action as a sequence of relational actions. For example, action a_1 (see the table) translates into the sole relational action that modifies the value of an entity attribute (in both V_d and G_d).

However, it turns out that an ER action can be *ambiguous*, in that it can map into different sequences of relational actions, depending on the user intention, as the following example illustrates:

Example 4.5 Suppose a user U applies action a_8 (see Table 4.2) to delete an attribute x of, say, a person entity e in an ER graph, e.g., V_d. Then U may mean to delete x from (a) V_s, i.e., do not display x in view V, or (b) G_d, thus declaring that entity e does not have attribute x, or (c) G_s, thus declaring that attribute x does not exist for person (the entity type of e).

Basic ER Actions	V_d	V_s	G_d	G_s
a_1: Modify attribute value	✓		✓	
a_2: Insert an existing attribute	✓	✓	opt.	
a_3: Insert a new attribute	✓	✓	✓	✓
a_4: Insert an existing entity	✓	✓	opt.	
a_5: Insert a new entity	✓	✓	✓	✓
a_6: Insert an existing relationship	✓	✓	opt.	
a_7: Insert a new relationship	✓	✓	✓	✓
a_8: Delete an attribute	✓	✓	opt.	opt.
a_9: Delete an entity	✓	✓	opt.	opt.
a_{10}: Delete a relationship	✓	✓	opt.	opt.

Table 4.2: Basic ER actions

Since we do not know U's intention, if U executes action a_8, then we first ask U (in an English phrase) to choose among options (a)-(c) in the above example. Next, we translate a_8 into the appropriate sequence of relational actions, depending on U's answer. For example, if U chooses option (c), then the sequence of relational actions is: delete x from V_s, delete x from G_d, delete x from G_s.

For each ER action, Columns 2-5 of Table 4.2 show which components (V_d, V_s, etc.) that the action may modify ("opt." means "optional", depending on external conditions such as user intentions).

Mapping User Edits into Sequence of Basic Actions: With the introduction of the ER interface, our problem can be recast as follows. When user U edits the structured data portion of wiki page W, we view it to be equivalent to U editing the ER graph V_d in the ER interface, using basic ER actions. We do not know what basic ER actions U executes. But we do know the end result, which is the ER graph V_d', as described earlier.

Thus, in this perspective, U has executed a sequence $\mathcal{S}_{ER}$ of basic ER actions on the original ER graph V_d, transforming it into a new ER graph V_d'. Our task then is to "reverse engineer" $\mathcal{S}_{ER}$, by comparing V_d with V_d', then execute $\mathcal{S}_{ER}$. Figure 4.9 shows the pseudo code of our current algorithm to reverse engineer $\mathcal{S}_{ER}$.

Input: Data graphs V_d and V'_d. V_d=(E, R, A), V'_d=(E', R', A'), where E, E' are sets of entity instances, R, R' are sets of relationship instances, and A, A' are sets of attributes.
Output: Sequence of GUI actions $\mathcal{S}_{ER}$.

1. FOR each entity instance $e \in E' - E$ DO
2. IF entity type exists THEN append a_4 to $\mathcal{S}_{ER}$;
3. ELSE append a_5 to $\mathcal{S}_{ER}$;
4. FOR each relationship instance $r \in R' - R$ DO
5. IF relationship type exists THEN append a_6 to $\mathcal{S}_{ER}$;
6. ELSE append a_7 to $\mathcal{S}_{ER}$;
7. FOR each attribute $a \in A' - A$ DO
8. IF attribute type exists THEN append a_2 to $\mathcal{S}_{ER}$;
9. ELSE append a_3 to $\mathcal{S}_{ER}$;
10. FOR each attribute $a \in A - A'$ DO
11. append a_8 to $\mathcal{S}_{ER}$;
12. FOR each relationship instance $r \in R - R'$ DO
13. append a_{10} to $\mathcal{S}_{ER}$;
14. FOR each entity instance $e \in E - E'$ DO
15. append a_9 to $\mathcal{S}_{ER}$;
16. FOR each attribute $a \in A \cap A'$ DO
17. IF it has the same value in V_d and V'_d THEN append a_1 to $\mathcal{S}_{ER}$;
18. Return $\mathcal{S}_{ER}$;

Figure 4.9: Generating $\mathcal{S}_{ER}$ from V_d and V'_d

To "push" the structured edits of U into the database G, we then execute the actions of $\mathcal{S}_{ER}$ sequentially. Recall that each such action is a basic ER action (see Table 4.2), which can be ambiguous. If this happens, recall also that we resolve the problem by asking user U a disambiguating question. We then execute each basic ER action by executing the sequence of relational actions that it maps to, as described earlier.

A minor problem is that $\mathcal{S}_{ER}$ is not unique. Given any two V_d and V'_d, multiple sequences of actions $\mathcal{S}_{ER}$ may exist that all transform V_d into V'_d. Fortunately they all have the same effect, as this theorem shows:

Theorem 4.6 Let $\mathcal{S}_1, \ldots, \mathcal{S}_k$ be all sequences of basic ER actions that transform a V_d into a V'_d. Then when executing any $\mathcal{S}_i$, the set of questions we pose to user U will be the same for all i. If U gives the same answers to these questions, then executing any $\mathcal{S}_i$, $i \in [1, k]$, results in the same V_d, V_s, G_d and G_s.

4.4.3 Propagate Structured Edits

Let W_1 and W_2 be two wiki pages that describe two researchers A and B, respectively. Suppose A and B share one publication p. So p appears in both W_1 and W_2. Now suppose that a user U has edited p in W_1. When should we update p in W_2? In general, once a user has edited the structured data portion of a wiki page W, how should we propagate this edit to other pages?

A solution is to immediately refresh other pages, e.g., page W_2 in the above example. We call this *eager propagation*. This solution ensures timely updates of pages, but can raise tricky concurrency control issues. Hence, we adopt a *lazy propagation* approach, where we refresh a page, say W_2, only when a user requests the page again. At that moment, we rematerialize the page from the structured database G and the text database T. Section 4.6 empirically shows that we can refresh pages on the fly quickly, in a few seconds, thus making this lazy approach a practical solution.

4.5 Managing Multiple Users and Machine

For completeness we will briefly touch on the key problems of managing multiple users and machines as they contribute to the portal.

First, we must manage concurrent editing of a wiki page by multiple users, or concurrent editing of some structured data pieces (e.g., a paper) that appear in multiple wiki pages. In this work we employ the optimistic concurrency control scheme of Wikipedia for this purpose.

Next, we must detect and remove malicious users. To do this, we employ a hierarchy of users, reminiscent to the Wikipedia solution for the same problem. Specifically, we require users log in to edit, and employ a set of editors whose job is to monitor most active wiki pages.

Finally, if a user U has modified a data item X, can machine M overwrite U's modification, and if so, then when? Our current solution allows M to overwrite U's data only for certain pre-specified data types (e.g., certain attributes of person), if M is sufficiently confident in its data. For all other data types, we do not allow M to overwrite U's modification, but allow it to add a suggestion next to U's modifications, in parentheses, e.g., "age is 45 (according to M, age is 47)".

4.6 Empirical Evaluation

To evaluate Swiki, we have been applying it to build a wikipedia-like portal for the database community (see [1] for the current portal, still under continuous development). We now report on preliminary experiments with this portal, which demonstrate the potentials of Swiki and suggest research opportunities.

Building an Initial Portal: We began by employing DBLife as machine M (see Section 4.3). It took a two-person team four weeks to develop DBLife from scratch. DBLife was first deployed on May of 2005, and has been on "auto pilot" since, requiring only about one hour of maintenance per month (for more details, see [35]). Each day DBLife crawls 10,000+ database research related data sources, extracts and integrates the data, to generate a daily ER data graph.

We used one such daily ER data graph A (98M of XML data) to initialize the structured database G. G's schema has five entities and nine relationships, and G's data contains 164,043 entity instances and 558,260 relation instances, for a total size of 413M. This size is greater than the ER data graph size of 98M due to the extra space needed to store temporal information. It took 216 seconds to load A into G, and 183 minutes to generate and store all wiki pages (164,043 pages for entities). These results suggest that we can create moderate-size initial portals (a one-time task) with relatively little efforts.

Next, we wanted to know if the initial portal can be maintained efficiently, assuming no user contributions yet. We found that over 10 days, as DBLife contributed data to the structured database G, G's size increased from 413M to 600M. This was somewhat surprising, because DBLife data should not have changed so much over 10 days. Upon a closer inspection, we found that the confidence scores of most relation instances in G (e.g., person X is related to person Y with score .8) were changed by DBLife everyday, due to the changing raw data (retrieved by DBLife). Hence, the confidence scores of most relation instances in G were updated everyday, leading to a rapid growth in G's size (recall that G is a temporal database that does not allow update in place, hence changes are added to G). Once we disallowed updating confidence scores, then G grew very slowly (by less than 5M). Thus, this experiment suggests that the current design of G is efficient

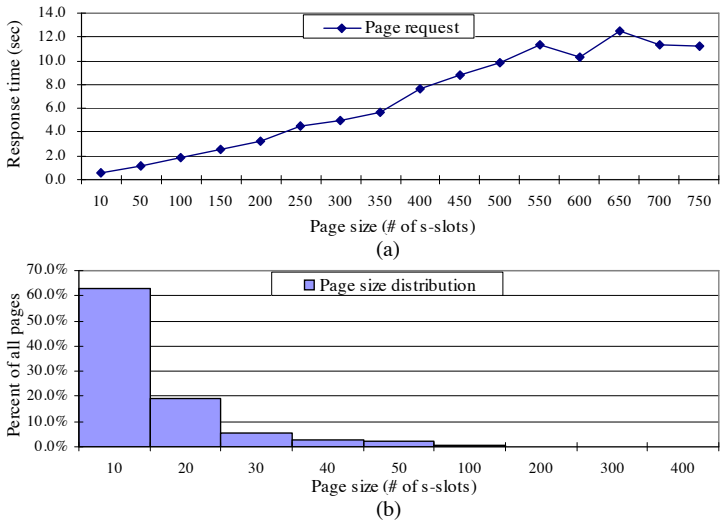


Figure 4.10: Time to request a wiki page and distribution of page size.

for maintaining all aspects of the initial portal over time, except for confidence/uncertainty scores. It is thus useful to examine how to modify the temporal design of G to efficiently accommodate frequent changes in uncertainty scores.

Expressive Power of the S-Slot Wiki Language: In the current DBLife system, each user superhomepage is a structured view V over the underlying structured database. We found that the s-slot wiki language (Section 4.3.3) was sufficiently powerful to enable us to express all structured data pieces in such views in wiki pages, except two types of data pieces: top-k and aggregate. A top-k data piece is technically a view that lists the top k items of a ranked list, e.g., the top three authors, cited papers, etc. An aggregate data piece is an aggregate view such as the total number of papers per author, or the total number of citations.

We found that top-k and aggregate views also appear in many other structured Web portals. Thus, any future attempt to extend wiki languages with structured constructs must address the problem of expressing such views. The challenge then is how to efficiently update such views.

Efficiencies of User Interaction: In the next step, we examined how fast users can interact with the portal. Figure 4.10.a shows the time it takes from when a user requests a page W until when W is served. Note that to ensure freshness, we materialize W on the fly, from the underlying structured

# of s-slots = 52 time in sec					
Type of edits	**5 edits**	**10 edits**	**15 edits**	**20 edits**	**25 edits**
modification	0.258	0.266	0.275	0.283	0.291
insertion	1.041	1.314	1.583	1.826	2.115
deletion	1.012	1.122	1.253	1.363	1.483
# of s-slots = 196 time in sec					
Type of edits	**5 edits**	**10 edits**	**15 edits**	**20 edits**	**25 edits**
modification	1.183	1.209	1.231	1.247	1.266
insertion	1.971	2.214	2.436	2.662	2.855
deletion	1.615	1.633	1.649	1.665	1.681

Figure 4.11: Time to process user edits on a wiki page.

database G and text database T (Section 4.4.3). Hence, it is critical that such materialization can be done quickly, to ensure real-time user interaction.

The results show that request time increases linearly w.r.t. page size, measured in the number of s-slots in the page, and stays small, e.g., under 2 seconds for page sizes up to 150. Figure 4.10.b shows that the vast majority of current pages have a size under 50 (the first five bars of the figure), and thus incur under 1 second request time. This result suggests that we can materialize wiki pages quickly, and that the lazy update approach (Section 4.4.3) can work well in practice.

Since processing user edits requires us to translate these edits across different user interfaces and then to invoke the underlying relational database, we wanted to know if it can be done efficiently. Figure 4.11 shows the time it takes from when a user submits his/her edits until when the edits have been processed, i.e., updates on V_s, V_d, G_s, G_d, if any, have been carried out. This time does not include the time users spent answering disambiguating questions (Section 4.4.2). The top table of the figure shows edit times over a wiki page with 52 s-slots (each time is averaged over 10 runs). Here each edit is a user action that affects a single s-slot.

The bottom table of the figure shows similar edit times, but over a wiki page with 196 s-slots. In both cases, the results show that the edit times remain small, under 2.2 seconds for the small wiki page and 2.9 seconds for the large wiki page. This suggests that Swiki can process user edits efficiently.

Ease of User Interaction: Next, we evaluated how easy it is for users to edit structured data in a wiki page W. We conducted a preliminary experiment with 6 users, where each user was asked to

Editing tasks	Time (sec)	Accuracy
editing a sentence of free text	13.2 (10~21)	100%
modifying a data path	16.4 (10~30)	100%
inserting a data path	52 (30~60)	100%
inserting two bonded data paths	55 (30~85)	100%
inserting a paragraph of data paths	152 (60~240)	100%
deleting data paths	36.6 (15~60)	100%

Figure 4.12: User performance on several editing tasks.

edit a certain item on the HTML representation of W. To do so, they had to go to W, locate and then edit the appropriate piece of structured data. We measured how long it took them to finish the given editing tasks and the correctness of the results. For comparison purposes, we also asked users to edit some free text.

Figure 4.12 shows that 100% correctness was achieved for all the editing tasks. Editing time is measured from when the edit button is clicked until when the new HTML page is rendered. Figure 4.12 shows the average and range of recorded editing time over all the users. The results show that the simplest structured data editing task, modifying a data path (modifying an attribute), took comparable time to editing a sentence of free text.

Inserting a data path generally involves adding several entities and relationships to the database. Users need to type a complete legal path. Inserting two bonded data paths is a bit more complex since users need to make sure that several entities (or relationships) are assigned the same id. Inserting a paragraph of data paths is probably the most complex task that generally involves multiple bonded data paths. Specifically, the users were asked to add a publication with a title, an ordered author list, and the conference, year, page, and citation information. The results show that the editing time is very reasonable considering the high complexity of the task.

Deletion of data paths would generate some ambiguities since the user may mean to delete the structured data from the underlying database or just from the wiki page. Thus after the user clicks the submit button, several questions may be presented as radio buttons to clear the possible ambiguities. Deleting a single data path or many data paths would take similar amount of time from the editing point of view. The only difference is the number of questions to ask.

This experiment is strictly preliminary. But it does suggest that the current solutions may already be adequate in the sense that users are able to correctly execute the various editing tasks within a reasonable amount of time.

The experiment also suggests that it may be even easier for users to edit if we introduce some macro that hide the details of the structured data and make the structured data look clean. This point was confirmed by the users' qualitative feedback on how convenient it is to use the system. On a scale of 1 (least convenient) to 5 (most convenient), the current system scored an average of 2.5. A typical comment is that while the system is easy to learn and functioning well, it is verbose. These comments meet our expectations since our goal for the current version focuses almost exclusively on the adequacy instead of convenience.

In general, as commented in Section 4.3.3, a lesson we learned from our current Swiki experience is that there is a spectrum of solutions on how structured data can be represented in wiki pages. Our s-slot solution represents one spectrum end and the natural-language solution (see Section 4.3.3) the other. In between we can have solutions that present structured data using, e.g., XML formats (in wiki pages).

The key tradeoff factors for these solutions include (a) how easy it is for users to edit, (b) how easy it is for machines to re-extract structured data, and (c) how easy it is for users to move various pieces of structured data around, i.e., rearrange them in the wiki page.

The s-slot solution appears best for (b) and (c), and so-so for (a). The natural-language solution is best for (a), so-so for (c), and difficult for (b). An XML-like solution appears best for (a) and (b), but so-so for (c). Developing more solutions, evaluating them, and selecting a good one is an interesting future research direction.

4.7 Summaries

In this chapter, we have described Swiki, an approach that employs both "machines" and human users to build structured Web portals. This new hybrid machine-human approach can bring significant benefits. It can achieve broader and deeper coverage, provide more incentives for users to contribute, and keep the portal more up to date, with less user efforts. We have applied Swiki to

build a wikipedia-like portal for the database community [1]. We reported on our experience with this portal that demonstrates the potentials of Swiki and suggests many research opportunities.

Indeed, it is clear that our work here has only scratched the surface of this direction (of combining "machines" and human to build structured wikipedias). Virtually any problem that we have discussed can be "drilled down" deeper. Example problems include: (a) extending the s-slot wiki language to handle top-k and aggregate views and studying updating for such views, (b) developing "macros" that hide the low-level structured constructs to allow users to edit certain structured data pieces more efficiently, (c) developing efficient eager-update-propagation schemes, (d) developing better solutions to handle machine updates to data already modified by users, and (e) learning how to leverage user edits to improve the extraction and integration accuracy of machines.

Chapter 5

Efficiently Incorporating User Feedback into IE/II Programs

In Chapter 4, we described how to open up a structured Web portal to users for feedback, and how the portal can automatically interpret user feedback and incorporate it. The solution was designed for improving the end IE/II results, which are the final outputs of IE/II programs. Besides the output data, the input and intermediate data of the programs can also benefit from user feedback. In this chapter, we describe a solution [23] for users to directly provide feedback and for IE/II programs to automatically incorporate such feedback. Our solution works as follows. A developer uses hlog, a declarative IE/II language, to write an IE/II program. Next, the developer writes declarative user feedback rules that specify which parts of the program data (e.g., input, intermediate, or output data) users can edit, and via which user interfaces. Next, the so-augmented program is executed, then enters a state of waiting for and incorporating user feedback. Given a piece of user feedback on a data portion of the program, we show how to automatically propagate the feedback to the rest of the program, and to seamlessly combine it with prior user feedback. We describe the syntax and semantics of hlog, a baseline execution strategy, and then various optimization techniques. Finally, we describe experiments with real-world data that demonstrate the promise of our solution.

5.1 Introduction

Over the past decade, several declarative IE/II languages, such as UIMA [46], GATE [31], AQL [77], and xlog [80], have been proposed for writing IE/II programs. These works show that IE/II programs written in such languages are easier to develop, debug, and maintain than

those written in procedural languages such as Perl and Java. Many other works then examine how to optimize programs written in such languages [56, 80, 77], to execute them effectively over evolving data [24, 25], to make them best-effort [79], to add provenance [19, 55], among others. IE/II programs have started to make their way into large-scale real-world applications, in both academic and industrial settings [43].

IE/II programs written in these declarative languages have proven highly promising, but they still suffer from a glaring limitation: *there is no easy way for human users to provide feedback into the programs*. To understand why user feedback is critical, consider DBLife, a real-world IE/II application that we have maintained for over the past few years [36]. DBLife regularly crawls a large set of data sources, extracts and integrates information such as researchers' names, publications, and conferences from the crawled Web pages, then exposes the structured information to human users in the form of a structured Web portal. Since automatic IE and II are inherently imprecise, an application like this often contains many inaccurate IE/II results, and indeed DBLife does. For example, a researcher's name may be inaccurately extracted, or the system may incorrectly state that X is chairing conference Y. Being able to flag and correct such mistakes would significantly help improve the quality of the system. And given that at least 5-10 developers work on the system at any time (a reasonable-size team for large-scale IE/II applications), the developer team *alone* can already provide a considerable amount of feedback. Even more feedback can often be solicited from the multitude of users of the system, in a Web 2.0 style.

