Reducing End-User Burden in Everyday Data Organization

by

Li Qian

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2013

Doctoral Committee:

Professor Hosagrahar V. Jagadish, Co-Chair
Assistant Professor Michael J. Cafarella, Co-Chair
Assistant Professor Eytan Adar
Assistant Professor Kristen R. LeFevre
Assistant Professor Qiaozhu Mei

To all the beloved.

ACKNOWLEDGEMENTS

It has been a long journey. I could not have completed it without the incredibly helpful guidance from my dear advisor, Professor H. V. Jagadish. His comprehensive knowledge of conducting great research, insightful vision towards real-life problems, profound wisdom in advising his students and extremely kind personality together make a lifetime treasure to me. It has been a true privilege to be his student.

I would also like to sincerely thank my dissertation committee, Professor H. V. Jagadish, Professor Michael J. Cafarella, Professor Kristen R. LeFevre, Professor Eytan Adar and Professor Qiaozhu Mei, for all their time and effort to help improve and refine my thesis. This dissertation would not be possible without their careful reviews and valuable comments.

I am fortunate to have had two summer internships at IBM Almaden Research Center. I would like to thank Lucian Popa, Mauricio A Hernandez and Bogdan Alexe for being my great mentors and introducing me to the frontier of industrial research and development. I would also like to thank Anna Wu, Boduo Li, Congle Zhang, Di pan, Flo Wfi Rian, Hailiang Huang, Kun Hu, Liping Peng, Mengdie Hu, Mengmeng Liu, Ming Ji, Mu Qiao, Teng Zhou, Wei Ning, Xiao Zhang, Xide Lin, Yinan Li, Yuanyuan Tian, Yunyao Li, Zhuowei Bao and Ziyang Liu, for sharing the passionate and memorable summer time with me.

It has been an honor to be part of the University of Michigan Database Group. I would like to thank my colleagues, Alex Roper, Allie Mazzia, Anna Shaverdian, Bin Liu, Chun-Hung Hsiao, Dan Fabbri, Dolan Antenucci, Fei Li, Fernando Farfan, Glenn Tarcea, Jing Zhang, Lujun Fang, Magesh Jayapandian, Manish Singh, Matthew Burgess, Michael Anderson, Rajesh Bejugam and Zhe Chen, for all the support and joy you have brought to me over these years.

I am lucky to have a wonderful group of friends in our CSE department and the lovely city of Ann Arbor. Thank you Bangxi Hu, Caoxie Zhang, Fangjian Jin, Fangshuo Zhao, Feng Qian, Gang Su, Hong Li, Huilin Chen, Huizhong Tan, Jie Yu, Jing Zhang, Jinjing Yang, Jiying Zheng, Junxian Huang, Ken Ling, Lingfeng Gao, Lisha Li, Mengyun Liu, Qiang Xu, Shuyi Li, Wen Chen, Xin Hu, Yang-bing Lou, Yanqin Cui, Yi Li, Ying Yu, Yinying Zhang, Yudong Gao, Yunjing Xu, Yuxiang Xie, Zhaoguang Wang, Zheyu Dong, Zhiyun Qian, Zhongni Tang and Zhoutong Fu for sharing your memory with me during this long journey. Thank my roommate Zhe Zhao, for correcting my unhealthy life style and introducing me to the wonderful world of Kendo. Thank you Watanabi Sensi and Joe Sen-si, for teaching me Kendo and training me to be both mentally and physically stronger. Thank my old friends, Fan Ye, Linying He, Luping Shen, Mi Luo, Song He, Ting Shen, Yu Chen, Zheda Chen and Zhengyu Tu for taking care of the long friendship with me. Thank my unbelievably sweet kitties, Mayhem and Mischief, for not grabbing my mouse during my work.

And above all, I would like to thank my parents, to whom I have owned too much. Thank you for all your love, care, understanding, support and blessings.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

Reducing End-User Burden in Everyday Data Organization

by
Li Qian

Chair: Hosagrahar V. Jagadish

As digital data permeates every aspect of our daily life, end-users find it appealing to organize their everyday data electronically. In fact, end-users are already used to managing their personal data such as contact books and calendars in electronic devices. Meanwhile, the desire for organizing more information into the computer is expanding for a broader group of users. For example, a scientist may need to regularly manage a substantial amount of science data on his desktop. Similarly, an online market investigator's daily task may be to collect data from various websites, in order to build a universal data repository for later analysis.

However, to organize such everyday data is challenging for these end-users. This is primarily because, end-users have limited knowledge about data schema, while a good schema is key to data management tasks such as database design, data transformation and data integration. While the user is struggling with schema in these tasks, various cognitive and operational burdens emerge.

First of all, when designing her data collection, the user has the burden to abstract her mental model of her real-life data into a reasonable database schema design. Moreover, when incorporating external data sources (such as new websites containing relevant data of interest), there is a burden to understand the external data semantics as well as a burden to transform the data from those sources into the user's own data repository. Meanwhile, if the user wants to filter the data, she has the burden to understand and specify the selection condition. Finally, when existing sources are updated or additional sources are added, there is a burden to understand these updates and fuse them into her existing data collection.

This dissertation introduces various approaches to help the end-user reduce these burdens in organizing their everyday data. To ease the birthing pain of creating new databases, the dissertation proposes a system with direct manipulation interface and user-friendly operators for the end-user to easily design and evolve her data schema. To facilitate incorporation of external data sources, a sample-driven schema mapping approach is introduced so that the user can freely provide sample instances in her collection and the system will automatically deduce the desired schema mapping from the external data sources to her own repository. In a similar flavor, we propose an approach to facilitate the user in specifying selection conditions via example data points she wants to select. Finally, to help the user incorporate source data updates into her own collection, the dissertation proposes a technique to automatically update the integrated data according to external source change, by conducting efficient incremental information integration.

CHAPTER I

Introduction

With booming information technology nowadays, digital data are permeating every aspect of our daily life. As a result, end-users are increasingly experiencing the necessity of electronically storing, managing and maintaining their personal data. Indeed, end-users are already accustomed to managing their personal data such as contact books and calendars in their electronic devices. Meanwhile, the desire for organizing more information into the computer is expanding. For instance, a scientist may need to regularly manage a great amount of science data on his desktop. Similarly, a web market investigator's may desire to collect data from various websites to build a universal data repository in her daily work.

While end-users are willing to electronically organize a large amount of everyday data, to do this is nontrivial for them. The bottleneck comes largely from the fact that, end-users are lacking the expertise to deal with data schema, which is key to the state-of-the-art data management tasks such as database design, data transformation and data integration. Indeed, dealing with schema in these tasks poses several significant challenges to these non-technical people. First of all, to mentally structure one's everyday data is already difficult for an end-user, let along to abstract this structure to a reasonable database schema design. Second, in the presence of the need of incorporating external data sources into existing data collection, the end-user has to transform the external data into a format that is consistent with the existing collection, by either manually copying and pasting data or establishing complicated schema mappings. Meanwhile, if the user wants to filter the data, she has the burden to abstract and specify the selection condition. Finally, as existing sources are updated and additional sources are added, the end-user has to incrementally update her data collection in order to reflect the changes on the source side. All these tasks introduce a great amount of cognitive and operational burdens for the end-users who have little technical expertise.

1.1 Motivation

1.1.1 Data Schema Design for End-Users

As our daily lives are being increasingly digitalized, non-technical people are discovering the necessity of storing, managing, accessing, and manipulating electronic data. In effect, we are seeing the masses, who lack technical expertise, managing personal data without help from any consultants or DBAs.

Where the users' data collection is a single list, such as a contact book with a list of contacts, most users have no difficulty with the structure. In fact, spreadsheets are still the most common data management application in this case and offer abundant usability. However, a list may not be a universal natural solution. Imagine, one of your friends purchased a new house in a different address. Do you add a new column “second address” just for that friend, or concatenate the new address with the old one in the same cell?

In fact, while the list provides a flat schema, the user’s mental data model may be hierarchical. For instance, one user may like the design of a contact book, where all the information are grouped by person identity. On the other hand, another user may favor an address book, where personal information are grouped by address so that the same family appear under one single address. In these cases, spreadsheets become unsuitable and the users have to specify their own ad-hoc data structure by other means.

This kind of challenge is not only for end-user with their simple data collection, but may also arise from more technical scenarios. Nowadays a lot of scientists have their science data digitalized. However, they are lacking the expertise to organize these data in a professional way. For instance, the biologists using a molecule database may wish to transfer several protein properties from the protein relation to the protein interaction relation for easy later analysis. But even a task as simple as this bottlenecks most biologists who do not know how to design and evolve database schema.

Although there are many data management applications specialized for specific tasks and designed with appropriate schemas for a large class of users, there will always be users who are not completely satisfied with what they can get out of the box and desire something more. Even users who are initially satisfied with an application may wish to enhance or customize it as their requirements change. Indeed, the end-users need to flexibly create and evolve data schema in an ad-hoc manner with a minimal burden placed on them. This is extremely challenging for them given that they are not trained to abstract their mental data model into a reasonable schema, with both performance and extensibility implications.

1.1.2 User-Friendly Data Migration

Sometimes, rather than manually accumulate one's data, an end-user would like to leverage existing data sources. For instance, an online market investigator may need to combine data from various online data sources to generate her own integrated repository. However, those external data sources may not be structured in the same way as the user expects to organize her own collection. Consequently, the user has to map the data transform external sources to her expected integrated format.

A naive approach would be to copy and paste data manually from the external sources to the target collection. Obviously this is too effort-consuming to be practical even for a few hundred data entries. Alternatively, one can employ the well-studied schema mapping technique [14, 61, 63] to establish the data transformation. However, the semantic and structural complexity of a schema mapping is the major barrier to its user-friendly applications under such circumstances.

Although a handful of mapping design systems have been developed, most of these systems are based on a match-driven methodology. This requires the user to have prior knowledge of both the source and target schemas as well as in-depth mapping semantics, which renders it inappropriate for end-user oriented schema mapping tasks. To facilitate the end-user to easily migrate external data into her collection, it is desirable to have a new approach which minimizes the cognitive and operational burden on the user side.

1.1.3 User-Friendly Data Filtering

When the user is either creating a view of his data or transforming the data from one schema to another, it is highly likely that the user is not interested in all pieces of data. For example, the online market investigator may only be interested in several product categories within certain time windows. However, the selection logic may be complicated in itself such that the end-user is not able to directly specify it in formal languages, even if she precisely knows the selection logic.

Although the user may specify the selection criteria via a traditional forms-based interface, the interface may not be sufficient in cases where the complexity of the selection logic the user desires exceeds the expressive power of such an interface. On the other hand, without the aid from an abstract selection logic, the user would have to manually select all desired data points, which is impractical if such data points are many. For instance, suppose the online market investigator is trying to monitor the price of all electronic devices after year 2010 and smart phones after 2012. The state-of-the-art forms-based interface would fall in short since they typically do not support disjunction. On the other hand, there may be thousands of such devices and smart phones, which the user is unlikely to be able to manually label. To enable the end-users to easily specify such kind of selection conditions, we need a selection framework which offers sufficient expressiveness and facilitates complex selection logic specification with as little manual burden as possible.

1.1.4 Incremental Data Collection Maintenance

After external data sources have been integrated, the user has a need to maintain her collection in the face of external source changes. For example, for the online market investigator, if the price information from a certain source website is changed, this update should be promptly reflected in the user's own collection. Moreover, if she want to integrate relative price information from a new website, the user has to seamlessly fuse these additional information into her existing data repository.

In both cases, there is a need to regenerate the integrated data. Naively, the user could re-execute the integration from everything again from scratch. However, that may not only place a heavy operational burden on the user, but also be extremely inefficient, given that the the incremental existing source change or the new source data may be relatively small in size. Moreover, this approach is not an option if the previous source data or the previous integration logic are no longer available. As a result, an incremental approach for integrating the new input into the existing output is in need.

1.2 Contributions and Dissertation Outline

In order to help end-users abstract their mental data model into a reasonable database schema design, this dissertation introduces a concept of "organic schema design", in which the end-users create and refine the structure of their data collection in a natural and casual style. During an organic schema design process, the end-users may freely explore the schema design space with just a little effort and stop at any schema when they are satisfied with the current design.

To enable such organic evolution, this dissertation introduces a next-generation spreadsheet interface, namely "Span Table", to gather possibly complicated everyday information into an integrated hierarchical view. On top of this span table is a carefully designed algebra, which abstracts common schema evolution procedures into primitive operators. Moreover, these operators are implemented with direct manipulation that only involves simple point-and-clicks that imply intuitive semantics.

In the presence of organic evolution, the data collection may be subject to de-normalization. Consequently, user data entry may invoke unnecessary repetition or harm data integrity. We propose a data entry guidance method, which auto-completes duplicated values and reacts to possible input errors by incrementally maintaining a set of functional dependencies.

This dissertation folds these ideas into a novel system called CRIUS. The detailed design of CRIUS and its evaluation can be found in Chapter II.

To render schema mapping democratic, this dissertation proposes a sample-driven approach that enables relatively unsophisticated end-users to easily construct their own data repository from external sources. Using this approach, an end-user may freely provide sample instances in her target data collection, and the system will automatically elicit the mappings that transform the source database into the partially-defined target. As the user provides more and more information, the system returns increasingly better estimates of the desired mapping.

To implement the sample-driven approach, the thesis starts from a single flat target table, and restricts the mapping family to be project-join SQL queries. Under these initial assumptions, the thesis develops an efficient sample search algorithm and shows that it can obtain provably correct results at interactive speeds. Based on this algorithm, the thesis prototypes a sample-driven mapping system, MWeaver, which allows the average user to perform a practical mapping task in about 1/5th the time needed for the traditional match-driven tools. Chapter III presents a detailed description of the design, implementation and evaluation of MWeaver.

While constructing selection logic from scratch and manually labeling all desired data points are both hard, a small set of user-input examples at the beginning of the specification procedure could produce a selection criteria close enough to what the user desires. From there it is much easier for the user to refine the selection logic and derive the desired selection logic.

Based on this observation, in this thesis we propose an approach named example-driven selection condition specification. The approach works in two phases. During the first phase, when the user is not confidant about directly specifying the selection condition from scratch, we ask the user to provide some example data points she wants to see in the output. Based on these examples, we automatically derive an initial selection condition which will select the user-input examples in additional to other data points the user likely desires.

Although the initial selection condition we derived may be close to the user-desired selection condition, it is not guaranteed to be an exact match. While providing more examples may provide better estimate of the desired selection condition, the convergence is not guaranteed and the user may be burdened to exhaustively provide examples. As a result, we propose a second phase, during which the user may revise an expressive representation of the selection condition via an algebra consisting of direct manipulation operators. This work will be discussed in Chapter IV.

In order to enable user-friendly incremental information integration, we propose a two-phase scheme for information integration, and introduce an efficient incremental integration approach based on this two-phase scheme. The highlight of our approach is, as long as the intermediate integration results are maintained, future incremental integration can be executed purely from previous intermediate results and the incremental new data. Even if the previous input data is no longer available or the previous integration logic is missing, we are still able to generate the same results as re-executing everything from scratch, and with a greatly improved performance. This work will be described in Chapter V.

CHAPTER II

CRIUS: User-Friendly Database Design

Non-technical users increasingly find it necessary to add structures to their data. This gives rise to the need for database design. However, traditional database design is deliberate and heavy-weight, requiring technical expertise that everyday users may not possess. For this reason, we propose that non-technical users who manage their everyday data should be able to create and refine data structures in an ad-hoc way over time, thereby “organically" growing their schemas. For this purpose, we develop a spreadsheet-like direct manipulation interface. We show how integrity constraints can still provide value, even in this scenario of frequent schema and data modifications. We also develop a back-end database implementation to support this interface, with a design that permits schema changes at a low cost.

We have folded these ideas into a system, called CRIUS, which supports a nested data model and a graphical user interface. From the user’s perspective, the chief advantages of CRIUS are its support for simple schema definition and modification through an intuitive drag-and-drop interface, as well as its guidance towards user data entry based on incrementally updated data integrity.

We have evaluated CRIUS by means of user studies and performance studies. The user studies indicate that 1) CRIUS makes it much easier for users to design a database, as compared to state-of-the-art GUI database design tools, and 2) CRIUS makes user data entry more efficient and less error-prone. The performance experiments show that 1) the incremental integrity update in CRIUS is very efficient, making the data entry guidance applicable and 2) the back-end database implementation in CRIUS significantly improves the performance of schema update tasks, without a significant impact on other operations.

2.1 Introduction

2.1.1 Motivation

As digital data permeates into our daily lives, non-technical people are increasingly discovering the necessity of storing, managing, accessing, and manipulating electronic data. In effect, we are seeing the masses, who lack technical expertise, managing personal and business data, without help from any consultants or DBAs. For example, many scientists have a great amount of science data digitalized. Similarly, an online market investigator may have collected product information from various websites for later analysis.

Where the data is a single list, most users have no difficulty with the structure. But even with slightly more complex structures, there frequently are choices to be made, with both performance and extensibility implications. In fact, the need for schema specification is a major barrier to database use.

Due to their simplicity, spreadsheets are still the most common data management application used by people without technical training. By design,

Name	City	Address
Orlando	Erie	2251 Elliot
Keith	Erie	3207 Grady

(a) Simple Spreadsheet

City	Name	[Address]
		Address
Erie	Orlando	2251 Elliot
Erie	Keith	3207 Grady
		7943 Walnut

(b) Structured Spreadsheet

Figure 2.1: Example Address Books

spreadsheets implement an extremely simple data model, consisting of a single flat table, with rows and columns. However, due to the increasing complexity of personal and small-business data, users often find it necessary to augment this basic data model with additional ad-hoc structure. Implicitly, these users are defining schemas for their data.

Example II.1. Consider the basic task of maintaining a personal address book. In the simplest case, this can be done using the basic data model of spreadsheets, as shown in Figure 2.1(a). However, it is easy to imagine situations in which the user will require more structure in order to manage her data. For example, suppose that the contact named Keith purchases a second home. In this case, the user is likely to capture this by defining some ad-hoc structure, for example as shown in Figure 2.1(b).

The above example illustrates the challenge of allowing non-technical users to define and evolve their schemas. One might argue that most data management applications will be designed by professionals, who will construct appropriate schemas for a large class of users. Indeed, this may be sufficient in some scenarios. However, there will always be users who are not completely satisfied with what they can get out of the box and desire something more. Even users who are initially satisfied with an application may wish to enhance or customize it as their requirements change. For instance, after the user has made some international friends, she may wish to record the nationality for each of her contacts. Ideally, the database should respond gracefully to changes like this, in a way that places minimal burden on the user.

In this chapter, we study the issue of user-defined structure in data. How can an end user, beginning with a simple spreadsheet, such as an address book with just names and phone numbers, extend this “database" to include more attributes and structure? This sort of organic “schema evolution" may happen because of an intrinsic need to capture some additional information (e.g., nationalities). It will also take place when the user comes upon the need to represent some information that does not quite fit the current structure. For example, when the user realizes that some of her contacts are married couples, she may decide that it makes more sense to store one address per family, rather than one address per person.

Traditional database design is deliberate – there is extensive gathering of requirements, careful analysis, and methodical step-by-step design, all performed by highly trained personnel. In contrast, the database “design” in our scenario is organic – it is not carefully considered, and it is expected to be modified as new data reveals weaknesses in the current design or exposes assumptions that are now violated. If schema modification is possible at a reasonable cost, this sort of organic schema growth is not a problem – rather it becomes the desired style of design. But to accomplish this, schema modification must be rendered easy and cheap.

Based on our experience, a spreadsheet-like nested-table may naturally support such organic schema growth for non-technical users. Observe, for example, the propensity of non-technical users to define complex spreadsheets (e.g., using Microsoft Excel), rather than migrating their data to relational databases, even those that are designed for personal and small-business use (e.g., Microsoft Access). We believe the reason for this is that a typical relational database is too discrete for end-users to manipulate conveniently. Users have to understand table schemas separately and learn the inter-table relationships. In contrast, having the information of interest folded into a single spreadsheet-like structure makes it much easier to comprehend.

While the users may enjoy the freedom of organically growing their schema, we have to be aware that user-defined schemas are subject to denormalization. Consequently, users have to explicitly deal with duplicated data entries, which may easily produce errors violating integrity constraints. In this chapter, we also study how to provide usable guidance towards data entry in such a freestyle environment, by efficiently managing integrity constraints.

2.1.2 Challenges

In summary, our goal is to support organic schema creation and modification using a natural spreadsheet-like interface, while ensuring efficient and effective data entry on this organic schema. To accomplish this, there are multiple challenges to be addressed:

- **Schema Update Specification:** Specifying a schema update as in Example II.1 is challenging with existing tools. For example, using conventional spreadsheet software, it is impossible to arrive at a hierarchical schema such as the one in Figure 2.1(b). Alternatively, using a relational DBMS, one has to manually split the table and set up the cross-table relationships. This is not easy for end-users, even with support from GUI tools. Instead, we would like to support schema creation and modification via a direct manipulation ("point-and-click") interface.

- **Data Migration:** Once a new schema is specified, there is still a critical task of migrating existing data to the new schema. If the schema is simply augmented, this migration may be easy. However, if the schema structure is changed (e.g, from allowing one address per contact to multiple addresses), then one has to introduce a complex mapping in order to "fit" the existing data into the new schema. Even if spreadsheet software supporting hierarchical schema is provided, the user still has to manually copy data in a cell-by-cell manner to perform such mappings. This process is both time-consuming and error-prone.
- **Data Entry:** As a result of the denormalization due to organic schema growth, data entry may become inefficient and error-prone. One may use integrity constraints to assist in data entry. However, since users cannot understand complex integrity constraints, and constraints are also subject to user update, this cannot be done in a naive way. It other words, we have to construct a positive feedback loop between data entry and constraint update in the usability context, with a practical response time.
- **Schema evolution Performance:** Schema evolution is usually a heavyweight operation in traditional systems. IT organizations allow days to execute schema evolutions, since they plan them carefully in advance. However,

everyday users require a swift feedback when updating schema as their plans are immediate and casual. Thus, we need to develop techniques to support quick schema evolution without giving up other desirable features.

2.1.3 Contributions

The primary contribution of this work is the design of a novel system called CRIUS. From the user's perspective, the main benefit of CRIUS lies in the user-friendly interface, which allows non-technical users to create and modify the schemas associated with their data in an organic way. Rather than requiring users to adjust data to the new schema in a cell-by-cell manner, we have designed the UI so that each schema update requires only a single drag-and-drop with the mouse. The details of this interface, addressing the specification challenge, can be found in Section 2.2.1. The design of direct manipulation operators on the interface, addressing the migration challenge, is covered in Section 4.5.

To address the data entry challenge, we have designed a data entry guidance feature, which auto-completes duplicated values and reacts to possible input errors based on functional dependencies (FDs). To make this feature practical, one has to break the performance bottleneck of repeatedly inducing FDs using traditional algorithms. Thus, we propose the first incremental algorithm for FD induction. How CRIUS assists data entry by FDs, and how these FDs are incrementally computed are discussed in Section 2.4.

CRIUS uses a relational database as a natural back-end. Ideally, the storage format should support efficient schema evolution, which has not conventionally been a priority in relational databases. For this reason, we suggest a storage format whereby nested relational tables are recursively, vertically partitioned into flat relations. These implementation issues, as discussed in Section 2.5, address the challenges of data migration and performance.

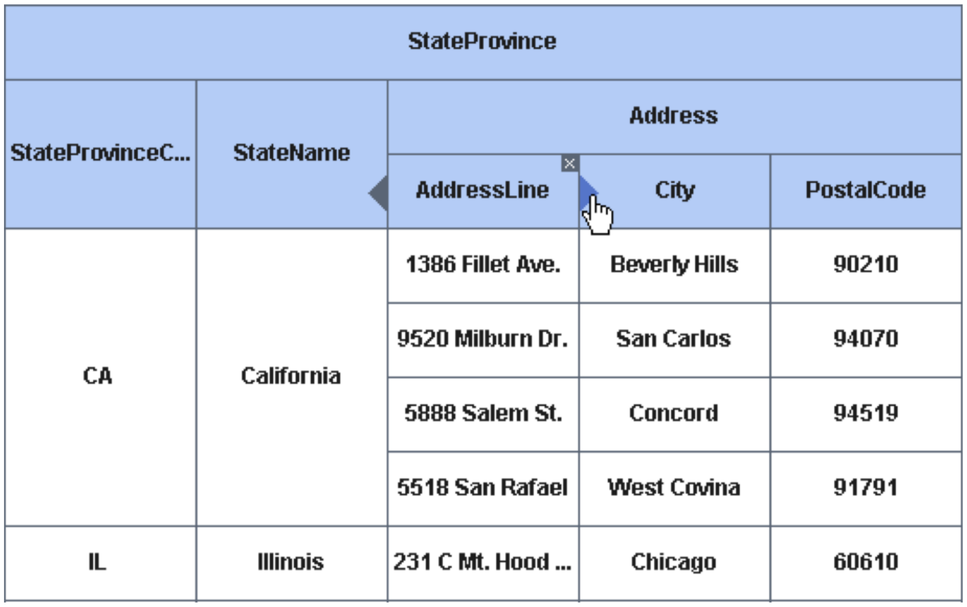
Finally, in Section 5.8, we describe the evaluation of our prototype system. Through user studies, we have found that schema development is much easier in CRIUS than using a GUI tool provided by Microsoft SQL Server Management Studio 2008. The study also shows that the integrity-based guidance feature does reduce typing effort and input errors. Our experiments demonstrate that our incremental FD induction algorithm is much faster than traditional approaches, and the schema modification and data reconstruction performance of CRIUS are reasonable.

2.2 CRIUS Design

2.2.1 Interface Design

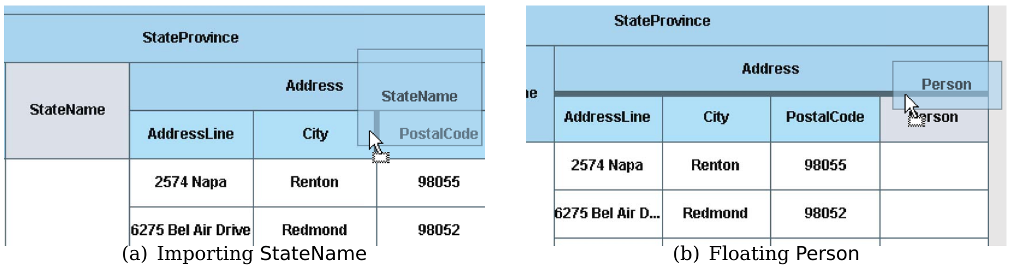
The presentation layer of our system is based on a next-generation spreadsheet, as described in the introduction. Information from multiple related "tables" is combined to present a cohesive nested representation, as shown in Figure 2.2.

Conventional spreadsheets are not designed to handle complex schemas. If a user wants to add more structure to her data, this must be done in an ad-hoc way. Worse, spreadsheets only support cell-by-cell data modifications. As a result, if the user wishes to modify her schema, this is a complex and error-prone process that often consists of cell-, row-, and column- level copy and paste operations.