The problem, however, is that there is no easy way to provide such feedback. The current "modus operandi" is that whenever one of us (developers) finds a mistake (e.g., the name "D. Miller R" should actually be just "Miller R"), we email a designated developer. If we receive emails from users about mistakes, we forward them to the designated developer as well. The "victim" developer then delves into the internals of the IE/II program to locate and correct the mistake, and then restarts the program to propagate the correction. Needless to say, this developer soon becomes overwhelmed and resents at having to do all of the mundane work, and this solution obviously does not scale with the amount of user feedback.

Chapter 4 studied how to automate the process of user feedback on the output of IE/II programs, but the solution laid out does not address the problem of incorporating user feedback into the programs. A better solution then is to provide *automatic ways* for developers and users alike to provide feedback on all the pieces of the data that can benefit from feedback. For example, if a user finds an IE/II mistake in intermediate results, she can report it using a form interface, then the system can automatically incorporate the report by evaluating the correction submitted and then propagating the correction to the rest of the program.

This is also the approach we take in this chapter, and our goal is to develop a general and efficient solution. Our basic idea is as follows. After writing an IE/II program, the developer writes a set of declarative *user feedback rules* to specify which data portions D of the program (e.g., input data, intermediate data, or output data) users can edit[1], via which user interfaces (UIs). Then when executing the program for the first time, the system materializes and exposes D via these UIs. Given a user edit, the system updates D, propagates the update to the rest of the program, then waits for the next user edit. The following tiny example illustrates the above idea.

Example 5.1 Suppose the developer wants to crawl the set of data sources listed in Table *dataSources* of Figure 5.1.a to discover research talks. Then the developer may start by writing a program in a declarative IE/II language. Figure 5.1.b shows such a sample program in the xlog language (see Section 5.2.1 for details). Roughly speaking, this program crawls the data sources (each to the specified crawl depth) to obtain a set of Web pages (in relation *webPages(p)*; see Rule R_1). Next, it extracts titles and abstracts from the Web pages (Rules R_2 and R_3, respectively). Finally, it outputs only those (title,abstract) pairs where the title appears immediately before the abstract (Rule R_4).

Next, the developer may write a set of user feedback rules, such as Rules $R_5 - R_7$ in Figure 5.1.c. Rule R_5 creates a view *dataSourcesForUserFeedback(url,crawl-depth)* from Table *dataSources(url,crawl-depth,date)*, then exposes this view via a spreadsheet UI for users to edit. Note that this view does not allow users to edit data sources added to the system before 1/1/2009 (e.g., because those data sources have been vetted by the developers). Note also that users can

[1]Henceforth we use "users" to refer to both developers and system users.

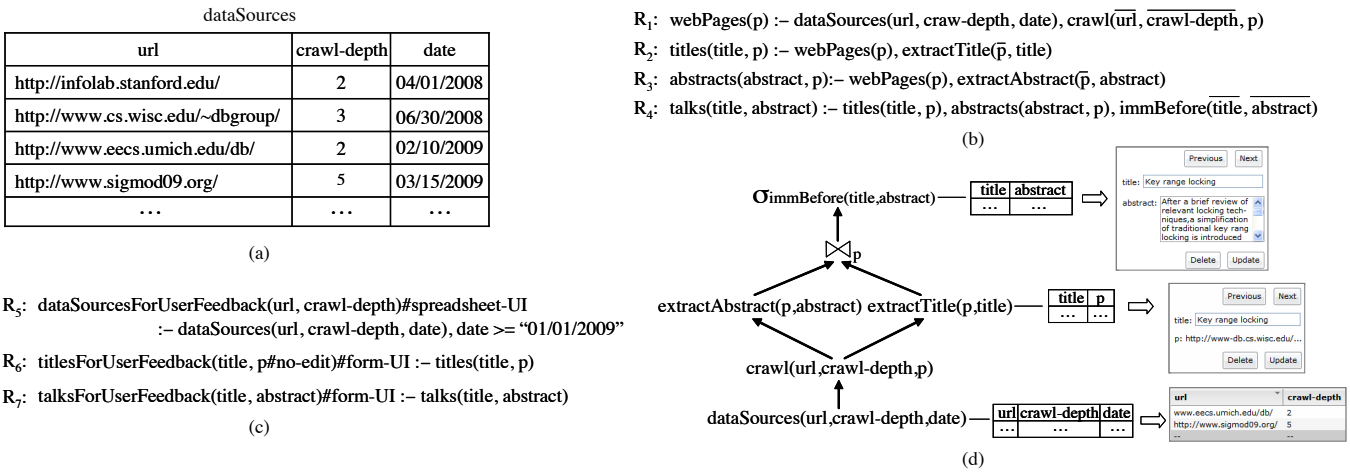


Figure 5.1: An illustration of our approach: given the set of data sources in (a), a developer U writes the IE/II program P in (b) to extract titles and abstracts of talks from the data sources; next U writes the user feedback rules in (c) to specify which parts of P users can edit and via which UIs; the system then executes P and exposes the specified data portions for users to edit, as shown in (d).

easily add new data sources by adding new tuples to the view. Similarly, Rules R_6 and R_7 allow users to edit titles and (title,abstract) pairs, respectively, using a form UI. Here, notation $p\#$no-edit in Rule R_6 states that p (i.e., the Web page in which a title appears) can be inspected but not edited by users.

The system now proceeds to execute the program for the first time. Conceptually, it compiles the program into the execution plan in Figure 5.1.d, then evaluates this plan "bottom up", in a fashion similar to evaluating relational execution plans [80] (see also Section 5.2.3). During the evaluation, the system also materializes and exposes the views specified by Rules $R_5 - R_7$ via the appropriate UIs, for subsequent user feedback (see Figure 5.1.d).

After the initial execution, the system then enters a loop of obtaining and incorporating user feedback. For example, after a user has modified the crawl depth of a url in the view *dataSources-ForUserFeedback(url,crawl-depth)*, the system would modify the base table *dataSources(url,crawl-depth, date)* accordingly, then propagate this modification by re-evaluating the execution plan. It then waits for the next user feedback, and so on.

As described, the above approach provides an automatic way to incorporate user feedback. Users can now provide feedback directly to the system instead of waiting for developers to manually incorporate it. Realizing this approach, however, raises many challenges. In this chapter, we identify these challenges and provide initial solutions.

We begin by considering how to model IE/II programs and user feedback, and how to incorporate such feedback. To address these issues, we develop hlog, a declarative language for writing "user feedback aware" IE/II programs (such as the one in Figure 5.1). hlog builds on xlog [80], and thus can be viewed as a Datalog extension, equipped with *declarative user feedback rules*. Incorporating user feedback into hlog programs turned out to be quite tricky. To see why, consider again the program in Figure 5.1.b. Suppose a user U_1 has deleted a tuple (t, a) from the output table *talks*. Suppose later a user U_2 inserts a new data source tuple (u, c, d) into the input table *dataSources*. Consequently, we re-execute the program to propagate U_2's update from table *dataSources* to table *talks*. A straightforward re-execution however will re-introduce the deleted tuple (t, a), "wiping out" the update of U_1. Furthermore, what if when processing the new data source tuple (u, c, d), the program discovers the same talk (t, a)? Should we keep this tuple, or delete it according to U_1's feedback? To address these problems, we develop a *provenance-based* solution for interpreting and incorporating user feedback.

After defining the syntax and semantics of hlog, we develop a baseline solution for executing hlog programs. We show how to store and manipulate tuple provenances, as well as user feedback. We discuss in particular the trade-offs between maximizing the amount of user feedback incorporated into a program and minimizing its execution time.

Finally, we develop a set of optimization techniques to speed up the baseline solution for executing hlog programs. First we examine how to execute such programs incrementally, after each user feedback. Incrementally updating relational operators (e.g., σ, π and $\bowtie$) has been studied extensively (see the related work section). Incrementally updating IE/II operators is more difficult, due to their "blackbox" nature. To address this problem, we identify a set of incremental properties that the developer can use to characterize IE/II operators. Once the developer has identified such properties, we can automatically construct incremental update versions for these operators.

The second set of optimization techniques that we develop concerns concurrency control. Multiple users may happen to view and update the exposed program data at the same time. To ensure the consistency of the program data, we need to enforce concurrency control (CC). To do that, we could require developers to force all data and IE/II computation into an RDBMS and use its CC capabilities. In practice, however, developers usually choose not to do so (at least not today) for ease-of-development or various performance reasons. Hence we seek to develop CC solutions outside RDBMS. In this chapter, we show how to explore the graph structure of an IE/II program to design efficient CC mechanisms.

In summary, we make the following contributions:

- Introduce the problem of allowing users to edit the data in an IE/II program to improve the quality of the program results.
- Develop a declarative language (hlog) with well-defined semantics that allows developers to quickly write IE/II programs with the capabilities of incorporating user feedback.
- Develop a solution to execute programs written in hlog.
- Propose optimization techniques for enhancing the performance of IE/II programs in terms of runtime and concurrency degree.
- Conduct extensive experiments over real-world data that demonstrate the promise of the proposed approach.

5.2 Syntax and Semantics

In this section we describe the syntax and semantics of hlog, our proposed declarative language to write "user feedback aware" IE/II programs. We first describe xlog, a recently developed Datalog variant for writing declarative IE programs [80], then build on it to describe hlog. For ease of exposition, we will focus on IE programs, deferring the discussion of II aspects to Section 5.2.4.

5.2.1 The xlog Language

We now briefly describe xlog (see [80] for more details). Like in traditional Datalog, an xlog program P consists of multiple *rules*. Each rule is of the form p :– $q_1, \cdots, q_n$, where p and q_i are predicates, p is the head of the rule, and q_i's form the body. Each predicate in a rule is associated with a relational table. A predicate is *extensional* if its table is provided to program P, and is *intensional* if its table must be computed using rules in P.

Language xlog extends Datalog by supporting procedural predicates (*p-predicates*) and functions (*p-functions*), as real-world IE often involves complex text manipulations that are commonly implemented as procedural programs. A p-predicate p is of the form $p(\overline{a_1}, \cdots, \overline{a_n}, b_1, \cdots, b_m)$, where a_i and b_j are variables. Predicate p is associated with a procedure g (e.g., written in Java or Perl) that takes as input a tuple $(u_1, \cdots, u_n)$, where u_i is bound to a_i, $i \in [1, n]$, and produces as output a set of tuples $(u_1, \cdots, u_n, v_1, \cdots, v_m)$. A p-function $f(\overline{a_1}, \cdots, \overline{a_n})$ takes as input a tuple $(u_1, \cdots, u_n)$ and returns a scalar value. The current version of xlog does not yet support recursion nor negation.

Example 5.2 Figure 5.1.b shows an xlog program P with four rules $R_1 - R_4$ that finds talks from a set of data sources. P has one extensional predicate (*dataSources*), four intensional predicates (*webPages*, *titles*, *abstracts*, and *talks*), three p-predicates (*crawl*, *extractTitle*, and *extractAbstract*), and one p-function (*immBefore*).

The p-predicate *crawl(*$\overline{url}$*,*$\overline{\textit{crawl-depth}}$*,p)* for example takes as input a url u and a crawl depth d (e.g., 3), crawls u to the specified depth, then returns all tuples (u, d, p) where p is a page found while crawling u. As another example, the p-function *immBefore(*$\overline{\textit{title}}$*,*$\overline{\textit{abstract}}$*)* returns true only if the input title appears immediately before the input abstract.

The output of an xlog program P is then the relation computed for a designated head predicate, as illustrated in the following example:

Example 5.3 Consider again program P in Figure 5.1.b with *talks* being the designated head predicate. Conceptually, for each url in *dataSources*, P applies the (procedure associated with)

crawl predicate to crawl that url to a pre-specified depth. This yields a set of Web pages p (see Rule R_1). Next, P applies the *extractTitle* predicate to each Web page p to extract talk titles (Rule R_2). Similarly, P applies *extractAbstract* to extract talk abstracts (Rule R_3). Finally, P applies *immBefore* to each pair of title and abstract and outputs only those pairs where *immBefore* evaluates to true (Rule R_4).

5.2.2 The hlog Language: Syntax

We now describe hlog. To write an hlog program, a developer U starts by writing a set of IE rules that describes how to perform the desired IE task. These rules form an xlog program P. U then writes a set of user feedback rules, or *UF rules* for short, that describes how to provide feedback to the data of program P.

The data of P falls into three groups, input, intermediate, and output data, as captured in the tables of the extensional, intensional, and head predicates, respectively. Today many IE applications allow editing only the input and output data. We found however that editing certain intermediate data can also be highly beneficial, because correcting an error early can drastically improve IE accuracy "down the road".

Furthermore, the boundary between intermediate and output data is often blurred. Consider for instance an IE program that extracts entities from text, discovers relations among entities, then outputs both entities and relations as the final results. Here entities are both intermediate results (since the program builds on them to discover relations) and final results. Clearly, correcting the entities can significantly improve the subsequent relation discovery process.

Consequently, in hlog we allow users to edit all three groups of data, that is, the extensional, intensional, and head predicates. Suppose U has decided to let users edit such a predicate p. Then U writes a UF rule of the form

$$v\#w :\!- p, q_1, \cdots, q_n.$$

This rule specifies a view $v :\!- p, q_1, \cdots, q_n$ over p, so that users can only inspect and edit p's data via the view v. Here each q_i is a built-in predicate "a op b", where a is an attribute of p, op is a primitive operator (e.g., "=", ">"), and b is an attribute of p or a constant. As such, v

is a combination of selections and projections over p. In this work we consider only such views because they are *updatable*: user edits over them can be unambiguously and efficiently translated into edits over p.

Using the notation $v\#w$, the UF rule also specifies that users can edit view v via a user interface (UI) w. We assume that the system has been equipped with a set of UIs (e.g., spreadsheet, form, wiki, and graphical), and that w comes from this set. Formally, we define a user interface w as a pair $\langle f_{out}, f_{in} \rangle$, where f_{out} is a function that renders the data of a view v into the format that w can display, and f_{in} is a function that translates actions users perform on w into operations (queries and updates) over v. Consider the spreadsheet interface for example. Here the f_{out} function converts view data into a spreadsheet file that the interface can read and display. The f_{in} function then translates user actions, such as deleting a row from a spreadsheet, into view updates, such as deleting a tuple from v. We discuss user actions in more detail in Section 5.2.3.2.

Example 5.4 UF rule R_5 in Figure 5.1.c specifies view *dataSourcesForUserFeedback* over predicate *dataSources*. The view allows users to edit only data sources added to the system on or after 1/1/2009 (e.g., possibly because all data sources added earlier have been vetted by the developer), via a spreadsheet UI. UF rule R_6 allows users to edit extracted titles via a form UI. Here *p#no-edit* means that users can inspect but cannot edit the url p.

5.2.3 The hlog Language: Semantics

In the following, we first describe a baseline semantics that is straightforward but fails to incorporate previous user feedback. Then we describe a better semantics that removes this limitation.

5.2.3.1 Baseline hlog Semantics

Recall from the introduction that after developer U has written an hlog program P, the system executes P for the first time, materializing and exposing certain data portions D of P for user edits. It then enters a loop of waiting for and processing user feedback. In what follows we describe these steps in detail.

Initial Execution: Given an hlog program P, first we compile the xlog portion of P (i.e., the IE rules) into an execution plan G. We omit the details of this compilation; see [80] for a complete description. Since the current version of xlog does not yet support recursion nor negation, the execution plan G can be viewed as a directed acyclic graph (DAG), with "leaf nodes" at the bottom and a "root node" at the top. Figure 5.2 shows a sample execution plan (reproduced from Figure 5.1.d) for the xlog program in Figures 5.1.b. Note that the internal nodes of such a plan are either relational or (procedural) IE operators.

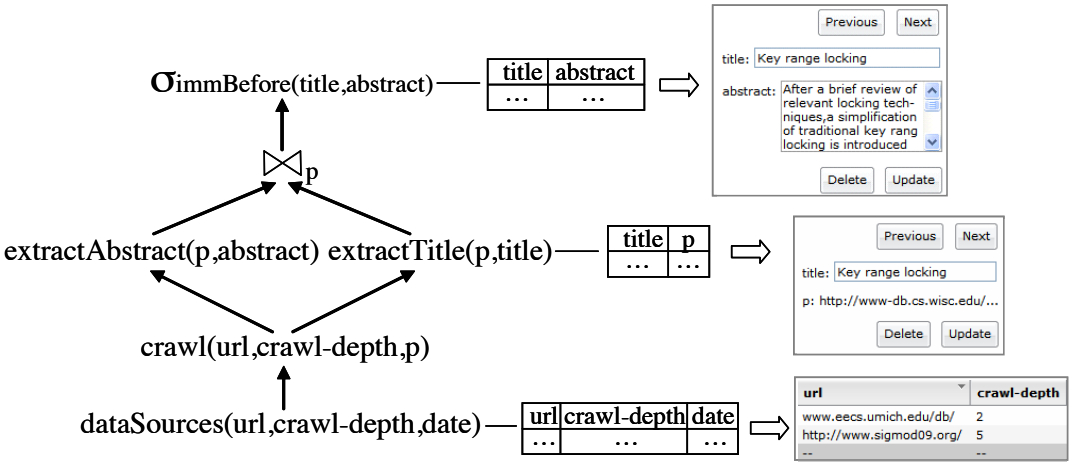


Figure 5.2: Execution graph of the hlog program in Figures 5.1.b-c, as reproduced from Figure 5.1.d.

Next, we evaluate the execution plan G in a "bottom up" fashion, starting with the leaf nodes. During the evaluation we materialize and expose certain data portions D of P via certain UIs. Specifically, if the hlog program P contains a UF rule that involves view v over predicate p and UI w, we materialize the data of p and the data of v, and then expose the data of v via UI w. Note that this differs from a traditional xlog (or RDBMS) execution, where intermediate data typically is not materialized (unless for optimization purposes). Here, we must materialize the data of p and v so that later we can incorporate user feedback. Figure 5.2 shows how the data of three predicates has been materialized and exposed, according to the UF rules $R_5 - R_7$ in Figure 5.1.c.

Loop of Waiting for and Processing User Feedback: After the initial execution, we enter a loop of waiting for and processing user feedback. Suppose a user performs an update M over a view v (e.g., modifying, inserting, or deleting a tuple; see Section 5.2.3.2). Then we translate this update M over view v into an update N over the "base" predicate p. In the next step, we start a *user feedback transaction*, or *transaction* for short. This transaction performs the update N on the (materialized) data of p. Next, it propagates the update "up" the execution graph G. That is, suppose the data from p is part of the input to an operator q of G, then we re-execute q with the newly revised p. Next we re-execute operators that depend on q, and so on. The transaction terminates after we have re-executed the root-node operator. We then wait for the next user transaction, and so on.

It is important to note that we consider propagating a user update only *up* the execution graph. Propagating update *down* the execution graph would require being able to "reverse" the input-output of IE blackboxes (i.e., given an output, compute the input). We believe requiring IE blackboxes to be "invertible" may be too strong a requirement.

Since multiple users may provide feedback at the same time, we need a way to enforce concurrent execution of the user transactions. To do that, we could require developers to force all data and IE/II computation into an RDBMS and use its CC capabilities. In practice, however, developers usually choose not to do so (at least not today) for ease-of-development or various performance reasons. Hence we seek to develop CC solutions outside RDBMS. For now, we adopt a simple CC solution: a user transaction T will x-lock the whole execution graph at the start (of its execution) and unlock the graph after the finish. Call this solution graph-locking. (See Section 5.4 for more efficient CC solutions.)

Let $\mathcal{T} = \{T_1, T_2, \ldots, T_n\}$ be a set of user transactions started and finished during the time period $[x, y]$. It is easy to see that the above algorithm guarantees a serial execution of the transactions. Hence, at the end of the time period, the final output of graph G (i.e., the output of the root operation) incorporates all user updates encoded in $\mathcal{T}$, in some serial order. This semantics is well defined. However, its notion of incorporating user updates is severely limited, in that it often clobbers previous user feedback: when an operator p in the execution graph is re-executed, its new

output will simply replace the old one; thus any previous user updates over p's output will be lost. In the next subsection, we consider how to address this problem.

5.2.3.2 Extending Baseline Semantics to Handle Previous User Feedback

Preliminaries: We start with three preliminaries. First, we observe that any user feedback F in our framework can be viewed as feedback over the *output* O of some (relational or IE) operator p in the execution graph G. (This is always true except when F is over a leaf node of G, that is, an extensional table. But we can easily handle this by pretending that each extensional table is the output of some dummy operator.)

Second, to be concrete, we will assume that a user update F (we use "user update" and "user feedback" interchangeably) can be only one of the followings: deleting a tuple $t \in O$, modifying a tuple $t \in O$ to t', or inserting a tuple t into O, where O is the output of an operator p. More complex types of user feedback exist, and our current framework can be extended to deal with many of these. But we will leave an in-depth examination of this issue as future work.

As a final preliminary, we will develop this subsection assuming that operator p is unary, that is, it takes as input a single table I. The notions we will develop can be generalized in a straightforward fashion to the case where p takes as input a set of tables.

Tuple Provenance: We are now ready to consider how to save user updates on O, and then apply them when p is re-executed. A simple solution is to save a user update F as an update *operation* (e.g., insertion, deletion, or modification), and then apply the operation when p is re-executed. This solution, however, may incorporate user updates incorrectly, as the following example illustrates.

Example 5.5 Suppose that given input I, p produces output O, and that a user has deleted an incorrect tuple t from O. Now suppose that we modify input I into I', and that re-executing p given I' produces output O', which consists of a single tuple t. Then using the above solution we should delete t from O'. But what happens if t is indeed the correct output for $p(I')$? In this case we have incorrectly applied an old user update.

This example suggests that we should interpret a user update on an output tuple based also on the *provenance* of that tuple from the input data. Specifically, suppose p takes as input a single table I and produces an output table O. Then we define the *provenance* of a tuple $t \in O$ to be the set S_t of tuples in I that p used to produce t.

We require that given any input I, p produces not just the output O, but also the provenances of all tuples in O, such that each tuple t has a unique provenance, that is, a set S_t in input I. (Note that there may be multiple sets of input tuples that p can use to produce t; S_t is the actual set p used.) If p is a relational operator, then it is relatively easy to modify p to do so. If p is an IE operator, then it is more difficult, but often do-able, especially for the creator of p. In the worst-case scenario, we can take the whole input table to be the provenance for each tuple in the output. (This happens when we have no knowledge about p; thus each input tuple may possibly contribute to the derivation of each output tuple.)

Interpreting and Incorporating User Updates: Having defined provenance, we can now interpret user update as follows. Suppose a user deletes a tuple t from output O of operator p. Let $S_t \subseteq I$ be the provenance of t, and $M \subseteq O$ be the set of tuples whose provenance is S_t, denoted as $M = \hat{p}(S_t)$. Here, $\hat{p}(\cdot)$ is a function over the provenances recorded by p; it takes as input a provenance S and returns the output tuples that p produced from S.

Let M' be the result obtained after deleting tuple t from M. Then we assume that by deleting the tuple t from output O, the user means to state that $\hat{p}(S_t)$, the output tuples that p produced from S_t, should really be M', not M.

Under this interpretation, the above user update can be (conceptually) saved as a tuple (S_t, M, M'), and we can incorporate this update in case p is re-executed as follows. Suppose p is re-executed over a new input I', and produces output O'. Then we check to see if S_t is a provenance in I' and if $\hat{p}(S_t)$ is M, as recorded by p. If answers to both questions are "yes", we remove from O' all tuples in M, and then add to O' all tuples in M'. Otherwise, this update is not applicable to the new output O', and thus we do nothing.

We now discuss how to handle other types of user updates. Suppose a user modifies a tuple t in the output O into t', then we can interpret, save, and incorporate this modification update in

a similar fashion. Suppose a user inserts a tuple t, then we ask the user for the provenance of t. Given the provenance, we can again interpret, save, and incorporate the insertion in the same manner. If the user does not give the provenance of t, then we create a special provenance S^*, and use it as the provenance of t. We assume that S^* appears in any input set I.

The New hlog Semantics: We are now ready to reconsider hlog semantics. In this new semantics, we redefine the notion of a user feedback transaction. Here, given an update (S_t, M, M') on an output tuple t of an operator, a transaction T first incorporates the update (see the preceding two paragraphs), and then propagates it up the execution graph, exactly as in Section 5.2.3.1.

However, before propagating the update up the graph, T saves the update (S_t, M, M') as a tuple for operator p so that if a subsequent transaction T' re-executes p, T' can incorporate this update. Over time, many updates may be saved for p, and they should be saved in their arrival order. When transaction T' incorporates these updates, it should incorporate them in that order.

As for transaction T itself, whenever it re-executes an operator q (that depends on p in the execution graph), T checks to see if q has any saved updates, and then incorporates these updates in their arrival order.

Again, we assume that each user transaction T will x-lock the whole execution graph at the start (of its execution) and unlock the graph after the finish.

Let $\mathcal{T} = \{T_1, T_2, \ldots, T_n\}$ be a set of user transactions, as defined above, during a time period $[x, y]$. It is easy to see that the above graph-locking algorithm guarantees a serial execution of the transactions. Hence, at the end of the time period, the final output of graph G (i.e., the output of the root operation) incorporates all user updates encoded in $\mathcal{T}$, in some serial order. Here, the notion of incorporating user updates is more expressive than that in Section 5.2.3.1, in the sense that a previous user update is clobbered only if there exists a new user update on the same tuple with the same provenance.

In the rest of the chapter, we will use the above conceptual algorithm to express the semantics of hlog.

PO

tid	rid
1	1
2	2
3	2

PR

rid	sid1	sid2
1	1	1
2	1	2

PS_1

sid	tid
1	1
1	3

PS_2

sid	tid
1	4
2	5

Figure 5.3: Example provenance tables for a case where operator p takes as input two tables and produces as output a table with three tuples.

5.2.4 Extending hlog to Handle II

So far we have introduced hlog as a language for writing IE programs. Since real-world applications often involve II activities, such as schema matching and de-duplication, developer U also needs a language to write II programs. Furthermore, since II is often semantics-based, and automatic II techniques are error prone, U also wants to leverage user feedback to improve the quality of II results.

Consequently, we have extended hlog to support II operations, in the same way that we support IE operations. Specifically, to create an II operator p, developer U first models it as a p-predicate or a p-function. Then U implements a procedure g (e.g., in Perl) that carries out the II activity, and associates p with g. To enable users to provide feedback on p's results, U writes a user feedback rule in a way similar to those written for IE predicates.

5.3 Executing hlog Programs

We now describe a baseline solution to execute hlog programs, according to the user update semantics discussed in Section 5.2.3.2.

Let P be an hlog program. In Section 5.2.3.2, we already discuss a high-level algorithm to execute P, including the initial execution and the subsequent loop of waiting for and processing user feedback. In this section, we focus on two most difficult steps of this algorithm: how to store provenance and how to exploit it to incorporate user feedback. The remaining steps are relatively straightforward.

5.3.1 Storing Provenance Data

Let p be an operator that takes as input two tables I_1 and I_2 and produces as output a table O. Recall from Section 5.2.3.2 that the provenance R of each tuple $t \in O$ is then a pair (S_1, S_2), where $S_1 \subseteq I_1$ and $S_2 \subseteq I_2$. That is, p uses the tuples in S_1 and S_2 to produce tuple t. Our goal is to store the provenances of all tuples in O.

Assume that all tuples in I_1, I_2, and O come with unique IDs (it is relatively easy to modify p to do so). Then we can store the above provenances in four tables, as illustrated in Figure 5.3. Table PO stores for each tuple in O the ID of its provenance R. Thus the first row of PO states that output tuple with tid=1 in O has a provenance with rid=1.

Table PR then stores for each provenance the IDs of its component sets, one for each input table. The first row of PR states that provenance with rid=1 consists of sets with $sid1$=1 and $sid2$=1. Table PS_1 then stores for each component set all of its component tuples. The first row of PS_1 for example states that the set with $sid1$=1 consists of the two tuples with tids 1 and 3 in the input I_1.

In the case that p takes a single input table, or more than two input tables, we can store its provenances in a similar fashion. For each operator p whose output is subject to user feedback, we require p to output provenance tables as described above, after each execution. Whenever a user updates the output of p, the provenance tables must also be updated, to reflect the user update. Such updating is relatively straightforward, and we will not describe it further, for space reasons.

5.3.2 Storing and Incorporating User Feedback

Consider an operator p that takes as input a single table. We now use p to describe how we store and incorporate user feedback. (The algorithm below generalizes in a straightforward fashion to the case of multiple input tables.)

Suppose when executed for the first time, p takes input I_1 and produces output O_1, as well as provenance tables PO_1, PR_1, and PS_1, as illustrated in the left part of Figure 5.4.

Now suppose a user updates a tuple $t \in O_1$, with the provenance rid_1. Recall from Section 5.2.3.2 that conceptually we should store a user update as (S_t, M, M'), specifying that when the

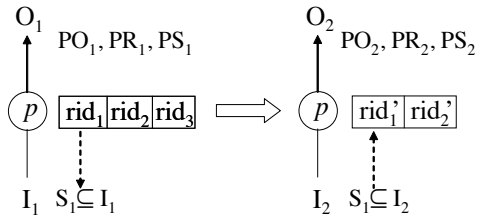


Figure 5.4: An example of incorporating user feedback.

value of $\hat{p}(S_t)$ is M' (i.e., p produces M' from S_t), we should update M into M'. Here, S_t is represented by rid_1, and M' is captured inside O_1, that is, M' consists of exactly those tuples in O_1 that have provenance rid_1. If the value of $\hat{p}(S_t)$ is always the same (i.e., $\hat{p}(S_t) \equiv M$) regardless of other input tuples, then as long as we store rid_1 and O_1, we have effectively stored (S_t, M, M'). In this case, we can just store this update as rid_1 (after we have updated output table O_1 and the provenance tables appropriately). Most IE/II operators have the property that for any provenance S_t, $\hat{p}(S_t)$ is constant. In the other case, where $\hat{p}(S_t)$ is not constant, we can store the update as (rid_1, M, u), where u is the update operation on M (i.e., $M' = u(M)$). In the following, we assume $\hat{p}(S_t)$ is constant.

Suppose that two more user updates come in on the output of p, and that we further store these updates as rid_2 and rid_3 (see Figure 5.4). In general, then, the set of user updates stored at p is a set of provenance IDs (together with the latest output table of p). These provenance IDs are unique. That is, if we have two updates with the same provenance, then the latter will override the former, and so we only need to store each provenance ID once.

Now suppose a transaction T re-executes p with input I_2, and produces output O_2, together with provenance tables PO_2, PR_2, and PS_2 (see the right part of Figure 5.4). Transaction T must then incorporate user updates rid_1, rid_2, and rid_3 into the output of p.

Consider the first update rid_1 (we can process these updates in any order; the end results will be the same). To incorporate this update, we first use the provenance tables PR_1 and PS_1 to find the provenance $S_1 \subseteq I_1$ (see Figure 5.4). Next, we check to see if S_1 is also a provenance in I_2. If so, then we say that update rid_1 is applicable to I_2.

In this case, suppose the provenance ID of S_1 in I_2 is rid'_1 (which can be different from rid_1, see Figure 5.4). Then to incorporate update rid_1, we remove from O_2 all tuples with provenance rid'_1, then copy into O_2 all tuples of O_1 with provenance rid_1. Note that if update rid_1 conceptually specifies that $p(S_t)$ must be M' (as discussed earlier), then copying from O_1 to O_2 in effect sets the value of $p(S_t)$ in O_2 to be M'.

If S_1 is not a provenance in I_2, then we say that the update rid_1 is not applicable to I_2, and we ignore this update. We proceed in a similar fashion with updates rid_2 and rid_3.

After we have processed all of these updates, we revise the set of updates stored at operator p. To do so, we keep all updates that are applicable, but revise their rids, and drop all inapplicable updates. For example, since update rid_1 is applicable, we keep it, and revise its rid to rid'_1 (since we now store I_2 and O_2, no longer I_1 and O_1, where rid is a valid rid). Suppose update rid_2 is applicable, then we do the same, and keep the new rid rid'_2. Suppose update rid_3 is not applicable, then we will drop it from the set of updates maintained at operator p (see Figure 5.4). In theory, we can keep the inapplicable updates around, in case later they become applicable. Doing so, however, would require us to store extra input and output tuples (i.e., the S_t and M') for each inapplicable update. This may take up a significant amount of space over time. Hence, for now we choose not to keep the inapplicable updates around.

We have thus described an algorithm to process user updates. Only one minor issue remains. Earlier we state that given a provenance S_1 in I_1, we must check to see if it is also a provenance in I_2. Checking this by comparing the contents of the tuples is often expensive. We can speed up this checking using a variety of methods, including incremental ID maintenance, which we will discuss in Section 5.4.2.

5.4 Optimizing hlog Execution

We now describe several optimization techniques to speed up program execution. First we describe how to incrementally execute an IE/II operator. Then we describe two concurrency control methods that exploit the graph structure of an IE/II program to achieve higher degrees of concurrency than the whole-graph-locking method in Section 5.2.3.

5.4.1 Incremental Execution

In the baseline solution, we execute an operator p from scratch each time. This is inefficient when only a small amount of p's input has been changed, and most of p's output remains the same. In this section, we describe how to execute an IE/II operator efficiently by incrementally updating its output. The basic idea is to exploit certain properties of the operator with respect to incremental update.

Incrementally updating the output of an operator after its input has been changed is not a new problem. In the relational setting, view maintenance [18, 52] considers a similar problem, where we would like to incrementally update a materialized view after its input relations have been changed. As one solution, we can use the distributive property [18] of the basic relational operators (i.e., σ, π and $\bowtie$) to derive incremental updates to the view.

If we view an IE/II operator p as a view definition, then the input of p is the set of base relations in the view definition, and the output of p is the materialized view. However, unlike relational operators whose semantics are well-understood, the operator p in general is a blackbox, and the distributive property may not hold on p.

To incrementally execute an IE/II operator, we extend the ideas from relational view maintenance. Instead of using a single property to describe all operators, we introduce a set of fairly general properties. Although an operator can be a blackbox, we can make it less "black" by allowing the developer to specify which properties hold for the operator. Given these properties, we can then update the output of the operator incrementally.

In the following, we first present five such properties. Then we provide an algorithm that exploits these properties to construct incremental versions of an operator automatically.

Incremental-Update Properties: Let p be an operator which takes n tables $I_1, \cdots, I_n$ as input, and outputs table O. Each property below captures some incremental relationship between an input table $I_i \in \{I_1, \cdots, I_n\}$ and output O. For conciseness, we use $R_i(S)$ to denote $I_1, \cdots, I_n$ with I_i replaced by S. That is, $R_i(S) = I_1, \cdots, I_{i-1}, S, I_{i+1}, \cdots, I_n$.

Definition 5.6 (Closed-Form Insertion) *An operator p is closed-form insertable w.r.t. an input table I_i if and only if there exists a function f such that for any set ΔI of tuples to be inserted into I_i, the condition $p(R_i(I_i \cup \Delta I)) = f(R_i(\Delta I), O)$ holds.*

Definition 5.7 (Closed-Form Deletion) *An operator p is closed-form deletable w.r.t. an input table I_i if and only if there exists a function f such that for any set ΔI of tuples to be deleted from I_i, the condition $p(R_i(I_i - \Delta I)) = f(R_i(\Delta I), O)$ holds.*

The two closed-form properties state that instead of executing p over the updated I_i (together with other input tables) from scratch, we can execute the function f that examines ΔI, which is often much smaller than I_i, to compute the new output. The function f may invoke p on $R_i(\Delta I)$. For example, consider the *crawl* operator in Figure 5.1.b, which crawls the set of data sources in table *dataSources* to find Web pages. If we add new data source tuples into table *dataSources*, we can obtain the new output by executing the function f that first crawls the new sources and then inserts the crawled Web pages into the current output.

To define the next property, we first define the notion of partitioning function. We say a function f is a *partitioning function w.r.t. a table I* if and only if f partitions I into k disjoint subsets $\{S_1, \cdots, S_k\}$, where $k > 0$, and $I = \bigcup_{i=1}^{k} S_i$. Denote the application of f to I as $f(I) = \{S_1, \cdots, S_k\}$.

Definition 5.8 (Input Partitionability) *An operator p is input partitionable w.r.t. an input table I_i if and only if there exists a partitioning function f on I_i such that*

- $p(I_1, \cdots, I_n) = \bigcup_{S_k \in f(I_i)} p(R_i(S_k))$, *and*
- $f(I_i)$ *is a non-trivial partitioning, in that at least one subset S_j, $j \in [1, k]$, is a proper subset of I_i.*

Intuitively, a partitioning function f partitions an input table into disjoint sub-tables. The input partitionability property implies that the output of an operator on one sub-table is independent of those of the other sub-tables. That is, if the input table is changed, we can update the output by

first identifying all the changed sub-tables in the input, applying the operator to them, and then combining the outputs of these sub-tables with the outputs of those unchanged sub-tables. For example, consider the operator *extractTitle* in Rule R_2 of Figure 5.1.b, which extracts titles from Web pages. A simple partitioning function on its input table *webPages* is to partition the table tuple by tuple. As a result, the output of the operator over the entire input table is the union of the titles extracted from each page. Note that *extractTitle* also has the two closed-form properties above.

As another example, consider a simple IE/II program that discovers people entities from a set of Web pages. First, the program takes each seed name (e.g., "David Smith") from a dictionary, and generates name variants (e.g., "D. Smith" and "Smith, D."). It then finds the mentions of these variants in the Web pages. After that, it executes an II operator *getPeopleEntities* that groups the obtained mentions by their seed names, and then outputs an entity for each group. The operator exhibits the input partitionability property where the partitioning function partitions the input mentions by their seed names. If new mentions are inserted into the input of the operator, we can incrementally update its output by executing the operator over the partitions that the new mentions belong to, and then unioning the outputs with those of the other partitions. However, unlike in the previous example, this operator does not have the closed-form properties.

Definition 5.9 (Partition Correlation) *An operator p is partition correlated w.r.t. an input table I_i if and only if there exists a function f such that for an arbitrary partition $\langle S_1, S_2 \rangle$ of I_i where $I_i = S_1 \cup S_2$, the condition $p(I_1, \cdots, I_n) = p(R_i(S_1)) \cup p(R_i(S_2)) \cup f(R_i(S_1), R_i(S_2))$ holds.*

Unlike the input partitionability property where the output of an operator comprises the output of the operator over each input partition, the partition correlation property captures the cases where the output also contains results obtained from both partitions. Take a data matching operator p for example. Suppose that p takes a single table as input. For each pair of input tuples, p outputs their tuple IDs, together with a score measuring the similarity of the two tuples. Suppose a few more tuples are now inserted into the input table. For certain types of data matching operator p, we can incrementally update the output as follows. Let the old input table be partition S_1 and the set of inserted tuples be S_2. First, we execute p over S_2 to compute $p(R_1(S_2)) = p(S_2)$. Next, we apply

a function f to S_1 and S_2. The function computes the similarity score of each pair of tuples, one from each partition. We then union $p(S_1)$, which is the old output O, with $p(S_2)$ and $f(S_1, S_2)$ to get the new output.