StateProvince				
StateProvinceC...	StateName	Address		
		AddressLine	City	PostalCode
CA	California	1386 Fillet Ave.	Beverly Hills	90210
		9520 Milburn Dr.	San Carlos	94070
		5888 Salem St.	Concord	94519
		5518 San Rafael	West Covina	91791
IL	Illinois	231 C Mt. Hood ...	Chicago	60610

Figure 2.2: Screenshot of CRIUS



StateProvince			
StateName	Address		
	AddressLine	City	PostalCode
	2574 Napa	Renton	98055
	6275 Bel Air Drive	Redmond	98052

(a) Importing StateName

StateProvince				
ie	Address			
	AddressLine	City	PostalCode	Person
	2574 Napa	Renton	98055	
	6275 Bel Air D...	Redmond	98052	

(b) Floating Person

Figure 2.3: Screenshot of Schema Evolution in CRIUS

In contrast, the CRIUS user interface supports easy schema creation and modification through a simple drag-and-drop interface, as shown in Figure 2.2. The shaded region at the top of the screen is the schema header, and the region below it displays the data body. The user can modify the schema by simply dragging a cell in the schema header using the mouse. For example, Figure 2.3(a) shows a screenshot of a user dragging an attribute StateName inward, and making it part of the sub-relation Address. (We refer to this operation as an IMPORT.) Conversely, the user can EXPORT StateName back to the

StateProvince relation in a similar way.

The user can also create new sub-relations by dragging attributes up and down. Figure 2.3(b) shows an example where Person is dragged up to insert a new intermediate level with only itself. Similarly, one can also drag it down, nesting it to a new sub-relation. (We refer to these operations as FLOAT and SINK, respectively.)

2.2.2 Operator Design

We refer to an instance of the spreadsheet in CRIUS as a *span table*. The UI allows users to restructure the span table schema using drag-and-drop direct manipulations (e.g., IMPORT, EXPORT, FLOAT, and SINK, as described above), and to augment/diminish schema using point-and-clicks (adding/dropping columns). It also supports data manipulation operations (inserting/deleting tuples, updating cells). Collectively, we will refer to this set of operators as the *span table algebra*. (A brief introduction to the span table operators can be found in Appendix 2.3.) In this section, we introduce our key schema update operators in the span table algebra, and compare the expressive power of the algebra to the nested relational algebra.

IMPORT and EXPORT

Example II.2. Suppose the user has an address book span table as shown in Figure 2.4(a), and wishes to associate Zipcode with Address rather than City. The CRIUS UI enables the user to do this by simply pressing the mouse on Zipcode , and dragging it onto [Address]. The system then needs to execute the schema update and transform the nested relation to the one shown

City	Zipcode	[Person]	
		Name	[Address]
			Address
Detroit	48205	Peter	1023 Westwood Ave
Erie	48109	Orlando	2251 Elliot Avenue
Erie	48105	Keith	3207 S Grady Way
			7943 Walnut Ave

(a) Address Book Before Evolution

City	[Person]		
	Name	[Address]	
		Zipcode	Address
Detroit	Peter	48205	1023 Westwood Ave
Erie	Orlando	48109	2251 Elliot Avenue
	Keith	48105	3207 S Grady Way
		48105	7943 Walnut Ave

(b) Address Book After Evolution

Figure 2.4: Address Book With Multiple Levels

in 2.4(b). Readers familiar with nested relational algebra may consider an implementation consisting of a series of nest and unnest: 1) unnest [Person], 2) unnest [Address], 3) nest Zipcode, Name and Address, and 4) nest Zipcode and Address. However, this would introduce a large amount of unnecessary computation moving data from unrelated columns (e.g., Name and Address). Moreover, this series of operations does not semantically conform to the user's intention of the simple drag-and-drop manipulation.

To overcome the problems with the traditional nested algebra in our scenario, we introduce two new schema modification operators, namely *IMPORT* and *EXPORT*. We set up some notation and then define the basic IMPORT and EXPORT operators.

A nested relation N has schema expression $S(N)$ and schema tree $Tree(N)$. $S(N)$ is the flattened version of $Tree(N)$ by a postorder traversal. For example, $S(N)$ for Fig. 2.4(a) is {City, Zipcode, {Name, {Address}}}. $t[\overline{X}]$ is a tuple projection, where t is a nested relational tuple and $\overline{X}$ is a list of attributes. $\sqcup$

is the relation union operation used in [87].

The IMPORT operator imports an atomic attribute into a nested relation. Intuitively, it pushes down the atomic attribute to become a “child” of a sibling group.

Definition II.3 (Basic Import Operator)**.** Given a nested relation N with $S(N) = \{\overline{AX}P\{\overline{Q}\}\}$, where $\overline{A}$ denotes a list of atomic attributes, $\overline{X}$ denotes a list of relation-valued attributes, P denotes an atomic attribute being transported and $\{\overline{Q}\}$ denotes the target relation-valued attribute, which consists of a list of both atomic and relation-valued attributes, $IMPORT_{P,\{\overline{Q}\}}(N) = N'$, where $S(N') = \{\overline{AX}\{P, \overline{Q}\}\}$ and N' is the set of all t' for which there exists $t \in N$, such that:

(1) $t'[\overline{A}] = t[\overline{A}]$

(2) $\forall X$ in list $\overline{X}, t'[X] = \bigsqcup\{t''[X] | t'' \in N \wedge t''[\overline{A}] = t[\overline{A}]\}$

(3) $t'[\{P, \overline{Q}\}] = \{t'' | \exists t''' \in N,\ s.t.\ t''[P] = t'''[P] \wedge t''[\overline{Q}] \in t'''[\{\overline{Q}\}] \wedge t''[\overline{A}] = t'''[\overline{A}]\}$

According to this definition, the manipulation in Example II.2 can be executed in two imports: 1) $IMPORT_{Zipcode,\{Name,\{Address\}\}}$ and 2) $IMPORT_{Zipcode,\{Address\}}$.

The EXPORT operator is the inverse of IMPORT. It raises an atomic attribute from a deeper nested level to a shallower nested level and naturally maps existing data to the new schema.

Definition II.4 (Basic Export Operator)**.** Given a nested relation N with $S(N) = \{\overline{AX}\{P, \overline{Q}\}\}$, where $\overline{A}$ denotes a list of atomic attributes, $\overline{X}$ denotes a list of relation-valued attributes, and $\{P, \overline{Q}\}\}$ denotes the source relation from which the atomic attribute P will be extracted and inserted into the target relation

$S(N)$, $EXPORT_P(N) = N'$, where $S(N') = \{\overline{A}\overline{X}P\{\overline{Q}\}\}$ and N' is the set of all t' for which there exists $t \in N$, such that:

(1) $t'[\overline{A}] = t[\overline{A}]$

(2) $t'[\overline{X}] = t[\overline{X}]$

(3) $\exists t'' \in t[\{P, \overline{Q}\}] s.t. t'[P] = t''[P]$

(4) $t'[\{\overline{Q}\}] = \{t''[\overline{Q}] | t'' \in t[\{P, \overline{Q}\}] \wedge t''[P] = t'[P]\}$

For instance, executing $EXPORT_{Zipcode}$ twice will bring the span table in 2.4(b) back to the span table in 2.4(a).

In practice, our algebra extends IMPORT/EXPORT to span multiple schema levels, which allows the user to import Zipcode from the root to Address in one step. The algebra also includes schema update operators to create new sub-relations (SINK/FLOAT) together with other data manipulation operators. We will detail the span table algebra in the following section.

2.3 Span Table Algebra

In this section, we define the operators in the span table algebra. An informal description is in Table 2.1. The span table algebra differs from the nested relational algebra in the following aspects. First, each span table operator is naturally and directly supported by the UI, while the nested algebra is hard to be implemented using direct manipulation. Second, the nested algebra is designed mainly for structural query, while the span table algebra aims to perform schema evolution. For instance, columns can be added/dropped in the span table algebra, but not in the nested algebra. Finally, data is static under the nested algebra, but can be updated using span table operators. Such update is non-trivial since it may change the table structure and affect further schema evolution.

Operators	Description
Import(A)	Move A inward into a descendant-relation.
Export(A)	Move A outward into an ancestor-relation.
Sink(A)	Push A to create a new leaf relation.
Float(A)	Lift A to create a new intermediate relation.
Add/Drop(A)	Add/drop attribute A.
Insert/Remove/Update(T)	Insert/remove/update a tuple T.

Table 2.1: Span Table Operators

2.3.1 Basics

We will assume the reader is familiar with the basic concepts of nested relational algebra (e.g., nest and unnest) [26, 87, 97, 88]. In CRIUS, we chose to adopt partitioned normal form (PNF) [87], which asserts functional dependencies from the set of all the atomic attributes at each schema level, since it is consistent with our need to preserve data integrity. We also adopt the schema tree described in [97]. To summarize, each node in the schema tree represents either an atomic attribute or a relation-valued attribute in the corresponding nested relation. An edge from one node to another indicates that the parent node contains the attribute represented by the child node. For example, the schema updating process in Figures 2.4(a) and 2.4(b) can be expressed by a schema tree evolution shown in Figure 2.5. We also define the *schema level* of a certain relation to be the depth of its corresponding node in the schema tree, with zero for the root.

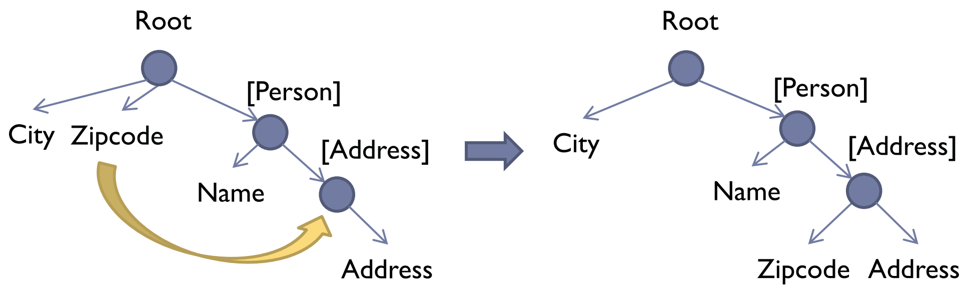


Figure 2.5: Address Book Schema Evolution

2.3.2 Schema Update Operators

The span table algebra has four schema restructuring operators: IMPORT, EXPORT, SINK, and FLOAT. Each operator not only updates the schema but also maps existing data to the updated schema.

- **IMPORT:** The basic IMPORT, as defined in Section 4.5, can only take place at the root of a schema tree and can only push attributes down to a schema level that is exactly one level deeper. These limitations must be lifted if we are to support drag-and-drop driven organic schema modification. In the following, we upgrade the basic IMPORT so that it is able to reach an arbitrarily deep schema level starting from anywhere in the schema tree. We denote the parent of a given attribute Q by $Parent(Q)$. $Parent(Root)$ is null. We denote the ancestor and descendent of Q by $Ancestor(Q)$ and $Descendent(Q)$, respectively. Both $Ancestor(Root)$ and $Descendent(Leaf)$ are nulls. If $T \in Descendent(L)$, we define $Path(L, T)$ to be the ordered list of schema nodes from L to T, excluding L and including T. Then, the full IMPORT can be defined as below.

  Definition II.5 (Import Operator)**.** Assume N is a nest-ed relation with schema tree $Tree(N)$. Let $L = \{\overline{A}, \overline{X}, P\}$ and T be the schema nodes in $Tree(N)$ such that $T \in Descendent(L)$ and T is relation-valued, then $IMPORT_{P|L,T}(N) = N'$, wh-ere $S(N')$ differs from $S(N)$ by removing P from L and inserting it to T. Also, N' is the result relation of the following operations: for each Q on $Path(L, T)$ following the order, on each sub relation with schema $Parent(Q)$ in N, execute $IMPORT_{P,Q}$.

In practice, by observing some transitive features of the upgraded IMPORT sequence, we can avoid repeated copying and merging, and thereby implement it cheaply compared to a literal implementation of Definition II.5.

- **EXPORT:** The basic EXPORT has been defined in Section 4.5. Now we extend it to its full version similar to IMPORT. Due to space, we omit the full definition and illustrate IMPORT and EXPORT by the following example.

  Example II.6. Consider an address book in Figure 2.6(a). The user may call EXPORT on State[1] to augment categorizing information by state name. The resulting relation is pictured in Figure 2.6(b), with the value of "Keith" duplicated. Finally, the user may decide to categorize data only by State. So she calls IMPORT on Name[2], resulting in the final address book in Figure 2.6(c), with the two "MI"s merged.

- **SINK:** IMPORT and EXPORT serve to transport atomic attributes across different levels of the schema tree. However, they do not create new schema levels, as is normally done by NEST in traditional nested algebra. A traditional NEST operator combines multiple attributes into a nested relation. This is difficult to do in a single drag-and-drop. A multi-step operation, with

[1] $EXPORT_{State|\{State,Address\},Name}$
[2] $IMPORT_{Name|\{Name,State,\{Address\}\},\{Address\}}$

Name	[Address]	
	State	Address
Orlando	MI	2251 Elliot Avenue
Keith	MI	3207 S Grady Way
	OH	7943 Walnut Ave

(a) Categorized by Name

Name	State	[Address]
		Address
Orlando	MI	2251 Elliot Avenue
Keith	MI	3207 S Grady Way
Keith	OH	7943 Walnut Ave

(b) Categorized by Name and State

State	[Address]	
	Name	Address
MI	Orlando	2251 Elliot Avenue
	Keith	3207 S Grady Way
OH	Keith	7943 Walnut Ave

(c) Categorized by State

Figure 2.6: Categorizing an Address Book

selecting attributes first and then nesting and moving them, is much less user-friendly. To solve this problem, we restrict the NEST operator to a s-ingle attribute, and name it SINK to distinguish it from NEST. SINK is also defined from the basic version.

Definition II.7 (Basic Sink Operator)**.** Given a nested relation N with $S(N) = \{\overline{A}\overline{X}P\}$, where $\overline{A}$ and $\overline{X}$ have the same meaning as before, and P denotes the atomic attribute to be nested, $SINK_P(N) = N'$, where $S(N') = \{\overline{A}\overline{X}\{P\}\}$ and N' is the set of all t' for which there exists $t \in N$, such that:

(1) $t'[\overline{A}] = t[\overline{A}]$

(2) $t'[\overline{X}] = \bigsqcup\{t''[X]|t'' \in N \wedge t''[\overline{A}] = t[\overline{A}]\}$

(3) $t'[\{P\}] = \{t''[P]|t'' \in N \wedge t''[\overline{A}] = t[\overline{A}]\}$

Definition II.8 (Sink Operator)**.** Assume N is a nested relation with schema tree $Tree(N)$. Let $L = \{\overline{A}, \overline{X}, P\}$ be a certain schema node in $Tree(N)$, then $SINK_{P|L}(N) = N'$, where $S(N')$ differs from $S(N)$ by replacing L with $\{\overline{A}, \overline{X}, \{P\}\}$. Also, N' is the result relation after executing $SINK_P$ on all sub relations with schema L within N.