Definition 5.10 (Attribute Independence) *Let A be a proper subset of attributes in an input table I_i, and $\bar{A}$ be the rest of the attributes in I_i. An operator p is independent of $\bar{A}$ in I_i if and only if given any two instances S_1 and S_2 of I_i, $\pi_A(S_1) = \pi_A(S_2) \Rightarrow p(R_i(S_1)) = p(R_i(S_2))$.*

Some operators evaluate only a subset of attribute values of an input tuple. In this case, changes to the unused attributes have no effect on the output of such operators. The attribute independence property captures this case.

The incremental properties above are operational. Each property implies its own way of incrementally updating the output of an operator. Thus, to incrementally execute an operator p, we need to know which properties apply to p and for each property, the specific function f that realizes it (e.g., the function f that satisfies the condition $p(R_i(I_i \cup \Delta I)) = f(R_i(\Delta I), O)$ in Definition 5.6). Given these, we can construct incremental versions of p accordingly.

Property Specification: To allow the developer to specify the incremental-update properties of an operator p, we extend hlog by adding a language construct "$p(I_i)$:{(*type,f*)}". The construct lists a set of properties that p has with respect to its input table I_i. Each property is specified by a pair $(type, f)$, where $type$ is the type name of the property, and f is the function that the developer needs to provide to instantiate the property (see Definitions 5.6-5.10). Here, $type$ can take values from $\{ci, cd, ip, pc, ai\}$. The values in the set stand for closed-form insertion (ci), closed-form deletion (cd), input partitionability (ip), partition correlation (pc), and attribute independence (ai). The function f has different forms depending on the type of the property. For example, if *type* $= ci$, then f takes as input (1) a set of tuples inserted into an input table I_i, (2) all the other input tables, and (3) the old output table, and produces a new output table. If *type* $= ai$, then f is the function that projects out the set of attributes that are irrelevant to the operation of p (See Definition 5.10).

```
# f1: crawls pages from the new data sources, and inserts them into webPages
# f2: deletes from webPages the pages crawled from the deleted data sources
P1:     crawl(dataSources) : {(ci, f1), (cd, f2)}
# f3: partitions webPages tuple by tuple
P2:     extractTitle(webPages) : {(ip, f3)}
P3:     extractAbstract(webPages) : {(ip, f3)}
```

Figure 5.5: An example of incremental-update property specification.

Figure 5.5 shows an example of property specifications for predicates *crawl*, *extractTitle*, and *extractAbstract* in Figure 5.1.b. In the figure, P_1 specifies two closed-form properties for predicate $crawl$. (The two properties must be specified in pairs. That is, if the developer U specifies the closed-form insertion property for an operator p, U must also specify the closed-form deletion property for p.) The closed-form insertion property, for example, is obtained from the observation that when new tuples are added to $dataSources$, we can update table $webPages$ by first crawling pages from these new sources and then adding them to $webPages$. Furthermore, P_2 and P_3 specify input partitionability properties for predicates *extractTitle* and *extractAbstract*. They share the same partitioning function f_3 because table $webPages$ can be partitioned in the same way (i.e., tuple by tuple) to satisfy the property for both predicates. Note that the developer does not need to specify all the applicable properties for an operator. For example, in addition to the closed-form properties, the operator *crawl* also has the input partitionability and attribute independence properties. Instead of specifying all the properties, the developer can specify those that he or she deems cost-efficient to execute.

Also note that each specification alone enables us to incrementally execute an operator p when among all the input tables of p, only the one given in specification has been changed. Therefore, to enable incremental update for any changes to the input of p, the developer needs to specify properties for each input table of p.

Algorithm: Once the developer has specified incremental properties for an operator p, we can create incremental versions of p accordingly. Figure 5.6 gives the pseudo-code of the algorithm that we currently use to realize incremental execution of p for each property specification. Briefly, the algorithm takes as input the specification, the old input and output tables, and the new input

table I_i'. To compute the new output, the algorithm first computes I^+ and I^-, the set of tuples inserted into I_i and the set of tuples deleted from I_i. It then executes the function f given in the specification to update the old output.

Algorithm: Generic Incremental Update Algorithm

Input: Spec: $p(I_i) \leftarrow (type, f)$
$I_1, \cdots, I_n$ – the old input tables of p
I_i' – the new value of the ith input table
O – the old output table of p

Output: O' – the new output table

Process:

1. $O' = \emptyset$; $I^+ = I_i' - I_i$; $I^- = I_i - I_i'$;
2. **if** $(type = ip)$ **then**
3. $\quad \{S_{1:u}\} = f(I_i)$; $\{S'_{1:v}\} = f(I_i')$;
4. $\quad$ **for each** $P \in \{S_{1:u}\} \cap \{S'_{1:v}\}$
5. $\quad\quad O' = O' \cup \{$output of partition P in $O\}$;
6. $\quad$ **for each** $P \in \{S'_{1:v}\} - \{S_{1:u}\}$
7. $\quad\quad O' = O' \cup p(R_i(P))$;
8. **else if** $(type = ci)$ **then**
9. $\quad O' = f(R_i(I^+), O)$;
10. **else if** $(type = cd)$ **then**
11. $\quad O' = f(R_i(I^-), O)$;
12. **else if** $(type = ai)$ **then**
13. $\quad$ **if** $(f(I^+) \neq f(I^-))$ **then**
14. $\quad\quad O' = p(R_i(I_i'))$;
15. $\quad$ **else** $O' = O$;
16. **else if** $(type = pc)$ **then**
17. $\quad U = p(R_i(I^-))$;
18. $\quad O' = O - U - f(R_i(I^-), R_i(I_i - I^-))$;
19. $\quad U = p(R_i(I^+))$;
20. $\quad O' = O' + U + f(R_i(I^+), R_i(I_i - I^-))$;
21. **return** O';

Figure 5.6: Generic incremental update algorithm.

5.4.2 Incremental ID Maintenance

Consider a single-input operator p. Let I_1 and O_1 denote its input and output from its previous execution, and I_2 and O_2 denote its new input and output. Recall from Section 5.3.2 that to decide whether a user update F to O_1 is applicable to O_2, we check whether the provenance $R \subseteq I_1$ of F is also a provenance in I_2. One way to check this is to compare the contents of tuples in R with those in each provenance of I_2. But this is often expensive. If two tuples $t \in I_1$ and $t' \in I_2$ have the same ID if and only if they have the same content, then we only need to compare R with each provenance in I_2 by their tuple IDs, which is much more efficient. Call such a condition *ID consistency*. To ensure this condition, clearly we need to reconcile the tuple IDs whenever the input table I_1 of p has been re-computed to be another input table I_2, by some operator q (below p in the execution graph).

By using incremental update, we can save a lot of ID reconciliation effort. This is because when we incrementally execute q, the unchanged tuples are retained in the output table of q (which is the input table I_2 discussed above for p). Thus for these tuples, their IDs are consistent (i.e., the same), and we only need to reconcile IDs for the newly generated output tuples. Specifically, for each new tuple t, we check whether t is present in the old output. If so, we set t's ID to be its ID in the old output. Otherwise, we assign t a new ID, which is guaranteed to be different from that of any other tuple. Therefore, by leveraging incremental execution, we can maintain ID consistency incrementally.

The above notion of ID consistency, however, requires the tables to have set semantics, that is, they cannot contain duplicates. This requirement may not work for certain IE/II operators in practice. To address this problem, we can employ a more relaxed notion of ID consistency. We say that tuple IDs are consistent across two tables I_1 and I_2 if whenever two tuples $t_1 \in I_1$ and $t_2 \in I_2$ have the same ID, they are duplicates. This consistency notion is more relaxed in that two tuples with different IDs can still be duplicates.

Using this relaxed ID consistency notion, when checking whether provenance $R \subseteq I_1$ is also a provenance of I_2, for each tuple $t \in R$, we check for its presence in I_2 by first checking if its ID

appears in I_2. Only if we do not find its ID in I_2 would we resort to comparing its content against the content of tuples in I_2.

5.4.3 Improved Concurrency Control

The simple graph-locking policy in Section 5.2.3 requires a transaction to exclusively lock the entire execution graph before it starts. Since a transaction only executes one operator at any time, exclusively locking the whole graph excludes other transactions from executing other operators. By exploiting the fact that an execution graph is a DAG, we can design more efficient concurrency control solutions. In what follows we describe two such methods: table locking and operator skipping.

5.4.3.1 Table Locking

Recall that a user feedback transaction starts by updating the table whose view the user has edited. Refer to this table as the starting table. The transaction then re-executes operators that depend on the starting table, and so on. It terminates after it has re-executed the root-node operator.

The table-locking policy does not require a transaction T to lock the entire execution graph G. Instead, it requires T to acquire locks on individual tables before it updates its starting table or executes an operator. Specifically:

- Before updating the starting table S, T requests an exclusive lock on S. Let p be the operator[2] that produces S. T also requests exclusive locks on the provenance tables and the update table (i.e., the table stores all user updates to S) of p, in an all or nothing fashion. T releases these locks when the update is completed.

- Before executing an operator p, T requests share locks on all the input tables of p in an all or nothing fashion.

[2]If S is an extensible predicate, then we pretend that it is the output of a dummy operator (see Section 5.2.3.2). In this case, T x-locks S only.

- Before writing the output of p, T acquires exclusive locks on p's output, provenance, and update tables, in an all or nothing fashion. After writing the output, T releases these locks, together with the share locks on the input tables of p.

Compared to the two-phase locking of the execution graph, the table-locking policy allows a transaction to interleave its lock acquisition and releasing activities. To execute an operator p, a transaction locks only the tables related to p. Furthermore, it releases these locks as soon as the output of p is written. Therefore, the policy allows other transactions to execute an operator q as long as there is no input dependency between p and q (i.e., p and q are not connected in the execution graph). The following theorems state that the table-locking policy guarantees the consistency of the system and that concurrent execution is deadlock-free.

Theorem 5.11 (Consistency) *Given a set of transactions $\{T_1, T_2, \cdots, T_n\}$, if all transactions follow the table-locking policy, then the system remains consistent after executing these transactions if it is consistent before.*

Proof 1 Let G be the execution graph of the program, and m be the number of nodes in G. We prove by induction on m that the table-locking policy guarantees that the concurrent execution of these transactions is equivalent to some serial execution. Here, we generalize the execution graph G so that it may have multiple root nodes (i.e., nodes without outgoing edges in G).

Basis (m=1): If G has only one node, then by the definition of execution graph, the table O associated with the node is both the input and the output of the program. As a result, each transaction in $\mathcal{T}$ updates O and then commits. Since the locking policy allows only one transaction to update O at any time, the system executes these transactions sequentially. This gives us a serial execution order.

Induction Step: Assume that serializability holds for $m = k$. We now show that it also holds for $m = k + 1$. Let p be an operator in G that does not have an outgoing edge. That is, no operator takes the output O of p as input. Consider the subgraph $G' = G - \{p\}$, which has k nodes. Let $\mathcal{T}_p \subseteq \mathcal{T}$ be the set of transactions that execute p. ($\mathcal{T}$-$\mathcal{T}_p$ are the transactions that updates O only.)

We modify each transaction $T \in \mathcal{T}_p$ into T' by removing from T its actions on executing p and updating p's output. The result is a set of transactions $\mathcal{T}'_p$ on G'. By induction, we can serialize these transactions in some order s. Now we show that the same order holds for $\mathcal{T}_p$ on G. This is because once all the transactions in $\mathcal{T}_p$ finish executing G', the input tables of p will not be changed. Thus executing p by any transaction produces the same output. Finally, we extend s into s' by appending those transactions in $\mathcal{T}$-$\mathcal{T}_p$ in the order of the time they updates O. Recall that each transaction in $\mathcal{T}$-$\mathcal{T}_p$ is conceptually an update (S_t, M, M') on the output O. Thus executing these transactions at the end of s' incorporates user feedback captured by these transactions into O. Therefore, the result of executing s' is the same as the result of the concurrent execution by the system.

Theorem 5.12 (Deadlock Freedom) *The table-locking policy cannot produce a deadlock.*

Proof 2 We prove deadlock freedom by contradiction. Suppose that the table-locking policy causes a deadlock among n transactions $\{T_1, \cdots, T_n\}$. Then there must be a cycle in the wait-for graph of these transactions, where each node in the graph is a transaction, and there is a directed edge from T_i to T_j if T_i is waiting for some lock held by T_j. Ignore the provenance tables and update tables for simplicity. Consider an edge from T_i to T_j. By the table-locking policy, T_i must be waiting for an x-lock on the output table O of some operator p, while T_j is holding an s-lock on the table. In addition, T_j must be waiting for an x-lock on the output table of another operator q, and O is an input table of q. This implies that p precedes q in the execution graph. Using the same reasoning, each edge in the wait-for graph implies a preceding relation between two operators in the execution graph. As a result, a cycle in the wait-for graph implies a cycle in the execution graph. This contradicts with the fact that an execution graph is acyclic.

5.4.3.2 Operator Skipping

Consider a set of transactions $\{T_1, \cdots, T_n\}$ running concurrently on an execution graph G. Consider an operator p in G. Let T be the last transaction that executes p. Recall that a transaction executes operators in G upwards. Thus at the time T executes p, all the input tables of p are at their

final states (i.e., no other transactions will change the value of any input table of p). This suggests that once the input tables of p are at their final states, it is sufficient to have only one transaction execute p to update its output. In other words, transactions other than T can skip executing p, and let T execute p and write the output.

Suppose after T executes p, the output of p is changed. Then by definition, T must execute each operator q which takes p's output as input. Suppose for simplicity that q is a single-input operator. Then T also writes the final value of the output of q. This suggests that a transaction T' ($T' \neq T$) can also skip executing q since the final state of q is set by T. Applying this reasoning recursively, T' can skip executing those operators in G that are reachable from p. If T' can skip all the operators it needs to execute, T' can commit immediately given that T commits eventually.

Based on this idea, we extend the table-locking policy to allow a transaction to skip executing an operator if some other transaction will eventually read the final values of the input of the operator and overwrite the output. We call the extended policy operator skipping.

To implement operator skipping, we maintain a transaction ID list for each operator. When a transaction T starts, it adds its ID to the list of each operator it is going to execute. Before executing an operator p, T checks whether it is the only transaction in p's list. If so, T executes p; otherwise, T skips p. In either case, T removes its ID from p's list. Operator skipping also guarantees consistency and deadlock freedom since it follows the table-locking policy.

5.5 Empirical Evaluation

As a proof of concept, we conducted a preliminary case study by applying our user feedback solution to DBLife [36], a currently deployed IE and II application. Our goal is to evaluate the efficiency of the solution in incorporating user feedback into IE and II programs, focusing specifically on the optimization techniques proposed in Section 5.4. The experiments show that the incremental execution approaches can reduce the program execution time considerably, and that by exploiting the DAG structure of an execution graph, the concurrency control methods, table locking and operator skipping, can significantly improve the system performance in terms of both transaction throughput and response time.

	Incremental Properties				
DBLife Operators	ci	cd	ip	ai	pc
Get Data Pages	✓	✓	✓	✓	✓
Get People Variations	✓	✓	✓		✓
Get Publication Variations	✓	✓	✓		✓
Get Organization Variations	✓	✓	✓		✓
Find People Mentions	✓	✓	✓		✓
Find Publication Mentions	✓	✓	✓		✓
Find Organization Mentions	✓	✓	✓		✓
Find People Entities			✓		
Find Publication Entities			✓		
Find Organization Entities			✓		
Find Related People	✓	✓	✓		✓
Find Authorship	✓	✓			✓
Find Related Organizations	✓	✓	✓		✓

Table 5.1: Incremental properties of DBLife operators.

Application Domain: DBLife [36] is a prototype system that manages the data of the database community using IE and II techniques. Given a set of data sources (e.g., homepages of database researchers and conference websites), DBLife crawls these data sources regularly to obtain data pages, and then applies various IE and II operators to the crawled pages to discover entities (e.g., people and organizations) and relationships between them (e.g, affiliated-with and give-talk). A variety of user services, including browsing and keyword search, are then provided over the obtained entities and relationships.

Methods: To evaluate the effectiveness of the framework, we implemented a modified version of the DBLife system under the framework. The modified DBLife system contains 13 operators, as listed in Table 5.1. These operators, ranging from crawling data sources to extracting mentions, to finding entities and relationships, cover the most essential operators in the entire DBLife workflow. Input to the modified DBLife program consists of four tables, which store data sources, researcher names, organization names, and publication titles, respectively. Output of the program consists of three entity tables and three relationship tables, produced by the last six operators in Table 5.1.

To evaluate the efficiency of the system in incorporating user feedback, we exposed the output of each operator, together with the four input tables, for user feedback. For simplicity, we

used identity views to expose these 17 tables. We built a transaction simulator to generate user feedback transactions on one snapshot of the program data. The simulator generated a user transaction as follows. First, it randomly selected one of the 17 tables. Then it randomly deleted one tenth, inserted one tenth, and modified another one tenth of the tuples in the table to simulate user feedback.

5.5.1 Incremental Execution

Broad Applicability of Incremental Properties: When developing DBLife [36], we did not expect its operators to be executed incrementally. Thus, we did not design them to be incrementally updatable. However, all the DBLife operators surprisingly have at least one incremental property presented in Section 5.4.1, and many of them have several. Table 5.1 lists the incremental properties (checkmarked in the table) of the operators we experimented with. This suggests that these properties may have a broad applicability to many real-world IE and II operators.

Efficiency of Incremental Execution: To evaluate the incremental update and incremental ID maintenance methods in Sections 5.4.1 and 5.4.2, we developed three versions of DBLife. They are (1) the basic version which executes each operator from scratch, (2) the incremental update version which executes the operators incrementally (the properties used for each operator are underlined in Table 5.1), and (3) the incremental update with ID maintenance version which leverages incremental update to maintain ID consistency.

In the experiment, we first initialized each version of the system by running the DBLife program over a given set of data sources. We varied the size of the set so that it gave 100-600 pages, at an interval of 100 pages. For each version, we then simulated 170 user feedback transactions, 10 on each table. Next, we executed these transactions separately. After each transaction completed, we restored the system to its initial state.

Figure 5.7 compares the average transaction execution time in the three versions as the size of data sources varies. As we expected, the basic version has the longest average execution time. As the size of data sources increases, the gap between the basic version and the incremental versions increases. Note that in the experiment, each transaction was simulated to update about one-third

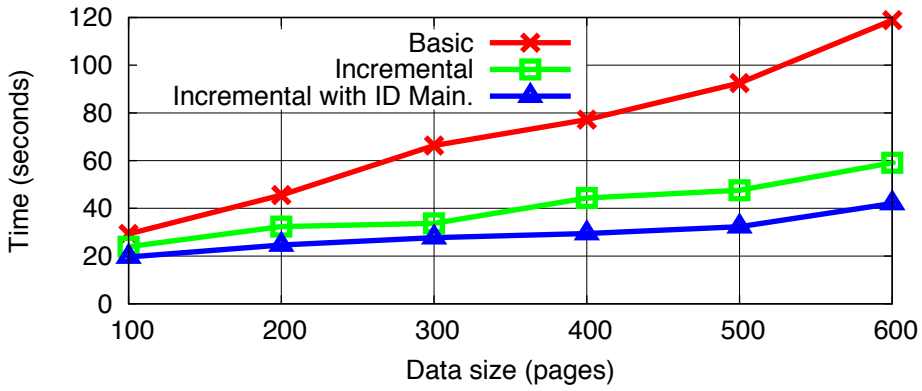


Figure 5.7: Transaction runtime in different DBLife versions.

of the tuples in a table. Thus if a transaction updates only a few tuples in a table as a user is likely to do, the gain of incremental update will be even more significant. Figure 5.7 also suggests that leveraging incremental update to maintain ID consistency is a good strategy to reduce the execution time.