- **FLOAT:** Theoretically, IMPORT, EXPORT and SINK form an orthogonal and complete (with respect to NEST and UNNEST in nested relational algebra) set of operators. However, in some cases, the users may wish to create a new schema level by sinking most of the attributes on the current level. For the sake of usability, we propose another operator, namely FLOAT, to complement SINK. FLOAT relatively lifts an atomic attribute up by sinking all its siblings by one level.

  Definition II.9 (Basic Float Operator)**.** Given a nested relation N with $S(N) = \{P\overline{Q}\}$, where $\overline{Q}$ is a set of attributes, and P denotes the atomic attribute to be floated, $FLOAT_P(N) = N'$, where $S(N') = \{P\{\overline{Q}\}\}$ and N' is the set of all t' for which there exists $t \in N$, such that:

  (1) $t'[P] = t[P]$

  (2) $t'[\{\overline{Q}\}] = \{t''[\overline{Q}] | t'' \in N \wedge t''[P] = t[P]\}$

  Definition II.10 (Float Operator)**.** Assume N is a nested relation with schema tree $Tree(N)$. Let $L = \{\overline{A}, \overline{X}, P\}$ be a certain schema node in $Tree(N)$, then $FLOAT_{P|L}(N) = N'$, where $S(N')$ differs from $S(N)$ by replacing L with $\{P, \overline{X}, \{\overline{A}\}\}$. Also, N' is the result relation after executing $FLOAT_P$ on all sub relations with schema L within N.

  Note that FLOAT and SINK can be applied at any nesting level but move the affected attribute by only one level. We do not extend to multi-level as we did for IMPORT and EXPORT because multi-level FLOAT/SINK can result in an intermediate sub relation with no atomic child, which violates PNF.

In addition to the four schema restructuring operators, the algebra also includes operators to augment/diminish schema, such as adding/dropping/permuting columns. All these operators are restricted to a single schema level and behave identically to their flat versions. In the interest of space, we skip their formal definitions.

2.3.3 Data Manipulation Operators

Besides the schema update operators, three data manipulation operators, namely INSERT/DELETE/UPDATE, are also provided with the algebra. These are consistent with their traditional semantics except that i) all of them are extended to the nested scenario so that manipulating data at arbitrary schema level becomes feasible, and ii) insertion and deletion trivially guarantee foreign key constraints in a cascading manner.

2.3.4 Expressive Power Analysis

Though nested relational algebra does not naturally lend itself to supporting a direct manipulation interface, we show that the span table algebra and nested relational algebra are actually equivalent in expressive power given a fixed set of universal attribute (recall that nested relational algebra cannot add/drop attributes). To do this, all we have to show is that NEST and UNNEST can both be expressed using the span table algebra, and vice versa. For simplicity, we will restrict IMPORT and EXPORT to their basic versions in lemmas II.14 and II.15. The general case follows directly by induction.

The IMPORT and EXPORT operators can be expressed in terms of NEST and UNNEST, as given by the following two lemmas. The SINK and FLOAT operators are just restricted versions of NEST.

Lemma II.11. *Let N be a relation with $S(N) = \{\overline{AX}P\{\overline{Q}\}\}$, then $IMPORT_{P,\{\overline{Q}\}}(N) = NEST_{P,\{\overline{Q}\}} \cdot UNNEST_{\{\overline{Q}\}}(N)$.*

Lemma II.12. *Let N be a relation with $S(N) = \{\overline{AX}\{P, \overline{Q}\}\}$, then $EXPORT_P(N) = NEST_{\{\overline{Q}\}} \cdot UNNEST_{P,\{\overline{Q}\}}(N)$.*

Also, we have the following theorem.

Theorem II.13. *Any NEST or UNNEST can be expressed as a sequence of span-algebra schema update operations.*

Thus, we have not sacrificed expressiveness for usability; anything that can be expressed using nested relational algebra can also be captured using the direct manipulations supported by CRIUS. The proofs of the lemmas and the theorem can be found in the following section.

2.3.5 Proofs

Lemma II.14. *Let N be a nested relation with $S(N) = \{\overline{AX}P\{\overline{Q}\}\}$, then $IMPORT_{P,\{\overline{Q}\}}(N) = NEST_{P,\{\overline{Q}\}} \cdot UNNEST_{\{\overline{Q}\}}(N)$.*

Proof. Let N be a nested relation with $S(N) = \{\overline{AX}P\{\overline{Q}\}\}$, $N' = UNNEST_{\{\overline{Q}\}}(N)$ and $N'' = NEST_{P,\{\overline{Q}\}}(N')$. According to definition,

$$\begin{aligned} UNNEST_{\{\overline{Q}\}}(N) = \{t' | \exists t_{t'} \in N \ s.t. \ t'[\overline{A}] = t_{t'}[\overline{A}] \\ \wedge t'[P] = t_{t'}[P] \wedge t'[\overline{X}] = t_{t'}[\overline{X}] \wedge t'[\overline{Q}] \in t_{t'}[\{\overline{Q}\}]\} \end{aligned} \quad (2.1)$$

where $t_{t'}$ is the tuple in N with the same values at $\overline{A} \cup P$ as t' (and is thus unique).

$$NEST_{P,\{\overline{Q}\}}(N') = \{t''|\exists t' \in N' \; s.t. \; t''[\overline{A}] = t'[\overline{A}]$$
$$\wedge\, t''[\overline{X}] = \bigsqcup\{t'[\overline{X}]|t'[\overline{A}] = t''[\overline{A}]\}$$
$$\wedge\, t''[\{P,\overline{Q}\}] = \{t'[P,\overline{Q}]|t'[\overline{A}] = t''[\overline{A}]\}\} \quad (2.2)$$

Since for each $t' \in N'$, we have a $t_{t'} \in N$, $\exists t' \in N'$ always implies $\exists t_{t'} \in N$. Furthermore, according to the first line in equation (1), $t'[\overline{A}] = t_{t'}[\overline{A}]$. Thus, the first line in equation (2) can be translated to $\exists t_{t'} \in N \; s.t. \; t''[\overline{A}] = t_{t'}[\overline{A}]$. Moreover, we define $E_{\overline{V},W}$ to be the set of t' which map to the same $t_{t'}$, with $t'[\overline{A}] = \overline{V} \wedge t'[P] = W$. Then, $\{t'|t'[\overline{A}] = t''[\overline{A}]\}$ can be classified into a group of $E_{t''[\overline{A}],W}$, each of which corresponds to a $t_{t'} \in N$ with $t_{t'}[P] = W$. We denote this group by $R_{t''[\overline{A}]}$. Now we can rewrite $\bigsqcup\{t'[\overline{X}]|t'[\overline{A}] = t''[\overline{A}]\}$ as $\bigsqcup_{R_{t''[\overline{A}]}}\{\bigsqcup\{t'[\overline{X}]|t' \in E_{t''[\overline{A}],t''[P]}\}\}$. According to (1), $t'[\overline{X}]$ are the same as long as their values at $\overline{A}$ and P are the same, which is the same as that of their corresponding $t_{t'}$. Thus, $\bigsqcup\{t'[\overline{X}]|t' \in E_{t''[\overline{A}],t''[P]}\} = t_{t'}[\overline{X}]$. So the second line in (2) can be translated to $t''[\overline{X}] = \bigsqcup_{R_{t''[\overline{A}]}}\{t_{t'}[\overline{X}]\} = \bigsqcup\{t_{t'}[\overline{X}]|t_{t'}[\overline{A}] = t''[\overline{A}]\}$. Similarly, $\{t'[P,\overline{Q}]|t'[\overline{A}] = t''[\overline{A}]\} = \bigcup_{R_{t''[\overline{A}]}}\{t'[P,\overline{Q}]|t' \in E_{t''[\overline{A}],t''[P]}\}$. According to (1), $t'[P] = t_{t'}[P]$ and $t'[\overline{Q}] \in t_{t'}[\{\overline{Q}\}]$ for $t_{t'}$ corresponding to each $E_{t''[\overline{A}],t''[P]}$. So, $\bigcup_{R_{t''[\overline{A}]}}\{t'[P,\overline{Q}]|t' \in E_{t''[\overline{A}],t''[P]}\} = \bigcup_{R_{t''[\overline{A}]}}\{t'[P,\overline{Q}]|t'[P] = t_{t'}[P] \wedge t'[\overline{Q}] \in t_{t'}[\{\overline{Q}\}]\}$. In other words, the third line in (3) can be translated to $t''[\{P,\overline{Q}\}] = \{t'|\exists t''' \in N, \; s.t. \; t'[P] = t'''[P] \wedge t'[\overline{Q}] \in t'''[\{\overline{Q}\}] \wedge t'[\overline{A}] = t'''[\overline{A}]\}$. In all, by substituting (1) into (2), we obtain the following equation:

$$
\begin{aligned}
&NEST_{P,\{\overline{Q}\}} \cdot UNNEST_{\{\overline{Q}\}}(N) \\
&\qquad = \{t'' | \exists t_{t'} \in N \ s.t. \ t''[\overline{A}] = t_{t'}[\overline{A}] \\
&\qquad \wedge t''[\overline{X}] = \bigsqcup \{t_{t'}[\overline{X}] | t_{t'}[\overline{A}] = t''[\overline{A}]\} \\
&\qquad \wedge t''[\{P, \overline{Q}\}] = \{t' | \exists t''' \in N, \ s.t. \ t'[P] = t'''[P] \\
&\qquad \wedge t'[\overline{Q}] \in t'''[\{\overline{Q}\}] \wedge t'[\overline{A}] = t'''[\overline{A}]\}\}
\end{aligned} \tag{2.3}
$$

Which is identical to the definition of IMPORT. □

Lemma II.15. *Let N be a nested relation with $S(N) = \{\overline{AX}\{P, \overline{Q}\}\}$, then $EXPORT_P(N) = NEST_{\{\overline{Q}\}} \cdot UNNEST_{P,\{\overline{Q}\}}(N)$.*

The proof is similar to the proof of Lemma II.14.

Theorem II.16. *Any NEST or UNNEST can be expressed as a sequence of span-algebra schema update operations.*

Proof. Let N be a nested relation with schema $S(N) = \{\overline{Q}, \{A_1, A_2, ..., A_n\}\}$, and denote $G_i = A_i, A_{i+1}, ...A_n$, then according to Lemma II.15:

$$
\begin{aligned}
&UNNEST_{\{A_1,A_2,...,A_n\}}(N) = UNNEST_{G_n} \cdot \\
&\qquad NEST_{G_n} \cdot UNNEST_{G_{n-1}} \cdot ... \cdot NESTG_3 \cdot \\
&\qquad UNNEST_{G_2} \cdot NESTG_2 \cdot UNNEST_{G_1}(N) \\
&\qquad = EXPORT_{A_n} \cdot EXPORT_{A_{n-1}} \cdot ... \\
&\qquad \cdot EXPORT_{A_2} \cdot EXPORT_{A_1}(N)
\end{aligned} \tag{2.4}
$$

Similarly, for NEST, suppose the nested relation being operated on is N with schema $S(N) = \{\overline{Q}, A_1, A_2, ..., A_n\}$, using the same notation for G, we have:

$$\begin{aligned} & NEST_{A_1,A_2,...,A_n}(N) = \\ & \qquad NESTG_1 \cdot UNNEST_{G_2} \cdot ... \cdot NEST_{G_{n-2}} \cdot \\ & \qquad UNNEST_{G_{n-1}} \cdot NESTG_{n-1} \cdot UNNEST_{G_n}(N) \cdot SINK_{G_n}(N) \\ & \qquad = IMPORT_{A_1} \cdot IMPORT_{A_2} \cdot ... \\ & \qquad \cdot IMPORT_{A_{n-2}} \cdot IMPORT_{A_{n-1}}(N) \cdot SINK_{A_n}(N) \end{aligned} \tag{2.5}$$

□

2.4 Integrity-Based Guidance

Unlike conventional databases where data is carefully normalized according to integrity constraints at design time, in our environment, non-technical users are not able to normalize their data. This increases the user burden to enter duplicated data, as well as the risk of erroneous data entry. To address this problem, CRIUS provides an important set of features that we call *integrity-based guidance*. The basic idea is to induce from the data and maintain a set of “soft” functional dependencies.

These “soft” FDs are then used in two ways to assist in data entry. The first is *inductive completion*, which auto-completes the determined attribute (the righthand side) of an FD according to existing data. For instance, suppose we have inferred the FD $Name \rightarrow Grade$ in Table 2.2. If the user were to enter a new row for $Peter$, CRIUS would automatically suggest a grade A. In addition, CRIUS uses these FDs for *error prevention*, by warning about possible data entry errors. Following the above example, suppose the user has updated Leo's grade in row 3 from A to B. CRIUS will prompt the user that the update may be a mistake since it violates the inferred FDs. The user may decide whether to undo it or not. In case the user commits the update, CRIUS further asks the user if he wants to: (i) also update the other Leo's grade in row 4 to preserve the FD or (ii) force the update. Option (ii) indicates that the previously induced FD $Name \rightarrow Grade$ is an artifact of the database instance and must be updated.

Since user updates may continuously invalidate induced FDs in our environment, we need to frequently update them. While there is a large body of literature on FD induction [51, 67, 79, 102], past solutions are too expensive to be adopted in our scenario where the need to update FD is frequent. To reduce performance cost, CRIUS instead *incrementally maintains* these FDs.

The incremental maintenance can be challenging. For example, if an FD (e.g., $Name \rightarrow Grade$ in Table 2.2) has been invalidated by a user update (e.g., the grade in tuple 2 from A to B), it is not enough to remove that FD from the minimal set, since one or more weaker FDs (e.g., $Name, Course \rightarrow Grade$) may still hold. These weaker FDs would not have been previously recorded in the minimal set because they were dominated by the now violated FD. Similarly, after a certain update, an FD may be dominated by some stronger FDs, and become no longer minimal.

In this chapter, we develop the IFDI (Incremental FD Induction) algorithm. To the best of our knowledge, this is the first algorithm to induce FDs incrementally upon value updates. The algorithm works by maintaining an in-memory lattice which contains all the information for FD induction, and incrementally

ID	Name	Course	Grade
1	Peter	Math	A
2	Peter	Physics	A
3	Leo	Math	B
4	Leo	Physics	B
5	Jack	Math	A

Table 2.2: Student Records

updating part of the lattice. The algorithm differs from existing FD induction algorithms in three ways: 1) Unlike traditional algorithms which fetch data from the database, IFDI only accesses the database once. Subsequent updates can be efficiently processed using the in-memory lattice; 2) IFDI needs to access at most half of the lattice nodes; 3) On average, IFDI only needs to update a very small portion of each lattice node. In this section, we describe the initialization and maintenance phases of the IFDI algorithm separately. In Section 5.8, we show that the cost of IFDI is significantly smaller than naive approaches.

We also briefly describe how we extend IFDI to the nested scenarios and prove that all the nested FDs are preserved under schema evolution when an appropriate representation is chosen and transformed. Our user study in Section 5.8 shows that the guidance feature based on these induced nested FDs does improve usability.

2.4.1 Inducing Initial FDs

Before the incremental maintenance, we first construct a set of important data structures and induce the initial set of FDs.

For flat relations, there is a body of literature on minimal FD induction [51, 67, 79]. As a starting point, we adopt the ideas of *attribute partition* and *at-*

tribute lattice introduced in [51]. An attribute partition on a set of attributes X, denoted by Π_X, is a set of *partition groups*, where each group contains all the tuples sharing the same values at X. For instance, in Table 2.2, $\Pi_{\{Name\}} = \Pi_{\{Name,Grade\}} = \{\{1,2\},\{3,4\},\{5\}\}$. An FD $X \rightarrow Y$ holds iff $\Pi_X = \Pi_{X \cup Y}$ [51, 102]. (For instance, $Name \rightarrow Grade$ holds since $\Pi_{\{Name\}} = \Pi_{\{Name,Grade\}}$.) We strip partitions by omitting partition groups of size one for simplicity (For instance, $\Pi_{\{Name\}} = \{\{1,2\},\{3,4\}\}$). An attribute lattice is a lattice in which each node corresponds to an attribute set and each edge represents a possible FD. An edge goes from node X to node Y iff Y contains X and exactly one more attribute. The attribute lattice for Table 2.2 is shown in Figure 4.3. (For simplicity, hereafter we abbreviate Name by N, Course by C and Grade by G.) The following example demonstrates the initialization phase of the algorithm.

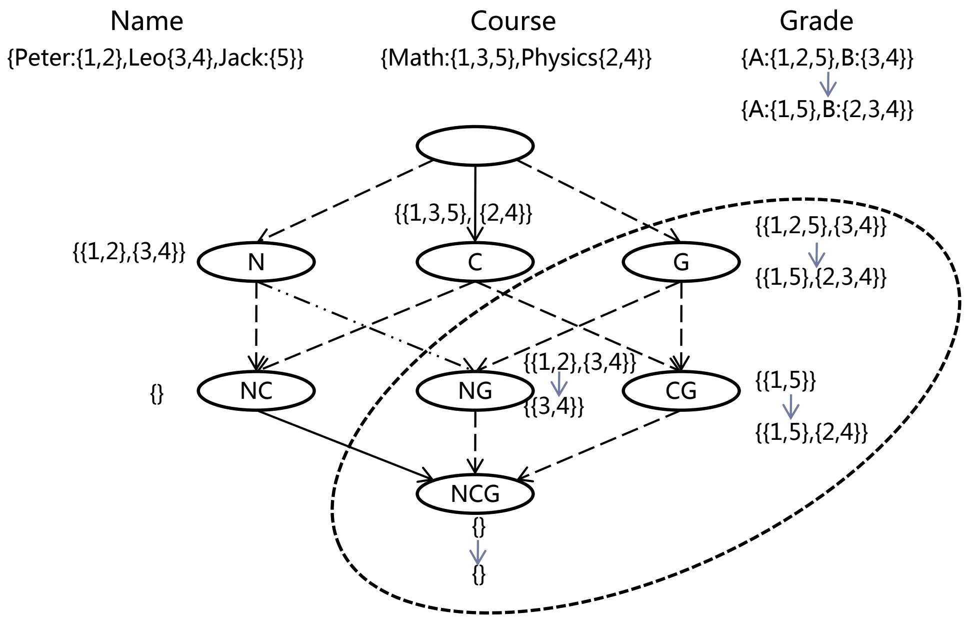


Figure 2.7: The Set Attribute Lattice of Table 2.2 in Example II.17, and its evolution (as depicted by the arrows) in Example II.18

Example II.17. The algorithm reads the metadata and generates the stripped partitions for each single attribute. The stripped partitions are then fed into the first level of the lattice (shown at the top in Figure 4.3). The algorithm processes the lattice in a top-down manner level by level, generating the child partition by taking the product [51] of any two parent partitions (For instance. $\Pi_C \cdot \Pi_G = \Pi_{CG}$). The algorithm outputs an FD if its parent and child partitions are identical. For Table 2.2, only two FDs are discovered: $Name \rightarrow Grade$ (the dash and dot line) and $Name, Course \rightarrow Grade$ (the solid line). However, since the latter is dominated by the former, it is pruned by the algorithm.

2.4.2 Maintaining FDs on Value Update

Once the lattice and the partitions for each node are constructed, the maintenance phase of IFDI is performed for each value update. IFDI checks for FD updates by traversing the lattice and comparing the partitions, in the same way as the existing algorithms. However, IFDI effectively reduces the cost of updating partitions and the number of lattice nodes that must be visited.

Efficient Partition Update: The partition product operation in the traditional FD induction algorithm is a performance bottleneck since it executes a linear scan on the partition, whose size is proportional to the number of tuples in the relation. However, when updating a partition in the incremental case, one does not need to visit most partition groups that are irrelevant to the updated cell.

Example II.18. After the Grade of Peter in row 2 in Table 2.2 is updated from A to B, the lattice evolves as shown in Figure 4.3. One may use the traditional algorithm to compute the new Π_{CG} from Π_C and the new Π_G: for each group in Π_G, assign each tuple in it to the groups in Π_C and then collect them group-wise as the new groups of Π_{CG}. Specifically, for $\{1,5\}$ in the new Π_G, 1 and 5 are assigned to the same group $\{1,3,5\}$ of Π_C. This group thus generates only one new group in Π_{CG}: $\{1,5\}$. However, for $\{2,3,4\}$ in Π_G, 3 is assigned to the first group in Π_C, while 2,4 are assigned to the second. Thus this group generates two new groups: $\{3\}$ and $\{2,4\}$. Collectively, the new Π_{CG} becomes $\{1,5\},\{2,4\}$ (with $\{3\}$ stripped).

IFDI handles the update of Π_{CG} in an incremental way. Instead of updating every group in Π_{CG}, IFDI only focuses on two groups: the group to which tuple 2 previously belonged, and the group to which it will belong after the update. IFDI removes the updated tuple from the old group and adds it into the new group, with the other groups unchanged. In the example, $\{2\}$ is a singleton in the old Π_{CG}, so there is no need for removal. When assigning tuple 2 to its new group, IFDI first retrieves the group containing 2 from Π_C ({2,4} in this case), and then scans the group for any other tuple that has the same value on the updated attribute as the modified cell (tuple 4 is returned here since $\{2,4\}$ is in Π_C and tuple 4 also has a grade of B). IFDI then retrieves the group containing tuple 4 from Π_{CG} (the stripped singleton $\{4\}$) and adds tuple 2 to that group (forming a new group $\{2,4\}$). Note that we have finished updating Π_{CG} without even touching group $\{1,5\}$ or $\{3\}$ in the old Π_{CG}.

Because IFDI maintains indexes for each partition and its groups, group retrieval, insertion and removal cost constant time. The major cost comes from scanning one group until a value-matched tuple is found. In the worst case, this may be as large as the size of the group. However, we prove that for many common cases, the cost is proportional to the number of distinct values in each column and exponential to the number of columns.

Less Lattice Traversal: Another advantage of IFDI is that it only walks through the lattice nodes that contain the modified attribute. This is because the other nodes must be unchanged after the update. For instance, in Figure 4.3, IFDI only visits the nodes within the ellipse, which saves half of the cost.

2.4.3 Extending IFDI to Nested FDs

In the nested scenario, [102] designed the first system for efficient discovery of XML FDs by extending the algorithm proposed in [51, 67, 79]. In CRIUS, we seamlessly integrate the flat-case IFDI with the discoverXFD algorithm from [102] and propose the incremental nested FD induction algorithm. The algorithm works recursively from the leaves to the root of the schema tree (see Appendix 2.3.1) and builds NFDs from potential flat FDs. It is implemented in CRIUS and studied empirically in Section 5.8.

2.4.4 Maintaining NFDs on Schema Evolution

Schema evolution may also affect Nested FDs: as the nested structure changes, certain FDs may be invalidated or satisfied. Unlike value update, the representation of NFDs largely affects its transformation on schema evolution. [46] proposed a NFD representation which can be either *global* or *local*. We prove that: 1) each local NFD can be represented as an equivalent global NFD and 2) each NFD is preserved on schema evolution, after a trivial transformation. Detailed proofs are omitted due to space.

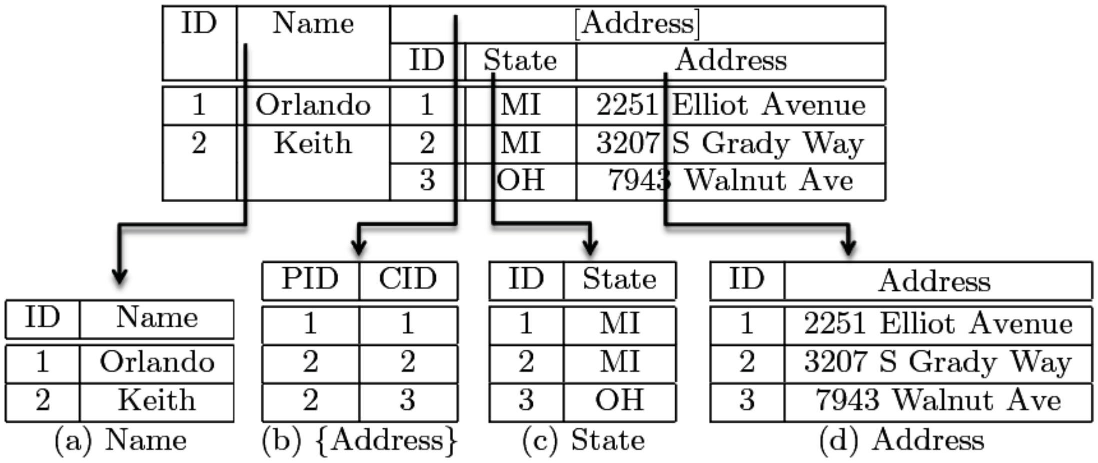
2.5 Relational Database Back-End

Storage management has been so carefully engineered for relational databases that a flat relational storage manager is a natural back-end for a system such as CRIUS. [31] has suggested a possibility to store nested relations using flat tables, but a practical storage plan is left open. There is also excellent related work on storing XML in relational databases, such as [89, 41]. However, such work has traditionally focused on query performance, and has not placed a high priority on the cost of schema evolution.

We extend column store [93, 42] ideas to nested relations and develop a recursive vertical partitioning approach, which eliminates the need to ever use ALTER TABLE when performing schema updates, delivering low-overhead schema evolution. We implemented this on a row-major RDBMS, because column store requires considerable customization on hierarchical structuring. We observe in our experiments that the storage format efficiently supports other tasks (e.g., data display) that are common in our spreadsheet-like environment.

2.5.1 Vertical Partitioning

We represent a nested relation as a recursively vertically partitioned relational database. The vertical partitioning is standard. The recursion is the result of the nesting – additional tables are required to link nested tuples with their corresponding nesting tuples. For example, Figure 2.8 shows a nested relation, and its decomposition, where table (b) links the IDs of nested and nesting tuples. We also maintain a *structure table* to record how the decomposed relations are structured to represent the nested relation.



ID	Name	[Address]		
		ID	State	Address
1	Orlando	1	MI	2251 Elliot Avenue
2	Keith	2	MI	3207 S Grady Way
		3	OH	7943 Walnut Ave

(a) Name

ID	Name
1	Orlando
2	Keith

(b) {Address}

PID	CID
1	1
2	2
2	3

(c) State

ID	State
1	MI
2	MI
3	OH

(d) Address

ID	Address
1	2251 Elliot Avenue
2	3207 S Grady Way
3	7943 Walnut Ave

Figure 2.8: Vertical Partitioning Example

2.5.2 Upward and Downward Mappings

Using the vertically partitioned storage format, we further define a mapping from the relational database to span tables (the *upward mapping*) and another mapping of opposite direction (the *downward mapping*). The upward mapping statically maps the relational database to a span table, according to the decomposition described above, while the downward mapping dynamically maps each span table operator to a sequence of manipulations on the underlying relational database. More details can be found in Appendix 2.6.

2.6 Downward Mapping

We describe the downward mapping for IMPORT with a few key points here. Other mappings are similar.

2.6.1 Mapping IMPORT

A multi-level IMPORT can be evaluated in two stages: (1) transporting source column with value naively duplicated and (2) the merge operation, which corresponds to $\bigsqcup$ operator in PNF. The first stage is making the schema change, and the second stage is adjusting the data to this changed schema.