5.5.2 Concurrency Control

In the next step, we evaluated the two concurrency control policies proposed in Sections 5.4.3.1 and 5.4.3.2. We used graph-locking policy which requires a transaction to exclusively lock the entire execution graph before execution as the baseline. In the experiment, we fixed the size of data sources to 500 pages, and used the incremental update with ID maintenance approach to execute the operators. Again, we simulated 170 user feedback transactions, 10 for updating each table. These transactions were then started one after another in a random order, at an interval of one second. The same order was used for comparing different policies in the experiment. We recorded the starting time and the commit time of each transaction, and also measured the space consumption of the system at different times.

Figure 5.8 shows the number of transactions completed at any given time. The total time to complete 170 transactions for the graph-locking (GL), table-locking (TL) and operator-skipping

(OS) policies is 7605, 5965 and 641 seconds, respectively. It is clear that TL and OS outperform GL since they allow transactions to be executed concurrently. However, the improvement of TL over GL in terms of throughput is not as significant as we expected. This is because many of the operators are long-running and CPU-bounded. Thus although TL allows multiple transactions to execute different operators at the same time, these transactions have to compete for CPU intensively.

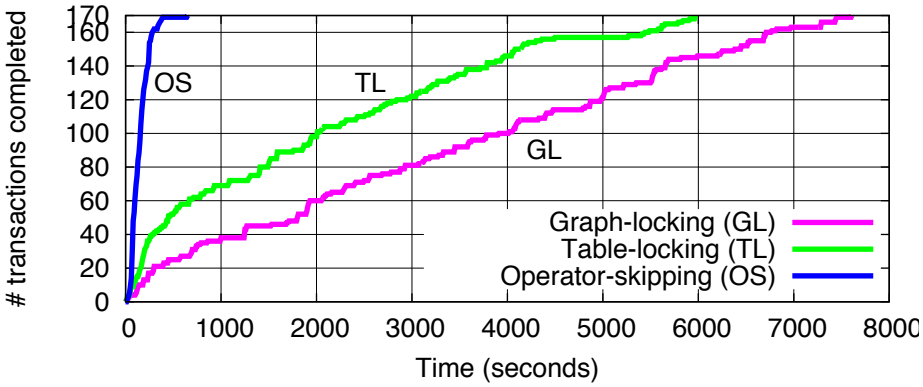


Figure 5.8: Comparison of transaction throughput.

Figure 5.8 also demonstrates that OS outperforms the other two policies significantly in terms of transaction throughput in any period of time. This is not surprising because each transaction by definition must propagate updates on its starting table all the way to the end tables in the program. Thus two transactions may easily overlap in terms of the operators to execute. This is especially true for those operators that produce the end tables. In particular, big transactions (e.g., those updating the input tables) usually subsume small transactions (e.g., those updating the end tables). Therefore, a transaction is likely to skip executing an operator p if some other transaction will eventually overwrite the output of p.

Table 5.9 lists the response time of the transactions under different policies. The average response time of table-locking is nearly one half of that of graph-locking, and the average response time of operator-skipping is only 1/40 of that of table-locking. Comparing the maximum response time of the three policies, we also observe much difference. Operator-skipping outperforms the

other two significantly because small transactions tend to commit immediately when there are active transactions in the system that subsume them. As a result, fewer transactions compete for CPU or IO resources.

	min	max	average
Graph-locking	0s	7,584s	3,203s
Table-locking	1s	5,485s	1,841s
Operator-skipping	0s	457s	43s

Figure 5.9: Comparison of transaction response time.

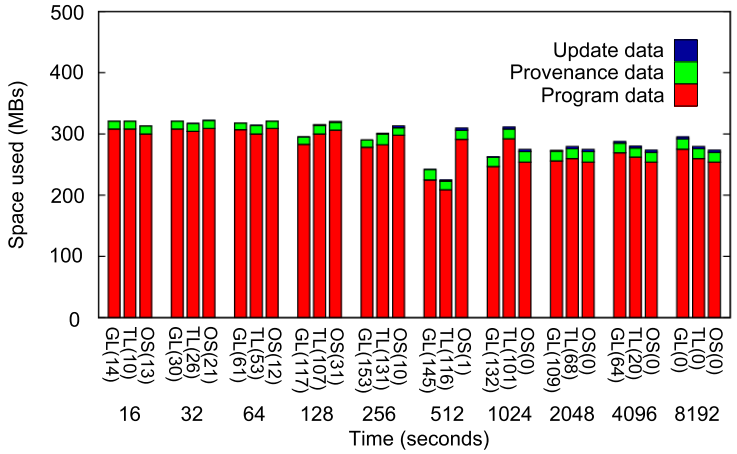


Figure 5.10: Space consumption under different concurrency control policies. Numbers in brackets give the number of active transactions at a certain time.

Figure 5.10 shows the amount of space used by DBLife at different time points during concurrent execution. As we can see, the total amount of space consumed remained at the same level, regardless of the number of active transactions in the system. Variations were mostly caused by the updates from the transactions.

By decomposing the total space consumption in Figure 5.10, we see that the extra amount of space incurred to store the provenance data and the user update data is reasonable. On average, provenance data only takes about 5.7% of the space that the program data needs. Furthermore, the

growing rate of the space consumption is much lower than that of the data sources. This is because intermediate results take up most of the space, and they are independent of the data sources.

5.6 Summaries

Despite recent advances in improving the accuracy and efficiency of IE/II programs, writing these programs remains a difficult problem, largely because automatic IE/II are inherently imprecise, and there is no easy way for human users to provide feedback into such programs. To address this problem, we proposed an end-to-end framework that allows developers to quickly write declarative IE/II programs with the capabilities of incorporating human feedback. In addition to the framework, we also provided optimization techniques to improve the efficiency and concurrency of program execution. Experiments with DBLife demonstrated the utility of the framework.

Our work has raised more research problems than those solved. For example, what are the strength and weakness of different user interfaces in supporting human interactions? How can we extend the framework to support views over multiple tables? Supporting multiple-table views requires us to revisit the semantics of program execution. This is because user updates on such views may potentially update multiple tables in an execution graph, and incorporating updates will be much more complicated. Furthermore, in this work, we assume that users are either reliable or there is a control mechanism that can filter and aggregate unreliable user feedback into reliable feedback. As a next step, we need to investigate issues raised from unreliable user feedback, and develop such mechanisms.

Chapter 6

Building a General Platform

In this chapter, we describe our implementation of the general platform in Figure 2.1, which is shown again in Figure 6.1 below. Along the way, we discuss our design decisions made for the components, and use source code examples for illustration. At the end of the chapter, we provide our case studies on application development using the platform in two real-world applications.

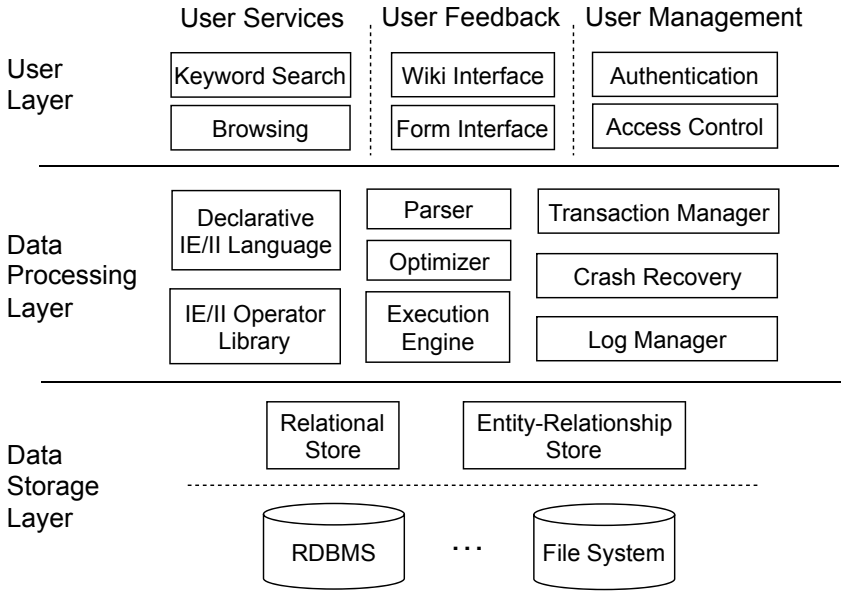


Figure 6.1: High-level architecture of the general platform.

6.1 Building the Data Storage Layer

This layer stores portal data including both structured data (e.g., entities, relationships, and their attributes) and unstructured data (e.g., HTML pages crawled from the Web). As we discussed

in Section 2.2, storage systems that are suitable for different forms and characteristics of the portal data may be used here.

An important lesson we learned from building DBLife [33] is that storing and processing the structured data could significantly benefit from leveraging RDBMS capabilities (e.g., sophisticated query processing and efficient concurrency control). And we believe this also holds for structured Web portal applications in general. In many IE/II applications, however, developers usually choose not to use an RDBMS (at least not today) for data storage and processing. This is mainly because of the overhead of using RDBMS to store and process unstructured data and the difficulty of extending it to support various IE/II operations. Furthermore, as we discussed in Chapter 4, structured Web portal applications create and manipulate entities and relationships, and often need to associate temporal and accounting information with them. Thus a research question we want to answer is how to extend RDBMS to meet these needs so that developers can enjoy both the efficiency and the convenience of RDBMS at the same time. To answer this question, we incorporated and extended an RDBMS in the data storage layer. Among various open-source RDBMS products available, we chose PostgreSQL [73] for its ease of extensibility. Below we describe the extensions we made in detail:

Unstructured data storage: Our focus is on storing and retrieving text data. In dealing with text data, developers often start with laying out a directory structure to store text data, then write programs to import the data into the created directory, where the data may be obtained from local or remote data sources (e.g., files residing on another computer in the local network or Web pages at a website). In subsequent implementation of application logic, developers must then manually translate references to the text data to access paths to the physical files in the directory. To make it easier for developers to perform these tasks, we created user-defined functions (UDFs) in PostgreSQL so that developers can use these UDFs to declare logical groups of text data to store, and to import text data from a variety of data sources.

An example of using the provided UDFs to retrieve and store text data is shown in Figure 6.2. In this example, we crawl a set of websites whose URLs are stored in table *surfaceWebSources*, then store the crawled pages into another table. To do that, we first create a table named *crawledPages*

which stores for each crawled page (1) the id of the web source it is obtained from, (2) the URL of the page, and (3) the content of the page as a text string (see Statement S1 in Figure 6.2). Then we simply write a SQL insert-by-select query (Statement S2) as if we could retrieve the pages by querying a table named *page*. The *page* table is created by a built-in UDF function *crawl*. The *crawl* function takes as input the URL of a website and a crawl depth (i.e., the number of levels into the website we will reach by following the links recursively on the pages) and outputs a table of crawled pages and their URLs. As we can see, instead of first writing a program outside the database to crawl the websites then importing the crawled pages into the database, developers can complete the task by the two SQL statements inside the database using the provided UDF.

```
S1:  CREATE TABLE crawledPages (
                    data_source_id INT,
                    url TEXT,
                    content TEXT);

S2:  INSERT INTO crawledPages
     SELECT src.id as data_source_id, page.url as url, page.content as content
     FROM surfaceWebSources AS src, LATERAL(crawl(src.url, src.crawl_depth)) AS page;
```

Figure 6.2: An example of retrieving unstructured text data from the Web.

The benefits are two-fold. First, developers no longer need to create and maintain the physical storage of text data. The storage system manages this automatically and makes the text data readily available in relational form. Second, by enabling the developers to delegate text storage to the storage system, we can equip the system with advanced text management capabilities from which developers can gain much more benefits. For example, some portal applications need to process text data that may go through revisions and thus change over time. An example of such text data is Web pages that are updated regularly. To answer historical or continuous queries over the evolving Web pages, the application needs to keep track of multiple snapshots of the pages. Storing each snapshot individually by its own, however, can be space-consuming. Since only a small portion of a Web page changes between consecutive snapshots, storing the snapshots incrementally would be much more space-efficient. Thus we extended PostgreSQL with versioned text storage capabilities

so that developers can delegate the system to manage text data that can benefit from incremental text storage.

As an example, suppose we want to regularly crawl a set of talk-event pages, where each page lists the information of research talks that will take place at an organization. Most likely these talk-event pages are updated infrequently (e.g., weekly), and each update only change a small portion of a page (e.g., a newly scheduled talk is included and the oldest talk is excluded). To keep track of the update history of these pages and to store all these snapshots of the pages efficiently, we can write SQL-like statements as those shown in Figure 6.3 below. In the figure, we first create a versioned text table *talkEventPages* which will store all snapshots of crawled event pages. Once we retrieve the latest event pages, we can insert the pages into the *talkEventPages* table using Statement S2. The statement takes the URL of a page, the content of the page, and a threshold parameter (0.5 in the example), and stores the content as the latest version of the page at the given URL. The threshold is used to determine whether to store the complete content of this version or to store only the difference with respect to the latest check point (i.e., the most recent complete version stored). Specifically, we compute the ratio of the size (in bytes) of the difference to the size of the page at the latest checkpoint. If the ratio exceeds the threshold, we store the entire current version and make it a new check point; otherwise, we store only the difference. Finally we can retrieve the latest snapshot of the event pages from the table using Statement S3.

```
S1:  PERFORM create_versioned_text_table('talkEventPages', 'url TEXT, content TEXT');

S2:  PERFORM store_text(talkEventPages, p.url, p.content, 0.5)
     FROM latestTalkEventPages p;

S3:  SELECT *
     FROM retrieve_current_snapshot(talkEventPages) AS p(url TEXT, content TEXT);
```

Figure 6.3: An example of storing and retrieving versioned text data.

Entity Relationship Management: As we discussed earlier, the notions of entities and relationships are prevailing to portal applications. Thus developers often explicitly model objects from the application domain as entities and relationships, and write programs to create and store the entities and relationships. Depending on the data storage system used, developers may have to translate the

mapping between the logical entities and relationships and their physical structures in the storage system, e.g., by implementing serialization and de-serialization methods for each type of entities and relationships. To save the developers from such burdens, we provided a built-in library of operators that developers can use to declare entity and relationship types, create instances, and map the instances into relational form for subsequent processing. An example is shown in Figure 6.4 below. Statement S1 creates an entity type *publication* with four attributes: *title*, *authors*, *year*, and *url*. Suppose we already have a table named *papers*, which has a column for each publication attribute with the same name. To create publication entities, we can execute Statement S2, which creates a publication entity from each paper record in the *papers* table. Finally we can retrieve attribute values of the publication entities using Statement S3.

```
S1: PERFORM create_ent_type('publication', 'title text, authors text, year int, url text');

S2: PERFORM create_entity('publication', ['title', 'author', 'year', 'url'], [title, note, year, url, pdf])
    FROM papers p;

S3: SELECT *
    FROM get_entities('publication', 'entity_id, title') AS (entity_id INT, title TEXT);
```

Figure 6.4: An example of creating and querying entities.

The benefits of using these provided facilities are two-fold. First, developers no longer need to be aware of how the entities and relationships are stored and managed. Second, since the actual scheme for storing the entities and relationships is now transparent to the developers, we can extend and provide additional functionalities into the storage system. For example, we extended PostgreSQL with support for storing and maintaining temporal and accounting information for each stored object. The internal representation of entities and relationships and the implementation followed the design in Chapter 4.

6.2 Building the Data Processing Layer

This layer processes portal data. Essential to this layer are a language for writing IE/II programs and facilities for managing the execution of these programs. In the platform built, we provided a

SQL-like language for writing IE/II programs, as we have already seen in the examples in Figures 6.2 and 6.4. And we extended PostgreSQL with the abilities of parsing and executing IE/II programs inside an RDBMS. In addition, we provided language constructs $BEGIN\_TX\_BLOCK$ and END_TX_BLOCK for declaring the beginning and the end of a transaction. Developers can use these constructs to enclose multiple IE/II steps in a program. The enclosed steps will then be executed as a unit or not at all. Moreover, as an IE/II program is usually long-running, if the system crashes in the middle, developers would like the system to be able to resume execution from where it left off (at the granularity of transaction) after the system recovers from the crash. To achieve this, we extended PostgreSQL with a log manager and a recovery module.

Besides the above, we also provided a built-in library of operators for common IE/II operations. Examples include extracting common HTML elements (e.g., titles, headers, URLs and anchor text) from HTML text, matching entities based on specified attributes, and subsequently merging the matched entities by aggregating their attributes.

6.3 Building the User Layer

A general platform would not be complete without modules that can be easily configured to provide common user services, support user feedback, and enforce access control. Thus we provided these modules as built-in components in the platform. Below we briefly describe our implementation of the modules and their features.

Keyword search: We supported keyword search over both unstructured data and structured data. We adopted Lucene[1], one of the most popular and widely used open source search engine implementations, to implement a generic and best-effort keyword searcher. Developers can simply specify which relational tables, including those representing text collections, from the underlying database to index; the search module automatically builds an index over the data and answers keyword queries using the index. In addition to the generic keyword search component, we also implemented a component for searching over the entities in the portal. This keyword-search-over-entities component indexes attribute values of the entities, and groups search results by entity type.

[1] http://lucene.apache.org/

It provides developers an easy way to expose this semantically rich structured data to users via keyword search. In addition to specifying among all the types of entities which ones to index, developers can also choose which functions to use in measuring the similarity between the search terms and entity attribute values. We provided implementations for two similarity measures [66]: a Jaccard distance measure based on words, and an *n*-gram measure ($n = 2$ or $n = 3$) based on either words or characters.

Browsing: Besides keyword search, browsing is another common way users explore portal data. Thus we implemented a generic browsing module. Since entities and relationships are explicitly modeled by the platform, the browsing module automatically enables navigation among entities via relationships between them.

Entity homepages: Information about an entity includes its attributes and its relationships to other entities. A nice way of presenting such information is to group different pieces of the information (e.g., by the type of the relationships that the entity participates in) and then create a page where each group is presented in a block. An example of such presentation is the researcher homepage DBLife [36] automatically created for Joseph M. Hellerstein, as shown in Figure 6.5. Each block on his homepage describes one facet, e.g., research papers he has published, research topics he is interested to, or conference services he has provided. In the platform, we designed and implemented an entity homepage publishing module which developers can use to easily configure which information to publish on entity homepages and how and where the information should be laid out. We provided four templates for presenting information: a record template (e.g, the profile block at the top-right of Figure 6.5 where each piece of information comes from a field in a record), a list template (e.g., the related-people block beneath the profile block), a record-list composite template (e.g., the publication block in Figure 6.5 where each publication is a record in the list), and a wiki template (e.g., the wiki text in Figure 4.1).

User feedback: To support easy user feedback into portal applications, we implemented a user feedback module using Google Web Toolkit[2]. With the module, developers can simply switch it on or off to enable or disable user feedback. Its implementation followed the design in Chapter 4.

[2]https://developers.google.com/web-toolkit

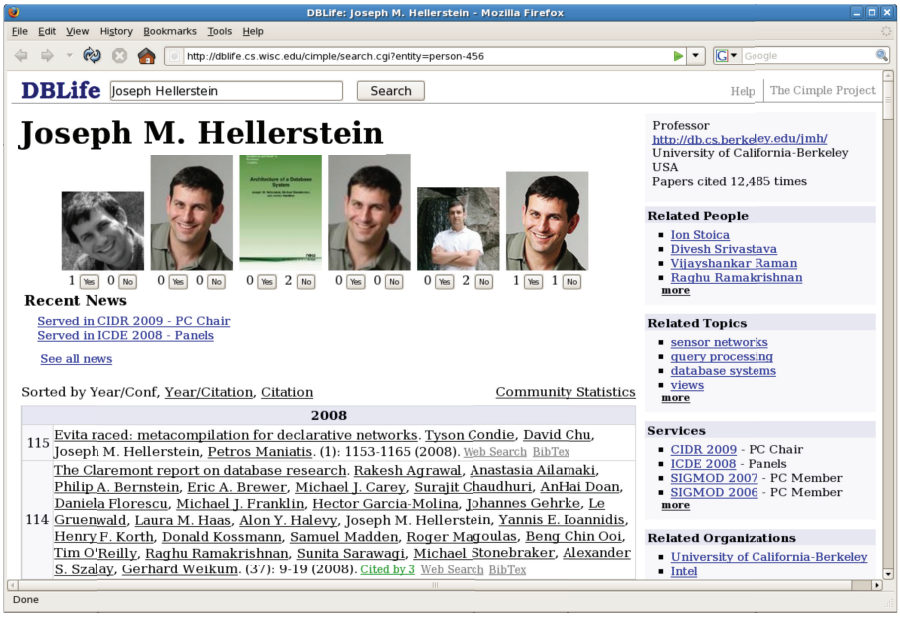


Figure 6.5: An example of entity homepage.