Let $Source$ denote the source attribute, and $Target$ denote the target relation. According to the tree structure, we can obtain a unique ordered path from $P(Source)$ to $Target$. Let the relations on this path, excluding $P(Source)$, be $\{R_1, R_2...R_n\}$. Also, for any attribute A, denote its corresponding flat table with $Flat(A)$. Suppose $S(Flat(Source)) = \{ID, A_S\}$. Then, the first stage can be formalized by:

$$\begin{aligned} Flat(Source) \leftarrow \Pi_{ID,A_S}(\rho_Y(Flat(Source)) \bowtie_{Y.ID=Z.PID} \\ \rho_{Z(PID,ID)}(\Pi_{L_i.PID,L_n.CID} \\ (\bowtie_{L_i.CID=L_{i+1}.PID} \\ \{L_i = Flat(R_i), L_{i+1} = Flat(R_{i+1}) : i = 1..n-1\}))) \end{aligned} \quad (2.6)$$

This formula finds the tuple relationship by joining all the linking tables from source parent to target. It then joins the result with the original source column, which is a value table, achieving the goal for replicating A_S according to tuple ID correspondence.

Also, the structure table has to be updated. However, the update is extremely simple in that we only need to update the parent relation of the source attribute to the target relation, and do the obvious bookkeeping on the auxiliary information.

For the second stage, a merge is applied recursively, on each relation merging all tuples whose set of atomic attributes at this schema level are the same. This involves three steps: updating atomic values, updating the parent link table, and updating the link tables for all relation-valued children.

We first define the ungrouped table, which is a join of parent link table and all atomic value tables at current level. Suppose the set of atomic attributes in current level are $\{R_1, R_2, ...R_n\}$, with $S(Flat(R_i)) = \{ID, A_i\}$. Also suppose the parent relation of current level is $Parent$, then:

$$\begin{aligned} &ungrouped(Parent) \leftarrow \\ &\qquad \Pi_{PID,CID,A_1,A_2,...,A_n}(\rho_Y Flat(Parent) \\ &\qquad\qquad (\bowtie_{Y.CID=V_i.ID} \{V_i = Flat(R_i)\})) \end{aligned} \tag{2.7}$$

After merge, the corresponding table would be:

$$\begin{aligned} &grouped(Parent) \leftarrow \\ &\qquad ungrouped(Parent)\ group\ by\ \{PID, A_1, A_2, ..., A_n\} \end{aligned} \tag{2.8}$$

If the size of ungrouped and grouped table is the same, than there is no need to merge. Otherwise:

$$Flat(R_i) \leftarrow \Pi_{CID,A_i}(grouped(Parent)) \tag{2.9}$$

For $i = 1..n$, and:

$$Flat(Parent) \quad \leftarrow \quad \Pi_{PID,CID}(grouped(Parent)) \tag{2.10}$$

The link tables of child relation-valued attributes are updated in a similar manner. The merge stage is done by following this procedure recursively. The recursion will never exceed the target relation since schema elements below that relation are untouched.

In general, the whole structure of the flat database needs very few changes, since most of the structural updating is reflected by the structure table. Further, no “ALTER TABLE” command is required, which avoids the high cost of traditional schema updates.

2.6.2 Mapping other operators

The EXPORT works quite similarly to IMPORT in terms of specifying tuple relationship by joining linking tables, except no merge is required. FLOAT and SINK also work in a similar manner, with much simpler procedures required.

For other schema update operators, Adding/Dropping columns requires only creating/dropping flat tables corresponding to the attributes and insertion/deletion of tuples in the structure table. Permutation is trivially exchanging attribute IDs in the structure table.

Data manipulation operations are much simpler. INSERT and DELETE update the corresponding value tables and link tables. UPDATE is similar to INSERT except an additional merge is required for preserving PNF.

2.7 Evaluation

Our experiments are designed to answer four main questions:

1. How usable is the drag-and-drop interface in CRIUS, compared to state-of-the-art GUI schema design tools?
2. How usable is the integrity-based guidance?
3. How efficient is the incremental FD induction algorithm, compared to traditional FD induction approaches?
4. What are the performance implications of the storage representation, for common tasks (schema modification and data display)? How does the vertically-decomposed format compare to a standard relational storage format?

2.7.1 User Study on Schema Operations

Our first set of experiments measured the usability of schema manipulation in CRIUS. We recruited eight volunteers with no data-base background, and two database experts for comparison. All subjects are aged from 18 to 25, and have a Bachelor's Degree. As a baseline, we compared CRIUS to Microsoft SQL Server Management Studio 2008 (SSMS), which is representative of the state-of-the-art GUI-based relational database design. In the first experiment, we asked the users to define three relations: 1) a Person relation with two attributes: FirstName and LastName, 2) an Email relation with one attribute Email. and 3) a Phone relation with one attribute Phone. We also required the users to structure the database hierarchically so that each person may have multiple emails and phones. We taught the users how to do this in both CRIUS and SSMS. In CRIUS, we taught them to use the Span Table operators to create the structure. In SSMS, we taught them to specify foreign key references from both Email and Phone to Person, by creating ID columns and dragging links between them to indicate foreign key references using the database diagram UI in SSMS.

We asked the same group of subjects to accomplish this task using both CRIUS and SSMS, and recorded the time to define the schema with both tools. We randomized the order of which system is used first, in order to counterbalance the learning effect.

The times shown in Figure 2.9 (D for database expert, N for non-technical users). Results demonstrate that design using CRIUS was about three times faster.[3] While the first experiment tested the performance of users construct-

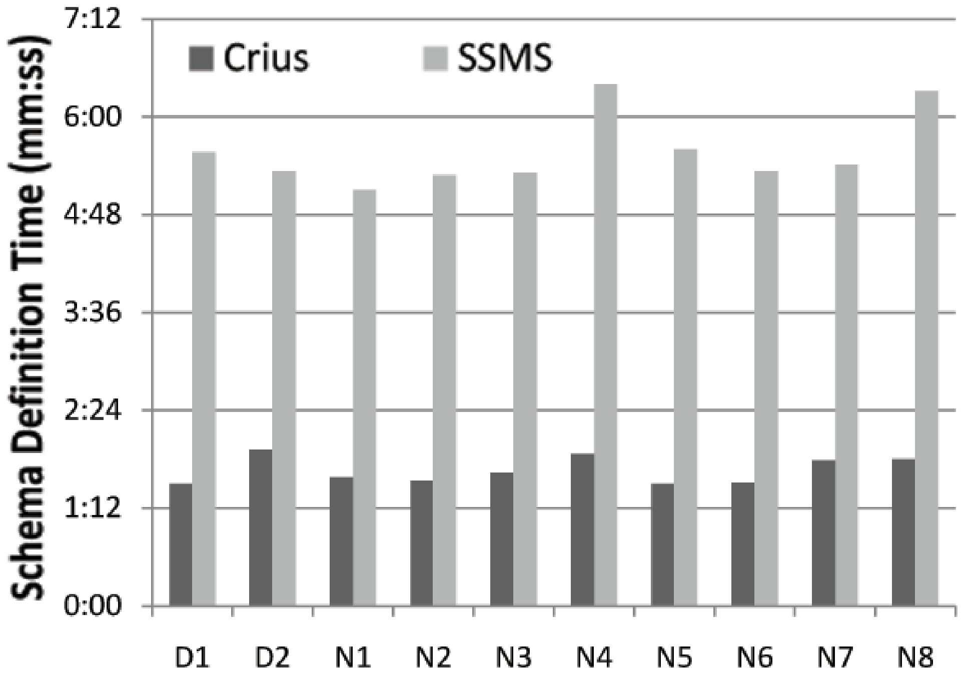


Figure 2.9: Time defining a schema with CRIUS vs. SSMS.

[3]Using the Mann-Whitney test (a standard test of statistical significance), this difference is significant with p-value < 0.0002.

ing a schema from scratch, we also conducted a second experiment, which asked users to modify an existing schema by moving an attribute.

Specifically, we used a more complex schema from MiMI [56]. We focused on two relations: the Gene relation and the Interaction relation. Gene records individual gene information. It consists of five attributes: gene_id, symbol, type, taxid, and description. Interaction stores the basic information describing how two genes which interact with each other, including nine attributes: int_id, gid1, symbol1, type1, taxid1, gid2, symbol2, type2, and taxid2. gid1 and gid2 are foreign keys referencing gene_id. We nested Interaction inside Gene by gene_id to bring them into a single spreadsheet. We asked the users to move one attribute (Description) from Gene to Interaction, which was a practical need from current MiMI users.

To accomplish the move using CRIUS, users only need to specify an IMPORT by a drag-and-drop. However, this task proved difficult in SSMS because, while users could construct a new schema using the GUI tool, migrating data from the old schema to the new one required them to write SQL. As a result, none of the non-technical users were able to complete the task using SSMS. In contrast, all users were able to complete the task within seconds using CRIUS.

Finally, to gain further insight into the usability of CRIUS, we asked the same users to perform the same schema update task using CRIUS and a strawman system that we constructed. The strawman implements a very similar drag-and-drop GUI interface to CRIUS; however, unlike CRIUS's span table algebra, it implements drag-and-drop manipulations that are direct implementations of nested relational algebra operators.

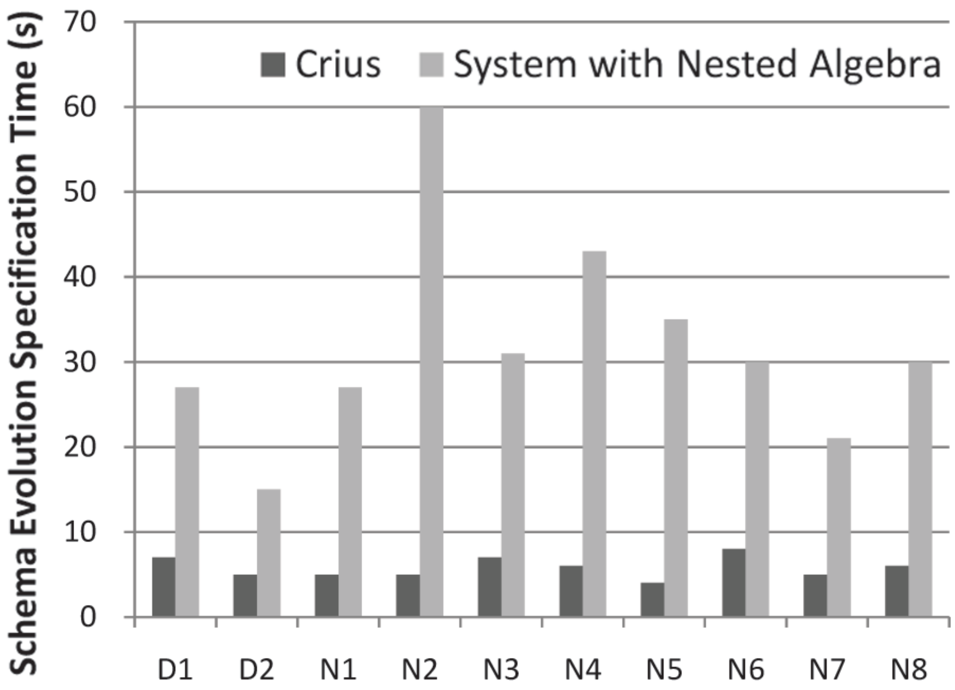


Figure 2.10: Time specifying an attribute transportation with CRIUS v.s. Nested Algebra GUI.

Figure 2.10 compares the user performance for this task.[4] All the users were able to accomplish the task almost thrice faster using CRIUS than using the system with a nested algebra interface.[5] This difference supports the intuition that the span table operators supported in CRIUS are more natural direct manipulations for users than nested algebra operators, even though the two are equivalent in expressive power within the schema restructuring domain.

2.7.2 User Study on Integrity-Based Guidance

Our second user study measured how much the users may benefit from the integrity-based guidance offered by CRIUS. Again, we recruited eight non-technical volunteers and two computer experts. The subjects were asked to complete three tasks on an address book in CRIUS twice, once with the guid-

[4]Since we reduced the database size to fit the user study, the query execution time is negligible compared to the user operation time.

[5]This is statistically significant with p-value < 0.0002.

ance feature on and the other off. Their tasks were: 1) insert a new contact and his address into the address book, 2) update the cell phone number of one contact and 3) update the address of one contact to the address of another contact. For each subject, we measured the time for each task, and the overall count of key strokes and mouse clicks.

For this user study, we designed an address book with schema $Name, \{Address, Zipcode, HomePhone, CellPhone\}$. It contained three contacts and six addresses, and induced these FDs: $Name \rightarrow CellPhone$, $HomePhone \rightarrow Zipcode$, $Address \rightarrow HomePhone$, $HomePhone \rightarrow Address$ and $Address \rightarrow Zipcode$. A brief tutorial on how to use CRIUS was given to each subject prior to the study. We started timing after the subject was clear about each task and ready to execute it. In order to counterbalance the learning effect, the guidance feature was turned on first for half of the studies, and turned off first for the other half.

All subjects finished all of the tasks. From the time shown in Figure 2.11, we observe a significant improvement with the guidance feature on. [6] This demonstrates that CRIUS successfully leverages integrity constraints to save data entry time and improves usability.

We also recorded the number of key strokes and mouse clicks, since they may serve as strong evidence of input errors. The numbers are shown in Figure 2.12. Although detailed intermediate errors were hard to report, these numbers show that users made many fewer errors with integrity-based guidance. [7] In other words, this supports that CRIUS improves usability by preventing input errors.

[6] According to the Mann-Whitney test, the p-values are 0.0002, 0.0209 and 0.001 for the three tasks, respectively.
[7] Key strokes has a p-value of 0.0002 and mouse clicks has 0.0019.