Authentication and access control: In addition to the above user service and feedback modules, a native implementation of authentication and access control modules is provided in the user layer. These modules provide basic user registration, password-based authentication, and page-level access control.

6.4 Building Platform Administration Interface

To ease the administration and monitoring of portal applications, we provided a Web-based administration tool as shown in Figure 6.6 below. It provides developers an easy way to browse portal data and programs, and to make changes, for example, starting/stopping publishing certain blocks of information on entity homepages. An example is shown in Figure 6.7.

6.5 Case Studies

The purpose of providing a general platform is to support developers in performing common tasks in building portal applications. Using the platform, developers can then focus more on developing application logics than carrying out these tasks. We implemented the platform as outlined

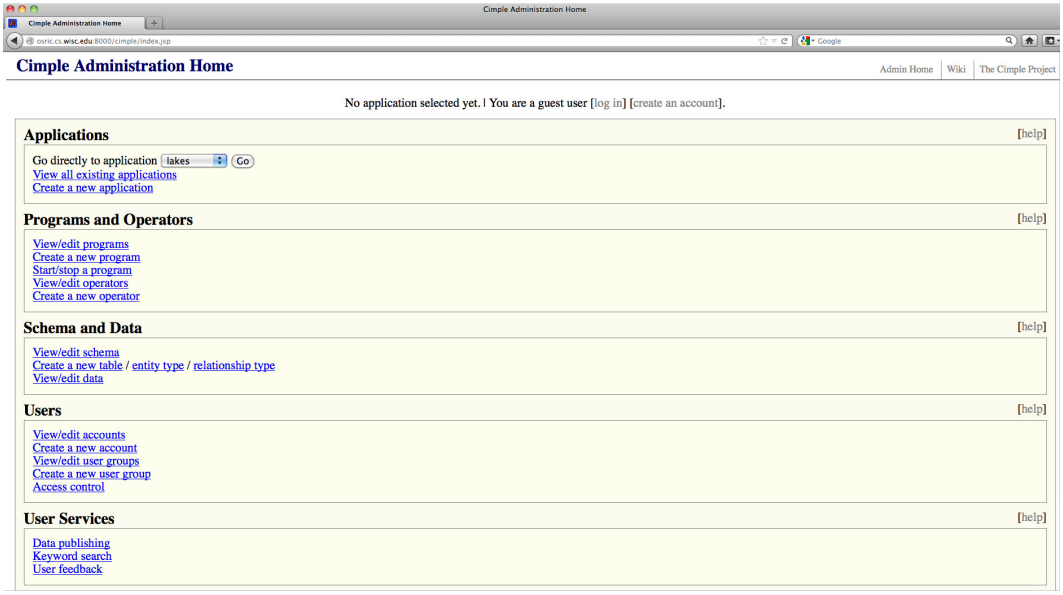


Figure 6.6: Web-based administration tool.

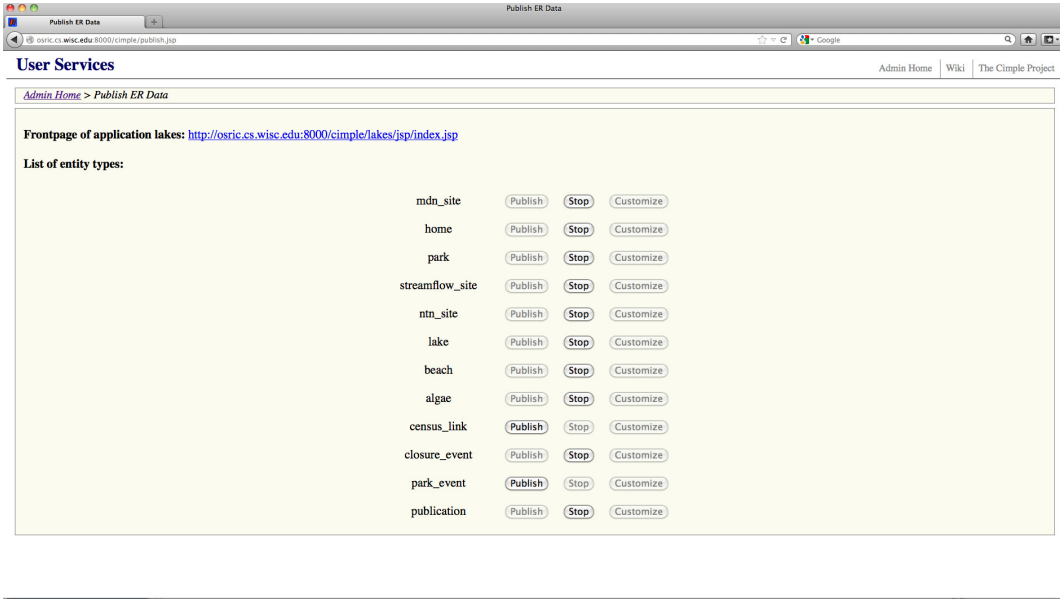


Figure 6.7: An example of publishing entity homepages.

above as a proof of concept. As case studies, we deployed the platform and developed two real-world applications. Our goal was to evaluate the usability and utility of the platform in building

structured Web portal applications. In this section, we present our case studies conducted, and discuss lessons learned from the case studies.

6.5.1 Building a Limnology Research Portal

Limnology is a science that studies lakes from all related aspects: geographical, biological, climatic, demographical, economical, among others. One important job of limnologists is to monitor lake-related ecosystem health. As an example, researchers at the Limnology Center in the University of Wisconsin-Madison actively monitor and study algae blooms on Lake Mendota. Algae blooms on the lake present a severe threat to public health. Toxins released by blue-green algae, for example, can cause significant health risks, affecting skin, liver and even nerve system. To be able to issue timely warnings when algae blooms outbreak, they need to monitor lake and weather conditions, process water quality data, and collect algae-related reports from various sources. Therefore, a portal that can automatically collect and integrate such information is of much practical value to the researchers. Building such a portal is a suitable case study for platform evaluation because it required us to interact with users (limnologists) from a scientific research field much different from ours. This is useful for us to evaluate whether the platform is generic and flexible in satisfying users' information needs and to examine its limitations. In building the portal, limnologists at the University of Wisconsin-Madison provided us a list of concepts they were interested in and the data sources from which information of these concepts can be obtained. Table 6.1 gives statistics on the portal data and IE/II programs developed, together with examples of interested concepts (entity and relationship types) and data sources from which information of the concepts was obtained. Figure 6.8 shows an example of lake entity homepage created by the Limnology research portal we built.

An important aspect of evaluating the utility of the platform is on measuring how much reduction in development effort taken to build a portal using the platform. One way to measure the reduction is by comparing the amount of work incurred with and without using the platform. In our case studies, we approximated this measure by counting how many operations we can realize by leveraging platform-provided operators and how many operations we have to implement from

Data sources and Portal Data	
No. of data sources	13
Examples: Wikipedia, Google Scholar, Google News, National Atmospheric Deposit Program Online.	
No. of text documents	11,193
No. of entity types	12
Examples: lakes, beaches, algae, streamflow sites, publications.	
No. of entity instances	74,497
No. of relationship types	15
Examples: lake-algae-cooccur, lake-mentioned-in-publication.	
No. of relationship instances	193,924
Portal IE/II Programs	
Total No. of IE/II programs	25
No. of IE/II programs executed inside RDBMS	18
No. of built-in operators used	40
No. of application-specific operators implemented	26

Table 6.1: Statistics on Limnology research portal development.

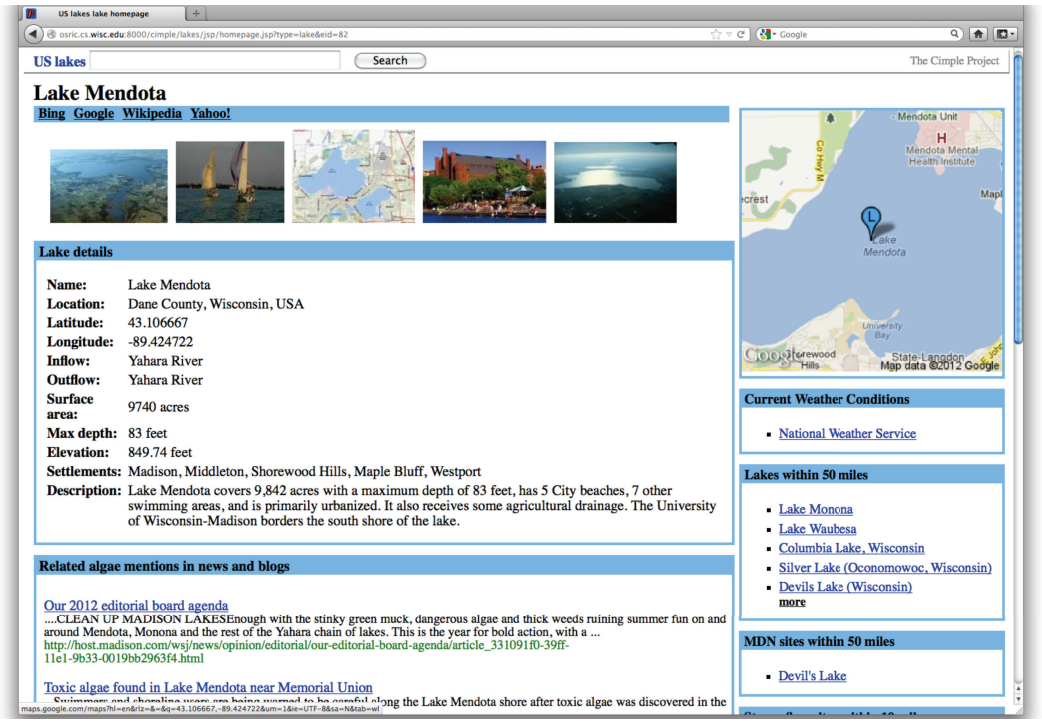


Figure 6.8: Homepage of Lake Mendota.

scratch. Specifically, we first depicted portal programs as workflows of operations, each performing a single task such as loading files from disk and extracting entity names from text. Then we counted the number of operations we had to implement (e.g., application-specific operations) and the number of operations we realized using the built-in operators the platform provided. The ratio between the two is 8 to 45, which indicates a significant saving using the platform. Most of the saving came from three sources: (1) using RDBMS capabilities to store and retrieve data, create indexes, and perform join operations, (2) leveraging the generic IE/II operators provided, and (3) using the built-in operators to manage entities and relationships. Although not shown in Table 6.1, saving on developing user services was most significant: using the platform, enabling user services such as keyword search required minimal configuration only.

6.5.2 Generating Natural Language Profiles for Musical Artists

During the time I was visiting Kosmix Inc. in summer 2010, a challenge engineers at the company faced was to automatically create profiles for popular musical artists (including individual musicians and musical bands) in a way that is easy for users to browse and consume. The structured information includes background information (such as a musician's birth place, birth date, and instruments played), past achievements (such as album and track releases), and current or future activities (such as upcoming concerts). This information could be obtained from various data sources such as Wikipedia[3], MusicBrainz[4] (an online music encyclopedia), and Last.fm[5] (an online music catalog).

One way to create profiles for musical artists would be to first have analysts collect structured data needed from those data sources and then have editors manually write profile text for each artist based on the collected structured data. But this approach is not cost effective and is not scalable. Also it incurs additional cost to keep the profiles up to date. A better approach is to have an automatic solution that collects the structured data from data sources regularly at appropriate intervals, integrates the data to be most comprehensive and accurate, and finally generates natural

[3] http://en.wikipedia.org/
[4] http://musicbrainz.org/
[5] http://www.last.fm/

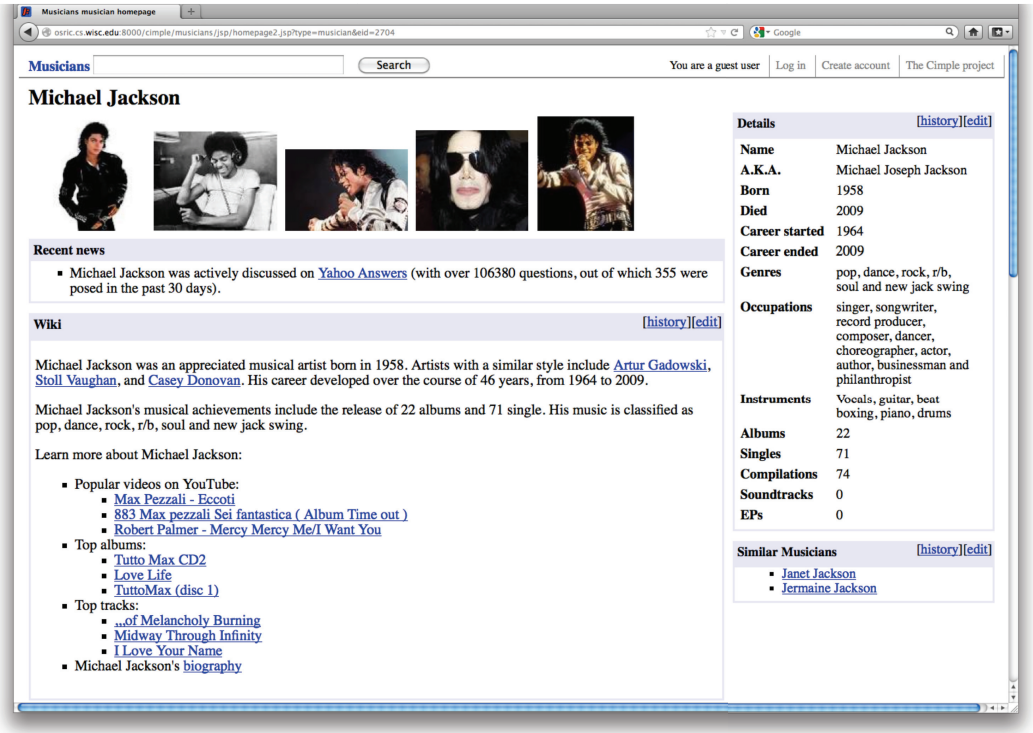


Figure 6.9: Homepage of Michael Jackson.

language text by instantiating pre-defined profile templates with the obtained structured data. This was the approach we took, and this problem made a good case study for us to evaluate the usability of the platform in building industrial strength applications.

Figure 6.9 shows the homepage of Michael Jackson created by the structured Web portal we built using the platform. The wiki text section was automatically generated by the application. Structured data such as name and birth year were embedded in the unstructured text. One of the application requirements was to make it easy for analysts and editors to modify the text generated if they want to correct an error or make an improvement. This was readily achieved by the built-in user feedback module. Figure 6.10 shows the record interface for editing the "Details" section of the page, and Figure 6.11 shows the wiki interface for providing feedback to the "Wiki" section. For example, to update the count of album releases, an analyst can update its number in the editing window in Figure 6.10. Once saved, the update will be pushed to the underlying database, and the album count mentioned in the "Wiki" section will be updated automatically.

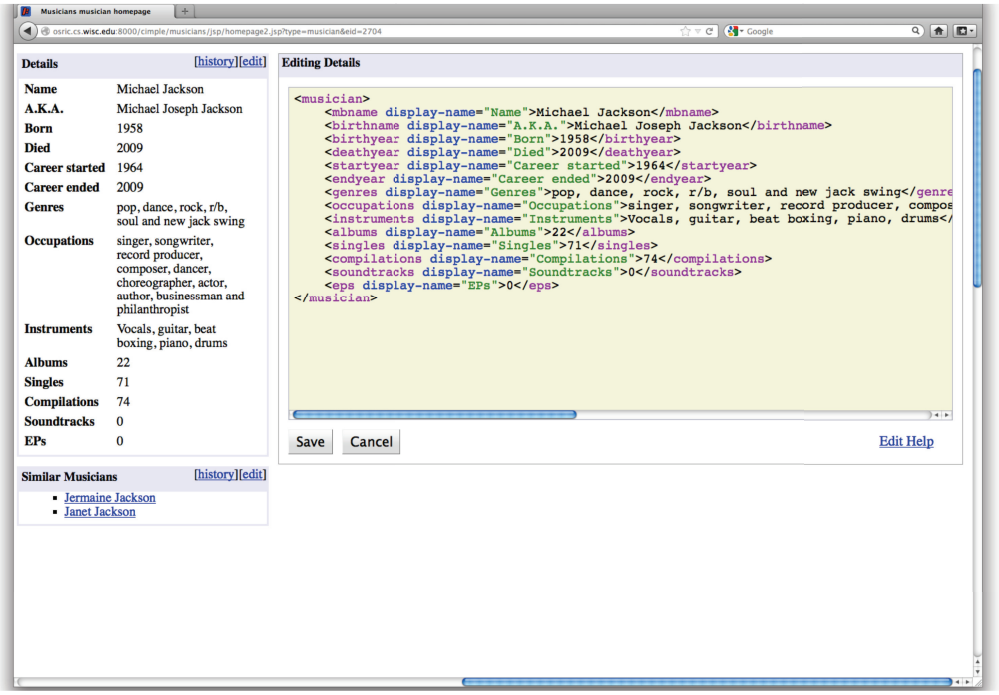


Figure 6.10: Record interface for editing artist details section.

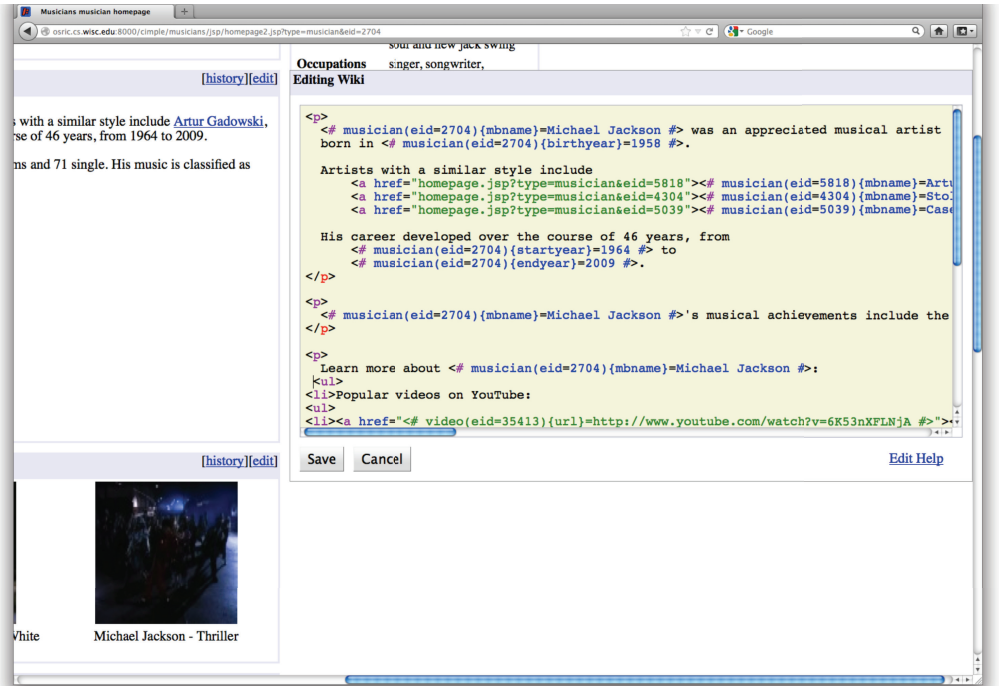


Figure 6.11: Wiki interface for editing wiki text section.

Data sources and Portal Data	
No. of data sources	8
Examples: Wikipedia, MusicBrainz.org, Last.fm, YouTube, Yahoo! Answer.	
No. of text documents	146,609
No. of entity types	7
Examples: musicians, bands, albums, tracks, videos.	
No. of entity instances	169,542
No. of relationship types	8
Examples: similar-in-style, album-release, track-release.	
No. of relationship instances	350,030
Portal IE/II Programs	
Total No. of IE/II programs	13
No. of IE/II programs executed inside RDBMS	10
No. of built-in operators used	45
No. of application-specific operators implemented	19

Table 6.2: Statistics on musical artist portal development.

Statistics on the application development are listed in Table 6.2. Similar to the case study of building a Limnology research portal, the platform saved a significant amount of effort in developing this portal application.

6.6 Observations

Although limited, the two case studies revealed the utility of a general platform in building structural Web portal applications. In particular we observed from the case studies that the current platform is useful from the following aspects:

- The built-in support for an entity-relationship model simplifies the tasks of collecting, analyzing and implementing user requirements (e.g., what the domain concepts are, how they are related, and where they can be obtained). We found that formulating concepts as entities and relationships is easy for users (domain experts and analysts) to comprehend and communicate. And the provided library of operators for creating and manipulating entities and relationships can significantly reduce the development effort.

- Relational database capabilities benefit writing and executing IE/II programs. With an RDBMS-based data storage and processing system, developers can enjoy the convenience, efficiency and robustness of RDBMS capabilities.

- Generic and easily customizable modules for realizing user services and user feedback are an integral part of the platform. While different applications may require different functions, support for common services makes it possible to quickly deliver a usable portal application.

The studies also revealed several limitations of the platform in developing real world applications:

- PostgreSQL, the RDBMS we adopted in the platform, supports user-defined functions written in multiple languages, such as SQL, C, Perl and Python. IE/II operations usually involve heavy string parsing and manipulation. Being able to use scripting languages like Perl that shines in its ability of complex string processing provides both us (as platform builders) and developers (as platform users) a convenient way to implement IE/II operations. Debugging the implementations inside the database, however, is difficult. We believe this is also the case with other RDBMS's. One plausible way to overcome this limitation is to provide an integrated development environment (IDE) in which developers can quickly carry out and test IE/II implementations, and deploy them easily afterwards.

- The current platform requires developers to implement IE/II operators following the relational model. That is, input and output of an operator need to be modeled as relations. This requirement brings the benefit that the so-written operators can be easily composed with each other and also with other relational operators to form large programs. The requirement, however, also comes with a cost of inflexibility. Some applications, especially those from scientific research areas, need to deal with data (e.g., time series) that follows other models. Since such data does not fit naturally with the relational model, the data and the programs that process the data may have to live outside the platform.

- The design and implementation of the current platform was based on a single machine. With the explosive growth of the amount of unstructured data, applications may need to process data that far exceeds the capacity of a single machine. Thus how to leverage systems and techniques in distributed computing in the platform is an important direction for future research.

Chapter 7

Related Work

As structural web portals are becoming more and more popular, the value of a general platform that developers can adopt to quickly build and deploy their portal applications has been increasing. In industry, numerous open-source Web application frameworks have been developed to increase the speed and agility of building applications. Popular examples include Ruby on Rails[1] and Django[2]. These frameworks provide a rich set of features to alleviate the overhead incurred by performing common tasks in Web application development. An important feature they offer is object-relational mapping with which developers can enjoy the flexibility of modeling application domain concepts as objects (entities and relationships) without worrying about translating the object representations into relational forms required by the underlying database. This is essentially in the same spirit of supporting an entity-relationship model and related operations in the platform we proposed. None of these frameworks, however, is tailored to building structured Web portal applications, where support for developing IE/II programs and managing their execution is crucial.

In the following, we first discuss related work in the context of building structured Web portals, then describe related work on the research problems we studied in Chapters 3-5. We conclude this chapter by discussing related industrial practices in developing IE/II applications and incorporating user feedback, their limitations, and opportunities for future research.

[1] http://rubyonrails.org/
[2] https://www.djangoproject.com/

7.1 Building Structured Web Portals

Existing solutions for building structured Web portals follow roughly two approaches. The first, *machine-based*, approach employs semi-automatic methods to extract and integrate data from a multitude of data sources, to create structured data portals. Examples include Cimple [45], Libra [71], Rexa [4] and BlogScope [15]. This approach incurs relatively little human effort, often generates a reasonable initial portal, keeps portals fresh with automatic updates, and enables structured services (querying, browsing, etc.) over portals. However, it usually suffers from inaccuracies, caused by imperfect extraction and integration methods, and limited coverage, because it can only infer whichever information is available in the data sources.

The second, *human-based*, approach manually deploys an initial portal, then replies on human users, dedicated developers or ordinary users, to revise and add contents. Examples include DBLP [63], Freebase [5], Intellipedia [6], and UMassWiki [7]. This approach avoids many problems of the machine-based approach, but suffers from its own limitations. In particular, it may be difficult to solicit sufficient user participation, and can incur significant user effort to keep portals up to date. Our approach, as supported by the platform, is a combination of both approaches.

The work closest to ours is that by DeRose [33] where the author proposed a top-down, compositional, and incremental approach to building structured community portals. In that work, the author also provided Cimple, a file-system-based software platform. This work made an important step toward a general platform, but it has several limitations. For example, the provided platform stores and processes portal data in the native file system. To handle the variety in the data (e.g., small string values and large files), the platform provides multiple data stores catering to storing different types of data. To support concurrency control, it provides a store-wide locking mechanism to realize coarse-grained ACID transactions. It is, however, difficult to extend the platform to support efficient access patterns and fine-grained concurrency control. Besides, the platform does not provide developers support for building user services and enabling user feedback. As we show in [42], supports for data exploitation and user feedback incorporation are crucial capabilities that a general platform should provide.

7.2 Matching Portal Schemas

Schema matching has received increasing attention over the past two decades (see [74, 44] for recent surveys). Many matching techniques have been developed, employing for example, machine learning [64, 41, 37], IR [28], and information theory [58]. Recent work has also explored incremental schema matching [17], self-organizing mapping [29], mapping debugging [27], mapping compilation [68], discovering mapping expression [11], information capacity in schema integration [70], data matching in ontology integration [83], hierarchy integration [88], and Web information integration [9, 16]. Once matches have been found and verified, they are typically elaborated into mappings [74] using a tool such as Clio [87].

A complementary problem (first raised in [54]) is then to revise schemas to make finding semantic matches easier. As far as we know, our work in Chapter 3 offers the first attack plan for this problem, placed in the context of revising mediated schemas of data-integration systems. The work closest to ours is eTuner [78]. That work however attacks a very different goal, namely, given a schema S, how to tune a matching system M (i.e., selecting the right matching components to be executed and correctly adjusting their knobs) to maximize matching accuracy over future schemas. In a sense, our problem can be considered complementary: given a matching system M, how to "tune" (i.e., revise) a schema S, to maximize matching accuracy of S with future schemas.

7.3 Opening Up Structured Web Portals for User Feedback

We are not aware of any published work that has studied how to model and incorporate user feedback into portal applications. Many portals (e.g., Wikipedia) do employ automatic programs (called "bots") to generate new pages according to some template, and to detect problems (e.g., vandalism) with current pages. But these programs do not contribute structured data nor do they update existing data, as we do here.

Perhaps the work closest to ours is Semantic Wikipedia [84]. This work develops new wiki language constructs that allow users to add structured data to wiki pages. We also develop similar wiki language constructs (see Section 4.3.3). But our constructs are far more powerful: we can

embed arbitrary ER data graphs in a wiki page, whereas the constructs in [84] in a sense only allow embedding node and relation *attributes*. More importantly, Semantic Wikipedia and several similar efforts, including semantic wikis [2] and Metaweb [3], have focused largely on *extending wiki languages* so that *users* can contribute structured data. They have not focused on allowing machines to contribute, nor do they study how to "push" structured contributions from users into an underlying database. Our work here is therefore complementary to these efforts.

Processing structured user edits in our context is a variation on the classical view update problem [14, 32]. Unlike relational view update, however, in our context users can also edit the schemas of views as well as of the underlying database. Since users employ the wiki interface, which is rather limited for expressing structured edits, this poses problems in interpreting user intentions that do not arise in relational view updates.

We recast processing structured user edits in our context as a problem of translating these edits across different user interfaces (wiki, ER, and relational, see Section 4.4.2). Such UI translations have been studied, e.g., translating a natural-language user query into a structured one [12, 65]. Translating free natural-language queries is well known to be difficult [12, 65]. Our problem here is still difficult, but more manageable, as we only translate *structured* edits.

Finally, our work in Chapter 4 can be viewed as a mass collaboration, Web 2.0 effort to build, maintain, and expand a hybrid structured data-text community database. Mass collaboration approaches to data management have recently received increasing attention in the database community (e.g., mass collaboration panel at VLDB-07, Web 2.0 track at ICDE-08, see also [10, 67, 85, 75, 40, 8]. Our work contributes to this emerging direction.

7.4 Incorporating User Feedback Into IE/II Programs

Soliciting and incorporating user feedback to improve IE/II results has received much attention in recent years. Recent works include using user feedback to correct schema matching results [41, 86], to generate integrated schemas [26], and to manage data in community information systems [45, 34, 60] and dataspace systems [47, 57]. While these works focused on leveraging user feedback to improve results of individual IE/II operations, our work aims to build an end-to-end

framework where user feedback can be incorporated into various stages of a complex workflow. To allow developers to write programs for such workflows, we also proposed a declarative language hlog, which extends recently developed IE language xlog [80] by providing constructs for specifying user interactions.

Using the derivations of updates, i.e., their provenance [21, 20] or lineage [30] for update reconciliation was also explored in [50]. In their work, the authors proposed a provenance model and used it for trust policies and incremental deletion. In contrast, we use provenance information to interpret the semantics of user updates and incorporate updates into execution results.

Many early works [18, 52, 53] have proposed and studied incremental view maintenance algorithms. Recent works have also considered how to incrementally execute IE/II operators [17, 24]. The proposed approaches, however, are specific to schema matching [17] and information extraction [24] settings. Thus they are not easily extensible to other operators. Our goal, in contrast, is to support incremental execution for a broad class of operators, whose semantics is unknown to the system.

Transaction concurrency control in relational databases has been well studied in the literature [49, 61, 62]. To the best of our knowledge, our work is the first attempt in exploiting the DAG structure of IE/II programs to provide efficient concurrency control outside RDBMS.

7.5 Observations on Industrial Practices

In recent years we have seen an increasing trend of building structured Web portals using automatic IE/II techniques, especially in the Web industry. During the past one-and-a-half years, I have been working in industrial companies building Web-scale IE/II applications. This work experience allows me to experience first-hand how such applications are developed in industry, and also to observe the "pain points" of current practices and the need for a general platform. The work conducted in this dissertation makes a step towards addressing this need. In the following, I describe related industrial practices in developing IE/II applications and incorporating user feedback, their limitations, and how our work offers a promising direction to address these limitations.

Developing IE/II applications: Over the past decade, a proliferation of open-source software has been developed to perform the tasks of extracting and integrating structured data from unstructured data. Examples include Apache openNLP[3] and NLTK[4]. These tools provide support for common IE/II tasks such as named entity extraction and coreference resolution, and enable developers to quickly assemble IE/II programs by wiring needed components together using scripting languages such as Perl and Python. While using these tools shortens the time to deliver a working solution, the so-developed IE/II programs are usually hard to debug, maintain and extend. And a declarative framework for developing IE/II applications has been a long standing desire in industry. Effort has been made to develop such frameworks. A notable example is the Apache UIMA project[5], whose goal is to provide developers a compositional software framework for creating, composing and deploying unstructured data management applications. The need for a declarative framework and the effort from the developer community validate our speculation on the utility of a declarative language for writing IE/II programs and our design of providing such a language, together with a built-in library of common IE/II operators, as an integral part of a general platform.

Incorporating user feedback: In this dissertation, we advocate that leveraging user feedback to improve the quality of IE/II results is important in building structured Web portal applications. This is indeed the case in industrial application development, where user feedback is often indispensable in building successful applications. User feedback is needed in these applications because automatic IE/II solutions rarely achieve the high level of accuracy applications require and thus humans must be kept in the loop to improve the accuracy. As a common practice, a team of data analysts often work closely with application developers to assess the quality of IE/II results and provide feedback (e.g., by verifying the input data to the programs and by correcting IE/II errors). Feedback from analysts is usually accumulated into batches and processed periodically by some custom programs developers write. This practice incurs a delay between the time user feedback is provided and the time it is incorporated. Furthermore, it requires developers to build tools that analysts can use to provide feedback and to write programs that are specifically for interpreting

[3]http://opennlp.apache.org/
[4]http://nltk.org/
[5]http://uima.apache.org/

and incorporating user feedback. Thus our work on developing a systematic framework for users (including developers, data analysts and ordinary users) to easily provide feedback and for the programs to automatically incorporate the feedback offers a promising direction to address these problems.

Chapter 8

Conclusions

In this dissertation, we described our envisioned platform to building structured Web portals, and provided an end-to-end implementation. We also identified several research challenges arising from building the platform, and presented out solutions. In this chapter, we summarize the main contributions of this dissertation, and discuss future directions for this work.

8.1 Contributions

The main contributions of this dissertation are the study of the capabilities and architecture of a general platform for building structured Web portals and the demonstration that such a platform can be useful for building portals in real-world applications.

Throughout the work, we also identified and addressed several research challenges. Chapter 3 studied how to analyze and revise portal schemas, which serve as a uniform interface for storing and querying portal data, to make them easier to match in aggregating structured data from heterogeneous data sources. We proposed a promising solution leveraging synthetic schemas, and showed that synthetic schemas are useful in improving the matchability of portal schemas.

Chapter 4 studied the problems of opening up a structured web portal for user feedback. We provided solutions to modeling the underlying structured database, representing views over the database, and exposing views via a wiki interface. We also provided an efficient solution to process user edits and "push" these edits into the underlying database.

Chapter 5 studied the problem of efficiently incorporating user feedback into IE/II programs that collect and integrate structured portal data. We proposed a framework that allows developers to

quickly write declarative programs with the capabilities of incorporating feedback. Optimization techniques were also provided to improve the efficiency of program execution in the presence of user feedback.

8.2 Future Directions

Much work remains in building a flexible, extensible and scalable platform, and also in improving the solutions we proposed to address the research challenges arise. We discussed directions for further work at the end of Chapters 3-6. Below we briefly reiterate a few of them.

First, with respect to platform development, an important direction to pursue is to study how to distribute data and IE/II program execution across multiple machines while retaining the simplicity of writing the programs. Ideas from Hive [82], a distributed data warehouse system, and its query language HiveQL, a SQL-like language that hides the complexity of distributed system from developers, could be borrowed and adapted.

Second, with respect to matching portal schemas with data source schemas, it would be interesting to study how to leverage user effort to provide and improve matches between schemas. One solution would be to provide user an interactive environment where a user can interact with the matching tool to generate and revise matches. Being able to do so would also make it possible for users to help expand the coverage of a portal by contributing and integrating data sources into the portal.

Finally, with respect to incorporating user feedback, in this work we restrict a view (via which the underlying portal data is exposed to UIs) to be defined over a single table. In other words, any piece of data a user can edit comes from a single table. The restriction simplifies the process of incorporating user's edit. The next step would be to extend the framework to support views over multiple tables. To do so would require us to revisit the semantics of program execution in light of user feedback.

8.3 Closing Remarks

Structural Web portal applications that extract and integrate structured data across the Web are becoming increasingly common. By providing developers with a general platform, we enable the developers to quickly build and deploy portal applications which otherwise would require a significant amount of effort to develop. It is our hope that this work would help and drive further research on related problems and those beyond, in the broader context of unstructured data management and Web data integration.

Bibliography

[1] http://dblife-labs.cs.wisc.edu/wiki-test/index.php/main_page.

[2] http://en.wikipedia.org/wiki/semantic_wiki.

[3] http://metaweb.com/.

[4] http://rexa.info/.

[5] http://www.freebase.com/.

[6] http://www.intelink.gov/.

[7] http://www.umasswiki.com/.

[8] Sixth international workshop on information integration on the web. 2007.

[9] K. Aberer, P. Cudre-Mauroux, and M. Hauswirth. The chatty web: Emergent semantics through gossiping. In *WWW-03*.

[10] S. Amer-Yahia. A database solution to search 2.0. *WebDB-07*.

[11] Y. An, A. Borgida, R. J. Miller, and J. Mylopoulos. A semantic approach to discovering schema mapping expressions. In *ICDE-07*.

[12] I. Androutsopoulos, G. D. Ritchie, and P. Thanisch. Natural language interfaces to databases–an introduction. *Journal of Language Engineering*, 1(1):29–81, 1995.

[13] D. Aumueller, H. H. Do, S. Massmann, and E. Rahm. Schema and ontology matching with COMA++. In *SIGMOD-05*.

[14] F. Bancilhon and N. Spyratos. Update semantics of relational views. *ACM Transactions on Database Systems*, 6(4):557–575, 1981.

[15] N. Bansal and N. Koudas. Blogscope: Spatio-temporal analysis of the blogosphere. In *WWW-07*.

[16] L. Barbosa, J. Freire, and A. Silva. Organizing hidden-web databases by clustering visible web documents. In *ICDE-07*.

[17] P. A. Bernstein, S. Melnik, and J. E. Churchill. Incremental schema matching. In *VLDB-06*.

[18] J. A. Blakeley, P.-A. Larson, and F. W. Tompa. Efficiently updating materialized views. *SIGMOD Record*, 15(2), 1986.

[19] P. Bohannon, S. Merugu, C. Yu, V. Agarwal, P. DeRose, A. Iyer, A. Jain, V. Kakade, M. Muralidharan, R. Ramakrishnan, and W. Shen. Purple SOX extraction management system. *SIGMOD Record*, 37(4), 2008.

[20] P. Buneman, S. Khanna, and W. C. Tan. Why and where: A characterization of data provenance. In *ICDT-01*.

[21] P. Buneman and W. C. Tan. Provenance in databases. In *SIGMOD-07*.

[22] X. Chai, M. Sayyadian, A. Doan, A. Rosenthal, and L. Seligman. Analyzing and revising data integration schemas to improve their matchability. In *VLDB-08*.

[23] X. Chai, B.-Q. Vuong, A. Doan, and J. F. Naughton. Efficiently incorporating user feedback into information extraction and integration programs. In *SIGMOD-09*.

[24] F. Chen, A. Doan, J. Yang, and R. Ramakrishnan. Efficient information extraction over evolving text data. In *ICDE-08*.

[25] F. Chen, B. J. Gao, A. Doan, J. Yang, and R. Ramakrishnan. Optimizing complex extraction programs over evolving text data. In *SIGMOD-09*.

[26] L. Chiticariu, P. G. Kolaitis, and L. Popa. Interactive generation of integrated schemas. In *SIGMOD-08*.

[27] L. Chiticariu and W. C. Tan. Debugging schema mappings with routes. In *VLDB-06*.

[28] C. Clifton, E. Housman, and A. Rosenthal. Experience with a combined approach to attribute-matching across heterogeneous databases. In *DS-7*, 1997.

[29] P. Cudre-Mauroux, S. Agarwal, A. Budura, P. Haghani, and K. Aberer. Self-organizing schema mappings in the gridvine peer data management system. In *VLDB-07*.

[30] Y. Cui and J. Widom. Lineage tracing for general data warehouse transformations. In *VLDB-01*.