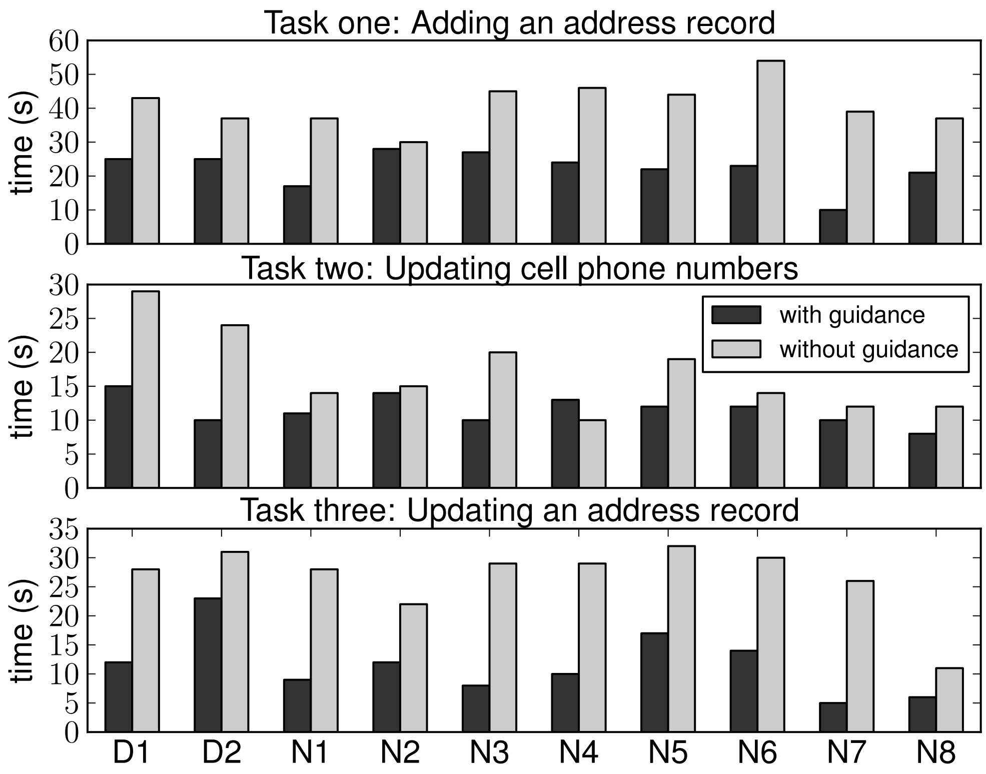


Figure 2.11: Time for data entry tasks, with guidance on and off

2.7.3 Performance of IFDI

To evaluate the effectiveness of our incremental FD induction algorithm, we conducted experiments to compare the performance for both the naive approach (which recalculates the FDs for each update) and the IFDI algorithm, in a simulated incremental update environment. We measured the average time for both approaches to generate a new set of FDs upon simulated updates in simulated tables, by varying the number of columns and rows.

We first fixed the number of columns to five and recorded the time at each row size. The result is shown in Figure 2.13(a). While the performance of the naive approach is linear in the number of rows, the cost for IFDI was extremely

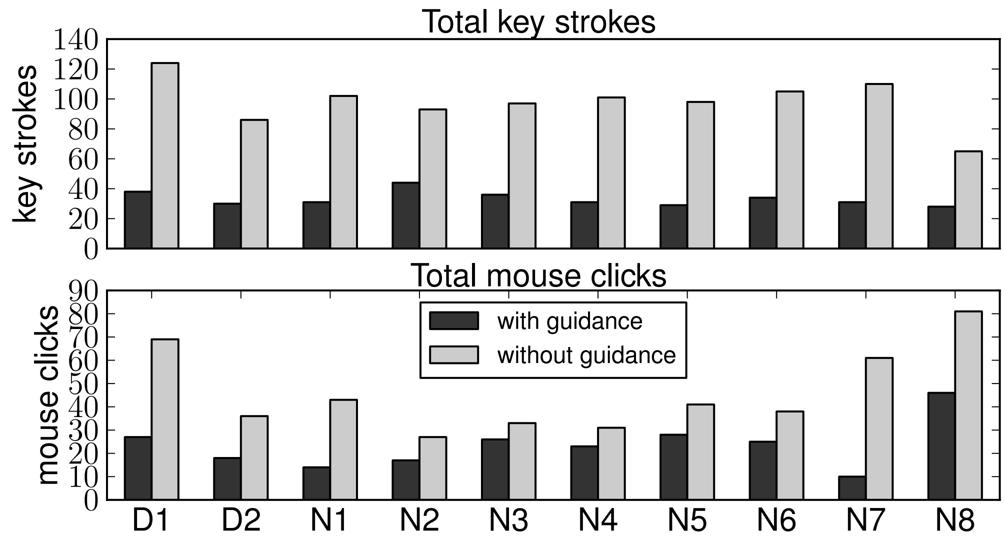
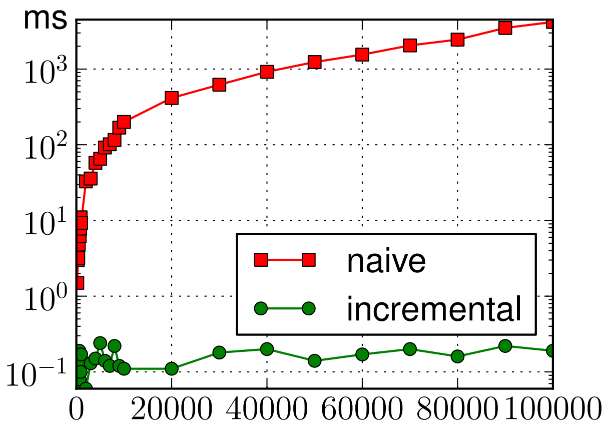


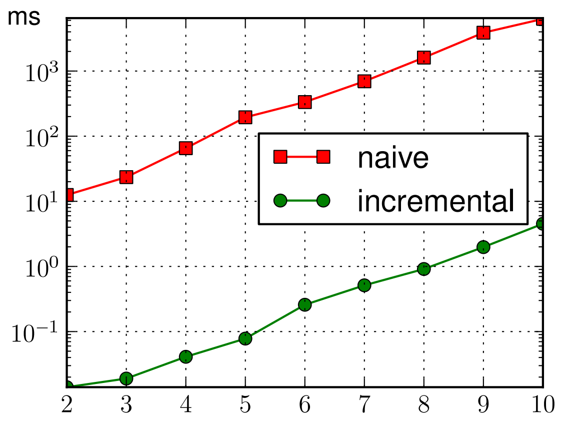
Figure 2.12: Number of key strokes and mouse clicks for data entry tasks, with and without guidance

small and nearly constant. This is because for each lattice node, the traditional algorithm reconstructed each partition by doing a partition product, which involves a linear scan on the input partitions. In contrast, IFDI only updates existing partitions and touches rows with the same values as the updated tuple.

Our second experiment fixed the number of rows to ten thousand and measured the average time for various number of columns. The result is shown in Figure 2.13(b). While the costs for both approaches increase exponentially, the IFDI is more than three orders of magnitude faster. Again, the improvement in performance is due in large part to an tremendous decrease in partition rows accessed by the incremental algorithm, as shown in Figure 2.14. For example, for five columns and ten thousand rows, the incremental algorithm accessed an average of 42.9 partition rows for each update, while the naive approach accessed an average of 257196.9 partition rows.



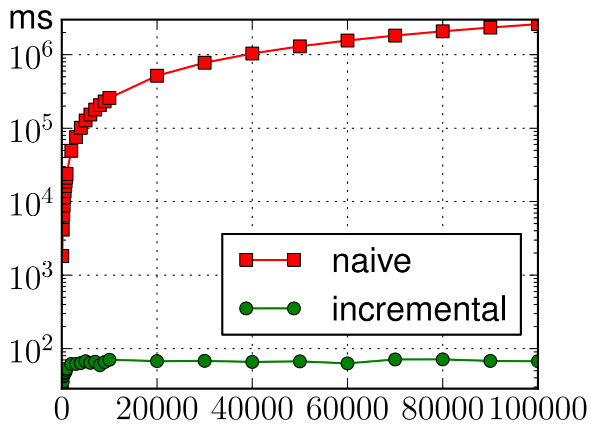
(a) a five-column table with varying row size



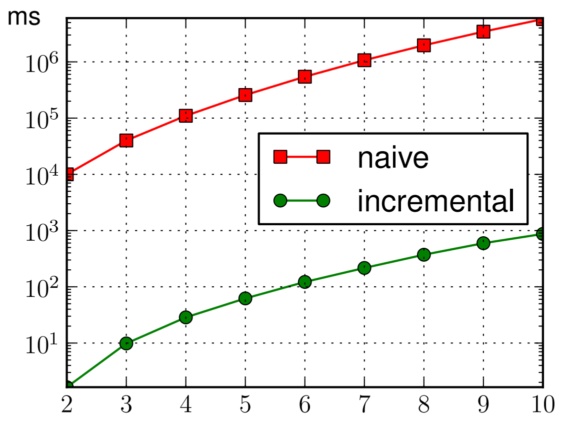
(b) a ten-thousand-row table with varying column size

Figure 2.13: Average time generating new FDs using naive approach and IFDI.

For both experiments, we simulated the table to have one near-FD (50% of the rows satisfied the FD) from the first column to the second. All the data (except in the determined column) was randomly generated from a value pool of size ten. Each cell update changed an FD-violating row to a FD-satisfying row by updating its second column. We repeated the experiment by varying the number of rows and columns. Time in both tests were averaged upon ten tables and one hundred updates for each table.



(a) a five-column table with varying row size



(b) a ten-thousand-row table with varying column size

Figure 2.14: Average number of partition rows accessed when generating new FDs using naive approach and IFDI.

2.7.4 Performance of Vertical Storage

Our last set of experiments compared the performance of the vertically-partitioned storage in CRIUS with a naive approach, which stores tables contiguously using a row-major layout. Recall that in designing the storage system we had two main goals: (1) efficient schema evolution, and (2) efficient data display.

Schema Update Performance

Our first experiment measured the time for the same schema update task in 2.7.1, on both a naive storage and the vertical partitioned storage in CRIUS. Specifically, we set up two versions of the MiMI: The first (Naive) stores the two relations just as they are stored in MiMi, with gene_id and int_id as the primary keys. The second (CRIUS) partitions the two relations in a per-column manner similar to that depicted in Figure 2.8, with a primary key ID column associated with each column table. In both cases, only a clustered index on the primary key is constructed for each relation. Our experiment repeatedly moved the description attribute between the Gene and Interaction relations in each database.

The result is shown in Figure 2.15. As expected, the vertically partitioned storage offered much faster schema modifications than the naive approach, particularly for large databases. This is because for each ALTER TABLE command, the naive method must restructure an entire relation, which greatly degraded performance. In contrast, CRIUS only manipulated the column tables involved in the move.

Data Display Performance

In addition to schema update, our vertically partitioned storage also supports other common tasks such as data display. To show this, our last experiment measured the time for loading data and constructing a span table from both the native storage and CRIUS storage. Specifically, we focused on a query to retrieve the gene_id, symbol, type, and taxid of all the genes that interac-

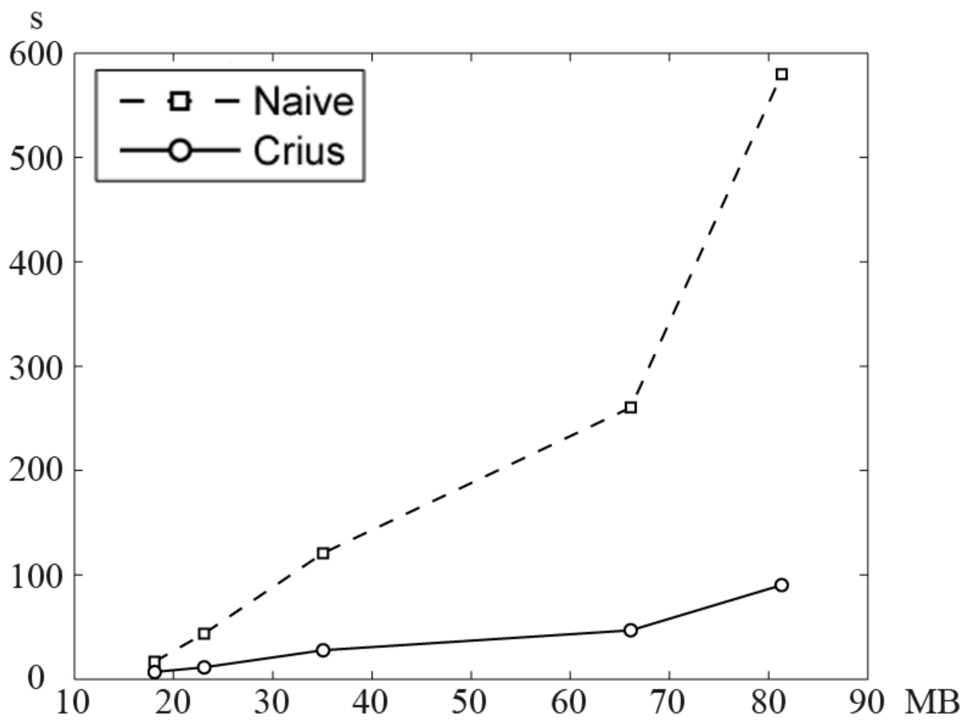


Figure 2.15: Average time transporting an attribute in CRIUS vs. naive storage, for different database scale

t with a given gene whose symbol matches a random pattern. We executed this query on both the naive and CRIUS databases. The results are shown in Figure 2.16.

The time cost increased linearly with the number of attributes projected in CRIUS, but remained constant for the naive storage. This is because in CRIUS, the number of required joins grows linearly with the number of columns selected. At the same time, while data display queries are more efficient using the naive storage format, the difference between the naive and CRIUS is not huge, despite the additional joins required by CRIUS. This supports that CRIUS is able to improve the performance of schema modification tasks, without losing much efficiency on the data display task.