[31] H. Cunningham, D. Maynard, K. Bontcheva, and V. Tablan. GATE: A framework and graphical development environment for robust NLP tools and applications. In *ACL-02*.

[32] U. Dayal and P. A. Bernstein. On the correct translation of update operations on relational views. *ACM Transactions on Database Systems*, 7(3):381–416, 1982.

[33] P. DeRose. *Building Structured Web Community Portals: A Top-Down, Compositional, and Incremental Approach*. PhD thesis, University of Wisconsin-Madison, 2009.

[34] P. DeRose, X. Chai, B. Gao, W. Shen, A. Doan, P. Bohannon, and J. Zhu. Building community wikipedias: A human-machine approach. In *ICDE-08*.

[35] P. DeRose, W. Shen, F. Chen, A. Doan, and R. Ramakrishnan. Building structured web community portals: The case for an incremental and compositional approach. In *VLDB-07*.

[36] P. DeRose, W. Shen, F. Chen, A. Doan, and R. Ramakrishnan. Building structured web data portals: A top-down, compositional, and incremental approach. In *VLDB-07*.

[37] R. Dhamankar, Y. Lee, A. Doan, A. Halevy, and P. Domingos. iMAP: Discovering complex matches between database schemas. In *SIGMOD-04*.

[38] P. Diaconis. Group representation in probability and statistics. *IMS Lecture Series. Institute of Mathematical Statistics*, 11, 1988.

[39] H. H. Do, S. Melnik, and E. Rahm. Comparison of schema matching evaluations. In *Web, Web-Services, and Database Systems*, 2002.

[40] A. Doan, P. Bohannon, R. Ramakrishnan, X. Chai, P. DeRose, B. Gao, and W. Shen. User-centric research challenges in community information management systems. *IEEE Data Engineering Bulletin, Special Issue on Data Management in Social Networks*, 2007.

[41] A. Doan, P. Domingos, and A. Y. Halevy. Reconciling schemas of disparate data sources: A machine-learning approach. In *SIGMOD-01*.

[42] A. Doan, J. F. Naughton, A. Baid, X. Chai, F. Chen, T. Chen, E. Chu, P. DeRose, B. Gao, C. Gokhale, J. Huang, W. Shen, and B.-Q. Vuong. The case for a structured approach to managing unstructured data. In *CIDR-09*.

[43] A. Doan, J. F. Naughton, R. Ramakrishnan, A. Baid, X. Chai, F. Chen, T. Chen, E. Chu, P. DeRose, B. Gao, C. Gokhale, J. Huang, W. Shen, and B.-Q. Vuong. Information extraction challenges in managing unstructured data. *SIGMOD Record*, 37(4), 2008.

[44] A. Doan, N. F. Noy, and A. Y. Halevy. Introduction to the special issue on semantic integration. *SIGMOD Record*, 33(4), 2004.

[45] A. Doan, R. Ramakrishnan, F. Chen, P. DeRose, Y. Lee, R. McCann, M. Sayyadian, and W. Shen. Community information management. *IEEE Data Eng. Bull.*, 29(1), 2006.

[46] D. Ferrucci and A. Lally. UIMA: An architectural approach to unstructured information processing in the corporate research environment. *Nat. Lang. Eng.*, 10(3-4), 2004.

[47] M. Franklin, A. Halevy, and D. Maier. From databases to dataspaces: A new abstraction for information management. *SIGMOD Record*, 34(4), 2005.

[48] C. Giles, K. Bollacker, and S. Lawrence. Citeseer: An automatic citation indexing system. In *DL-98*.

[49] J. Gray, R. A. Lorie, G. R. Putzolu, and I. L. Traiger. Granularity of locks and degrees of consistency in a shared data base. In *IFIP-76*.

[50] T. J. Green, G. Karvounarakis, Z. G. Ives, and V. Tannen. Update exchange with mappings and provenance. In *VLDB-07*.

[51] H. Gregersen and C. S. Jensen. Temporal entity-relationship models - a survey. *Knowledge and Data Engineering*, 11(3):464–497, 1999.

[52] T. Griffin and L. Libkin. Incremental maintenance of views with duplicates. *SIGMOD Record*, 24(2), 1995.

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

[54] A. Halevy and C. Li. Information integration research: The NSF IDM workshop breakout session. In *IDM-03*.

[55] J. Huang, T. Chen, A. Doan, and J. F. Naughton. On the provenance of non-answers to queries over extracted data. *PVLDB*, 1(1), 2008.

[56] P. G. Ipeirotis, E. Agichtein, P. Jain, and L. Gravano. To search or to crawl?: Towards a query optimizer for text-centric tasks. In *SIGMOD-06*.

[57] S. R. Jeffery, M. J. Franklin, and A. Y. Halevy. Pay-as-you-go user feedback for dataspace systems. In *SIGMOD-08*.

[58] J. Kang and J. Naughton. On schema matching with opaque column names and data values. In *SIGMOD-03*.

[59] G. Kasneci, M. Ramanath, F. Suchanek, and G. Weikum. The YAGO-NAGA approach to knowledge discovery. *SIGMOD Record*, 37(4), 2008.

[60] Y. Katsis, A. Deutsch, and Y. Papakonstantinou. Interactive source registration in community-oriented information integration. In *VLDB-08*.

[61] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2), 1981.

[62] P. L. Lehman and S. B. Yao. Efficient locking for concurrent operations on b-trees. *ACM Trans. Database Syst.*, 6(4), 1981.

[63] M. Ley. The DBLP computer science bibliography: Evolution, research issues, perspectives. 2476, 2002.

[64] W. Li and C. Clifton. SEMINT: A tool for identifying attribute correspondence in heterogeneous databases using neural networks. *DKE*, 33, 2000.

[65] Y. Li, H. Yang, and H. V. Jagadish. Constructing a generic natural language interface for an xml database. In *EDBT-06*.

[66] C. D. Manning, P. Raghavan, and H. Schtze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[67] R. McCann, A. Doan, V. Varadarajan, A. Kramnik, and C. Zhai. Building data integration systems: A mass collaboration approach. In *WebDB-03*.

[68] S. Melnik, A. Adya, and P. A. Bernstein. Compiling mappings to bridge applications and databases. In *SIGMOD-07*.

[69] S. Melnik, H. Garcia-Molina, and E. Rahm. Similarity flooding: a versatile graph matching algorithm. In *ICDE-02*.

[70] R. J. Miller, Y. E. Ioannidis, and R. Ramakrishnan. The use of information capacity in schema integration and translation. In *VLDB-93*.

[71] Z. Nie, J. Wen, and W. Ma. Object-level vertical search. In *CIDR-07*.

[72] M. Petropoulos, A. Deutsch, and Y. Papakonstantinou. Interactive query formulation over web service-accessed sources. In *SIGMOD-06*.

[73] PostgreSQL Global Development Group. PostgreSQL. http://www.postgresql.org/.

[74] E. Rahm and P. A. Bernstein. A survey of approaches to automatic schema matching. *VLDB Journal*, 10(4), 2001.

[75] R. Ramakrishnan. Community systems: The world online. In *CIDR-07*.

[76] R. Ramakrishnan and J. Gehrke. *Database management systems (3. ed.)*. McGraw-Hill, 2003.

[77] F. Reiss, S. Raghavan, R. Krishnamurthy, H. Zhu, and S. Vaithyanathan. An algebraic approach to rule-based information extraction. In *ICDE-08*.

[78] M. Sayyadian, Y. Lee, A. Doan, and A. Rosenthal. Tuning schema matching software using synthetic scenarios. In *VLDB-05*.

[79] W. Shen, P. DeRose, R. McCann, A. Doan, and R. Ramakrishnan. Toward best-effort information extraction. In *SIGMOD-08*.

[80] W. Shen, A. Doan, J. F. Naughton, and R. Ramakrishnan. Declarative information extraction using datalog with embedded extraction predicates. In *VLDB-07*.

[81] R. T. Snodgrass. *Developing Time-Oriented Database Applications in SQL*. Morgan Kaufmann Publishers, Inc., 1999.

[82] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive - a warehousing solution over a map-reduce framework. *PVLDB*, 2, 2009.

[83] O. Udrea, L. Getoor, and R. J. Miller. Leveraging data and structure in ontology integration. In *SIGMOD-07*.

[84] M. Volkel, M. Krotzsch, D. Vrandecic, H. Haller, and R. Studer. Semantic wikipedia. In *WWW-06*.

[85] F. Wang, C. Rabsch, P. Kling, P. Liu, and P. John. Web-based collaborative information integration for scientific research. In *ICDE-07*.

[86] W. Wu, C. T. Yu, A. Doan, and W. Meng. An interactive clustering-based approach to integrating source query interfaces on the deep web. In *SIGMOD-04*.

[87] L. L. Yan, R. J. Miller, L. M. Haas, and R. Fagin. Data driven understanding and refinement of schema mappings. In *SIGMOD-01*.

[88] K. Zhao, R. Ikeda, and H. Garcia-Molina. Merging hierarchies using object placement. In *ICDE-08*.

APPENDIX
Basic Relational Actions

We define a set of basic relational actions that a user can execute over V_d, V_s, G_d and G_s. There are 10 actions for V_d, 8 for V_s, 10 for G_d, and 8 for G_s. In the following, we give our implementation of each action. To distinguish actions in different categories, we prefix each action by its category name. For example, we denote action a_i for V_d as V_d::a_i.

A.1 Actions for V_d

Action a_1: Modify an Entity Attribute Value

Steps:

1. Execute G_d::a_1.

Action a_2: Modify a Relation Attribute Value

Steps:

1. Execute G_d::a_2.

Action a_3: Insert an Entity Attribute

Let eid be the entity ID and E be the entity type. Let A be the attribute to insert.

Steps:

1. Execute G_d::a_3;
2. Add an inclusive path $E(id = eid)\{A\}$ to V_s.

Action a_4: Insert a Relation Attribute

Let rid be the relation ID and R be the relation type. Let A be the attribute to insert.

Steps:

1. Execute G_d::a_4;
2. Add an inclusive path $R(id = rid)\{A\}$ to V_s.

Action a_5: Insert a New Entity

Let e be the entity to insert and E be its type.

Steps:

1. Execute G_d::a_5, let eid be the ID of e;
2. Add an inclusive path $E(id = eid)$ to V_s.

Action a_6: Insert a New Relation

Let r be the relation to insert and R be its type. Let eid_1 and eid_2 be the IDs of the two entities that r relates, and E_1 and E_2 be their types.

Steps:

1. Execute G_d::a_6, let rid be the ID of r;
2. Add an inclusive path $E_1(id = eid_1).R(id = rid).E_2(id = eid_2)$ to V_s.

Action a_7: Delete an Entity Attribute

Let eid be the ID of the entity and E be its type. Let A be the attribute to delete.

Steps:

1. Execute G_d::a_7;
2. Add an exclusive path $E(id = eid)\{A\}$ to V_s.

Action a_8: Delete a Relation Attribute

Let rid be the ID of the relation and R be its type. Let A be the attribute to delete.

Steps:

1. Execute G_d::a_8;

2. Add an exclusive path $R(id = rid)\{A\}$ to V_s.

Action a_9: Delete an Entity

Let e be the entity to delete. Let eid be e's ID and E be e's type.

Steps:

1. FOR each relation r that relates e DO

   Execute V_d::a_{10};

2. FOR each attribute A (including *exists*) of e DO

   Execute V_d::a_7;

3. Add an exclusive path $E(id = eid)$ to V_s.

Action a_{10}: Delete a Relation

Let r be the relation to delete. Let rid be r's ID and R be r's type. Let eid_1 and eid_2 be the IDs of the entities that r relates, and E_1 and E_2 be their types.

Steps:

1. FOR each attribute A (including *exists*) of r DO

   Execute V_d::a_8;

2. Add an exclusive path $E_1(id = eid_1).R(id = rid).E_2(id = eid_2)$ to V_s.

A.2 Actions for V_s

Action a_1: Insert an Entity Attribute

Let eid be the ID of the entity and E be its type. Let A be the attribute to insert.

Steps:

1. Add an inclusive path $E(id = eid)\{A\}$ to V_s.

Action a_2: Insert a Relation Attribute

Let rid be the ID of the relation and R be its type. Let A be the attribute to insert.

Steps:

1. Add an inclusive path $R(id = rid)\{A\}$ to V_s.

Action a_3: Insert an Entity

Let eid be the ID of the entity and E be its type.

Steps:

1. Add an inclusive path $E(id = eid)$ to V_s.

Action a_4: Insert a Relation

Let r be the relation to insert. Let rid be r's ID and R be r's type. Let eid_1 and eid_2 be the IDs of the entities that r relates, and E_1 and E_2 be their types.

Steps:

1. Add an inclusive path $E_1(id = eid_1).R(id = rid).E_2(id = eid_2)$ to V_s.

Action a_5: Delete an Entity Attribute

Let eid be the ID of the entity and E be its type. Let A be the attribute to delete.

Steps:

1. Add an exclusive path $E(id = eid)\{A\}$ to V_s.

Action a_6: Delete a Relation Attribute

Let rid be the ID of the relation and R be its type. Let A be the attribute to delete.

Steps:

1. Add an exclusive path $R(id = rid)\{A\}$ to V_s.

Action a_7: Delete an Entity

Let e be the entity to delete. Let eid be e's ID and E be e's type.

Steps:

1. FOR each relation r that relates e DO

   Execute V_s::a_8;

2. FOR each attribute A (including *exists*) of e DO

   Execute V_s::a_5;

3. Add an exclusive path $E(id = eid)$ to V_s.

Action a_8: Delete a Relation

Let r be the relation to insert. Let rid be r's ID and R be r's type. Let eid_1 and eid_2 be the IDs of the entities that r relates, and E_1 and E_2 be their types.

Steps:

1. FOR each attribute A (including *exists*) of r DO

   Execute V_s::a_6;

2. Add an exclusive path $E_1(id = eid_1).R(id = rid).E_2(id = eid_2)$ to V_s.

A.3 Actions for G_d

We use the following notations to represent tables in G:

E_A_m – table for attribute A of entity type E that stores attribute values entered by machine M;

E_A_u – table for attribute A of entity type E that stores attribute values entered by human users;

E_A_p – table for attribute A of entity type E that stores attribute values used in generating V_d;

R_A_m, R_A_u, R_A_p – similar to those above but for relation type R instead;

R_ID – relation ID table for relation type R.

Action a_1: Modify an Entity Attribute Value

Let E be the type of the entity and A be the attribute. Let w be the ID of the user who modifies A.

Steps:

1. IF $w = M$ THEN

   Logically delete the current value of A in E_A_m;

   Insert the new value of A into E_A_m;

   ELSE

   Logically delete the current value of A in E_A_u;

   Insert the new value of A into E_A_u;

2. IF the current value in E_A_p was entered by M

   OR w != M THEN

   Logically delete the current value of A in E_A_p;

   Insert the new value of A into E_A_p;

Action a_2: Modify a Relation Attribute Value

Let R be the type of the relation and A be the attribute. Let w be the ID of the user who modifies A.

Steps:

1. IF $w = M$ THEN

   Logically delete the current value of A in R_A_m;

   Insert the new value of A into R_A_m;

   ELSE

   Logically delete the current value of A in R_A_u;

   Insert the new value of A into R_A_u;

2. IF the current value in R_A_p was entered by M

   OR w != M THEN

   Logically delete the current value of A in R_A_p;

   Insert the new value of A into R_A_p;

Action a_3: Insert an Entity Attribute

Let eid be the ID of the entity and E be its type. Let A be the attribute and w be the ID of the user who inserts A.

Steps:

1. IF $w = M$ THEN

   IF exists a record with $id = eid$

   AND $stop$="9999-12-31 23:59:59" in E_A_m THEN

   Logically delete the record;

   Insert the new value of A into E_A_m;

   ELSE

   IF exists a record with $id = eid$

   AND $stop$="9999-12-31 23:59:59" in E_A_m THEN

   Logically delete the record;

   Insert the new value of A into E_A_u;

2. IF exists a record with $id = eid$

   AND $stop$="9999-12-31 23:59:59" in E_A_p THEN

   IF the record was entered by M OR $w! = M$ THEN

   Logically delete the record;

   Insert the new value of A into E_A_p;

Action a_4: Insert a Relation Attribute

Let rid be the ID of the relation and E be its type. Let A be the attribute and w be the ID of the user who inserts A.

Steps:

1. IF $w = M$ THEN

   IF exists a record with $id = rid$

   AND $stop$="9999-12-31 23:59:59" in R_A_m THEN

   Logically delete the record;

   Insert the new value of A into R_A_m;

   ELSE

   IF exists a record with $id = rid$

   AND $stop$="9999-12-31 23:59:59" in R_A_m THEN

   Logically delete the record;

   Insert the new value of A into R_A_u;

2. IF exists a record with $id = rid$
   AND $stop$="9999-12-31 23:59:59" in R_A_p THEN
   IF the record was entered by M OR $w! = M$ THEN
   Logically delete the record;
   Insert the new value of A into R_A_p;

Action a_5: Insert a New Entity

Let E be the entity type and max_eid be the largest ID in table *entity_ID*.

Steps:

1. Insert record $(max\_eid + 1, E)$ into *entity_ID*.

Action a_6: Insert a New Relation

Let r be the relation to insert and R be its type. Let $eid1$ and $eid2$ be the IDs of the entities that r relates. Let max_rid be the largest ID in table *R_ID*.

Steps:

1. Insert record $(max\_rid + 1, eid1, eid2)$ into *R_ID*.

Action a_7: Delete an Entity Attribute

Steps:

1. Execute G_d::a_1 with NULL as the attribute value.

Action a_8: Delete a Relation Attribute

Steps:

1. Execute G_d::a_2 with NULL as the attribute value.

Action a_9: Delete an Entity

Let e be the entity to delete.

Steps:

1. FOR each relation r that relates e DO

Execute G_d::a_{10};

2. FOR each attribute A (including *exists*) of e DO

   Execute G_d::a_7;

Action a_{10}: Delete a Relation

Let r be the relation to delete.

Steps:

1. FOR each attribute A (including *exists*) of r DO

   Execute G_d::a_8;

A.4 Actions for G_s

Action a_1: Create an Entity Attribute

Let E be the type of the entity and A be the attribute to create.

Steps:

1. Insert attribute A into table *meta_attribute*;
2. Create tables E_A_m, E_A_u and E_A_p.

Action a_2: Create a Relation Attribute

Let R be the type of the relation and A be the attribute to create.

Steps:

1. Insert attribute A into table *meta_attribute*;
2. Create tables R_A_m, R_A_u and R_A_p.

Action a_3: Create an Entity Type

Let E be the entity type to create.

Steps:

1. Insert entity type E into table *meta_entity*;
2. Execute G_s::a_1 for attribute *exists*.

Action a_4: Create a Relation Type

Let R be the relation type to create.

Steps:

1. Insert relation type R into table *meta_relation*;
2. Create table R_ID;
3. Execute G_s::a_2 for attribute *exists*.

Action a_5: Drop an Entity Attribute

Let E be the entity type and A be the attribute to drop.

Steps:

1. Logically delete attribute A in table *meta_attribute*;
2. Logically delete all records in E_A_m, E_A_u and E_A_p.

Action a_6: Drop a Relation Attribute

Let R be the relation type and A be the attribute to drop.

Steps:

1. Logically delete attribute A in table *meta_attribute*;
2. Logically delete all records in R_A_m, R_A_u and R_A_p.

Action a_7: Drop an Entity Type

Let E be the entity type to drop.

Steps:

1. FOR each relation type R that relates E DO

   Execute G_s::a_8;
2. FOR each attribute A (including *exists*) of E DO

   Execute G_s::a_5;
3. Logically delete entity type E in *meta_entity*;

Action a_8: Drop a Relation Type

Let R be the relation type to drop.

Steps:

1. FOR each attribute A (including *exists*) of R DO

   Execute G_s::a_6;

2. Logically delete entity type R in *meta_relation*;