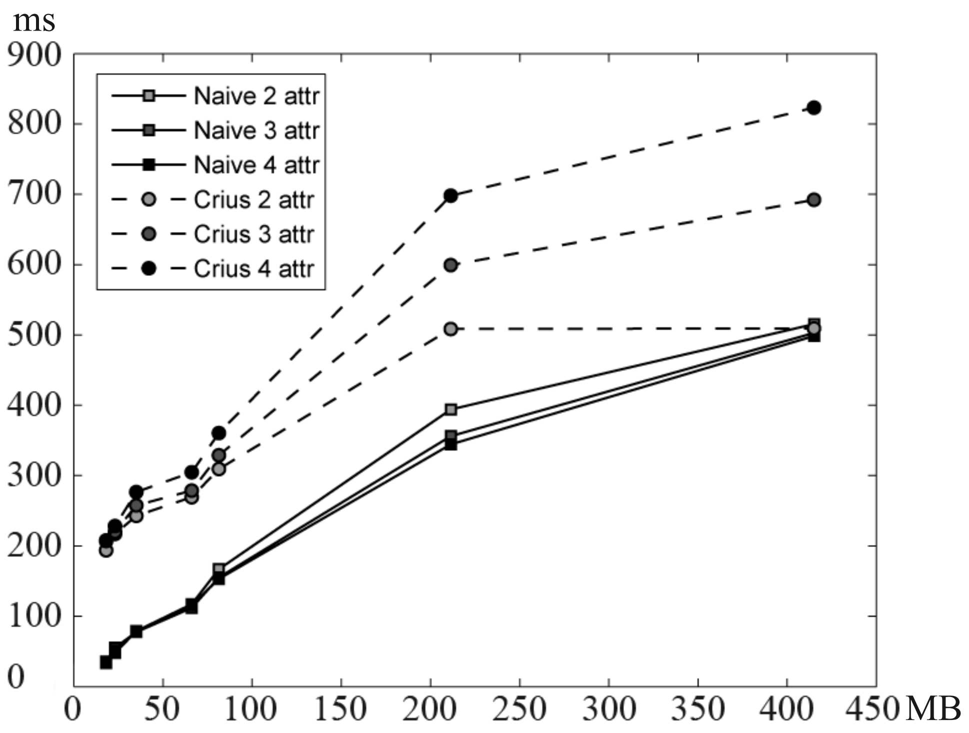


Figure 2.16: Average data display time in CRIUS vs. the naive storage, for different database scale

2.8 Conclusions

In this chapter, we described the design and implementation of CRIUS, a system that allows non-technical users to develop and evolve schemas for their data, within the familiar context of a spread-sheet-style interface and data model. In support of the friendly drag-and-drop user interface, we developed a novel span table algebra that is equivalent in expressive power to the nested algebra.

User operations on a spreadsheet are usually error-prone. Integrity constraints, such as functional dependencies can save user input and protect the data from mistakes. However, incremental FD induction is non-trivial when the database is modified frequently. In this chapter, we introduced the first algorithm for incrementally updating the set of induced FDs upon value update and schema evolution. CRIUS uses these FDs to recommend auto-completions for updates and to warn the user about potential data entry errors.

While the data model exposed to CRIUS users is a nested span table, the underlying storage model may be different. Because of its pervasiveness, we elected to store our span tables as flat relations in a relational database system. While past work has considered storing nested tables in flat relations, the mapping is only treated as a simulation to establish theoretical results. In contrast, CRIUS requires materialization of such a mapping to support efficient query processing and efficient schema evolution. For these reasons, we implemented a recursively vertically decomposed storage format. We also show how span table algebra operators can be mapped to SQL operations on the relational database.

Finally, we evaluated the CRIUS implementation via both user studies and performance experiments. The user study showed that the drag-and-drop schema modification meets our primary goal of making schema evolution easy to specify, and our integrity-based guidance feature effectively reduces data entry effort and input errors. The performance results indicated that the our incremental FD induction algorithm is much faster compared to the traditional approaches, and the vertical decomposition storage format is considerably more efficient than past techniques for schema evolution, while query processing performance remains reasonable.

CHAPTER III

MWeaver: Sample-Driven Schema Mapping

End-users increasingly find the need to perform light-weight, customized schema mapping. State-of-the-art tools provide powerful functions to generate schema mappings, but they usually require an in-depth understanding of the semantics of multiple schemas and their correspondences, and are thus not suitable for users who are technically unsophisticated or when a large number of mappings must be performed.

We propose a system for **sample-driven** schema mapping. It automatically constructs schema mappings, in real time, from user-input sample target instances. Because the user does not have to provide any explicit attribute-level match information, she is isolated from the possibly complex structure and semantics of both the source schemas and the mappings. In addition, the user never has to master any operations specific to schema mappings: she simply types data values into a spreadsheet-style interface. As a result, the user can construct mappings with a much lower cognitive burden.

In this chapter we present MWeaver, a prototype sample-driven schema mapping system. It employs novel algorithms that enable the system to obtain desired mapping results while meeting interactive response performance requirements. We show the results of a user study that compares MWeaver with two state-of-the-art mapping tools across several mapping tasks, both real and synthetic. These suggest that the MWeaver system enables users to perform practical mapping tasks in about *1/5th* the time needed by the state-of-the-art tools.

3.1 Introduction

A schema mapping transforms a source database instance into an instance that obeys a target schema. It has long been one of the most important, yet difficult, problems in the areas of data exchange and data integration [14, 61, 63]. Traditional database applications in E-business, data warehousing and semantic query processing have required good schema mappings among heterogeneous schemas. Moreover, as the amount of structured Web-based information explodes (*e.g.*, Wikipedia, Freebase, Google BigTable, etc.), users are directly exposed to the task of combining, structuring and re-purposing information [24]. Doing so inevitably requires schema mapping to be democratic: non-technical users should be able to cook their data with their own flavor, even if they cannot master the “professional kitchenware” designed for database experts.

Due to the importance of the schema mapping problem, a handful of mapping design systems have been developed. These systems include InfoSphere Data Architect (from Clio [84]), BizTalk Mapper [2], Altova MapForce [1], and Stylus Studio [3]. All of these systems are based on the same general method-

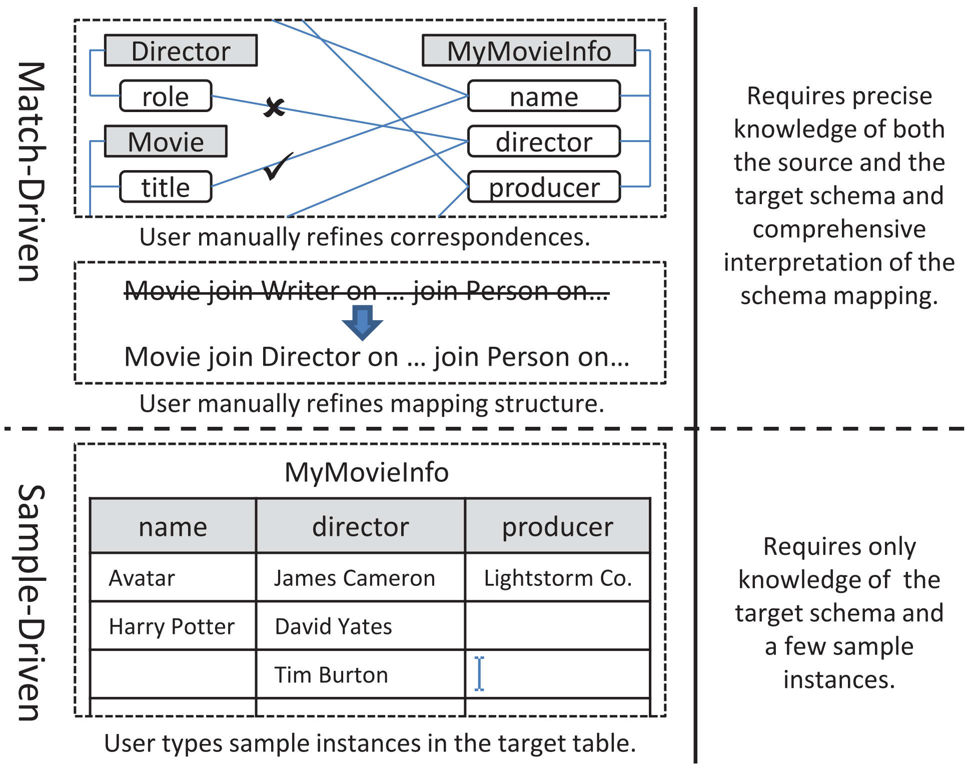

Match-Driven
Director
role
Movie
title
MyMovieInfo
name
director
producer
User manually refines correspondences.
~~Movie join Writer on … join Person on…~~
Movie join Director on … join Person on…
User manually refines mapping structure.
Requires precise knowledge of both the source and the target schema and comprehensive interpretation of the schema mapping.
Sample-Driven
MyMovieInfo
name
director
producer
Avatar
James Cameron
Lightstorm Co.
Harry Potter
David Yates
Tim Burton
User types sample instances in the target table.
Requires only knowledge of the target schema and a few sample instances.

Figure 3.1: A Comparison between the Match-Driven Approach and the Sample-Driven Approach

ology that was first proposed in Clio [84]. The methodology consists of two phases. In the first *matching* phase, a set of correspondences between source and target schema elements is solicited from the user, with possible aid from automated techniques that find similar attribute pairs [72, 68, 60, 33, 69, 32, 36, 78, 35]. During the second *mapping* phase, the set of matches yields an executable transformation from the source schema to the target schema.

Unfortunately, this traditional *match-driven* model is unsuitable for many modern schema mapping tasks. The user must either build attribute-level matches from scratch, or else painstakingly double-check an automatically-generated set of matches. An implicit assumption made by these systems is that the user has detailed knowledge of both the source and target schemas.

For traditional schema mapping tasks that involve a sophisticated administrator and a single high-value target database, this assumption makes sense. But modern mapping scenarios feature relatively unsophisticated users and a multiplicity of tasks: a DBA may only map two HR databases together when a corporate acquisition takes place, whereas a Web advertising analyst may need to combine schemas of different data-sets multiple times a day. For these less-technical users who perform a large number of mappings, the laborious *match-driven* process can be a heavy burden.

A Sample-Driven Approach In this chapter, we propose a *sample-driven* approach that enables relatively unsophisticated end-users, not DBAs, to easily construct their own data. The key idea behind our approach is to allow the user to implicitly specify schema mappings by providing sample data of the target database. Behind the scenes, the system automatically elicits the mappings that transform the source database into this partially-described target. After the user has provided enough information, the system can determine a single best mapping. The process is iterative. As the user types more information from the target database, the system provides increasingly better estimates of the correct mapping. Figure 3.1 depicts a high-level comparison between the traditional match-driven approach and the sample-driven approach.

Our *sample-driven* approach reduces the user's cognitive burden in two ways. First, the user no longer needs to explicitly understand the source database schema or the mapping. She simply types in sample instances until the system converges to a single proposal. Second, the operations that the user performs are common and require no special training: she simply types in data as in a spreadsheet. In contrast, current tools are applications designed for trained DBAs, and require users to decide whether individual attribute-level matches are correct. The following example provides some intuition about how the *sample-driven* approach works.

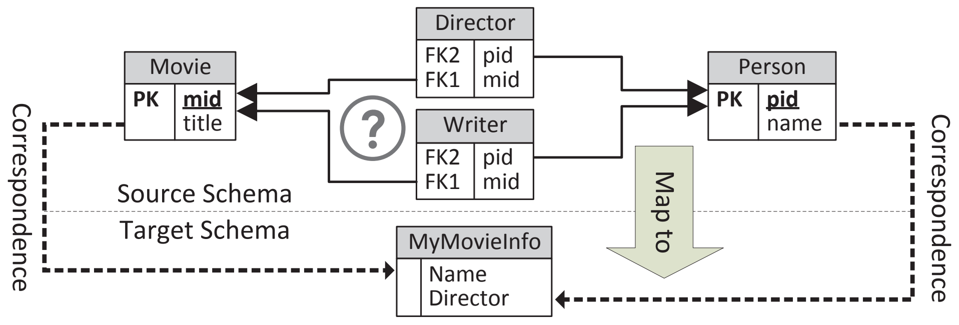


Figure 3.2: An Example Schema Mapping with The Question Mark Indicating a Join Path Ambiguity.

Example III.1. A user is exploring the Yahoo Movies database, and wishes to store the movie title as **Name** and the director name as **Director** in a target **MyMovieInfo**, as shown[1] in Figure 3.2.

A typical match-driven system proposes attribute correspondences to the users, as shown in Figure 3.3. The user needs to either pick out each correct correspondence from multiple candidates, or scan the source schema for the correct correspondence if it is not proposed; both situations require a comprehensive knowledge of the schemas.

In a sample-driven approach, the user freely provides sample instances for the target **Director** field. For each director name she provides, the system automatically searches the source database to find all attributes that contain the name. For example, if a user enters `Ed Wood`, the system may find the value in both **Person**.**name** and **Movie**.**title**. As the user enters more names, the set of

[1]We only show a subset of the source schema due to space. The solid arrows represent foreign key constraints.

attributes eventually converges to a single attribute **Person.name**, indicating that the user has implicitly specified the correspondence from **MyMovieInfo.Director** to **Person.name**.

Even if a match-driven system perfectly generates all the attribute-level correspondences, it may has to deal with multiple possible mappings. In this example, imagine the system matches **MyMovie-Info.Name** to **Movie.title**, and **MyMovieInfo.Director** to **Person.name**. There are still two possible ways to construct the mapping[2]: one by joining **Movie** and **Person** via **Director**, the other by joining via **Writer**. (See Figure 3.2.) Current match-driven systems usually pick only one mapping, which may not be the desired one [9]. Even if the system presents both candidate mappings to the user, she still has to manually select the desired join via **Director**.

In contrast, the sample-driven approach also considers data-level information to help find the correct mapping. For example, if the user enters (`Harry Potter`, `David Yates`) in the target, the system will know that the join must NOT be via **Writer**, as the source indicates that the writer of `Harry Potter` is `J. K. Rowling`.

Of course, the user must be familiar with the target schema in order to provide samples. While traditional database schemas can be quite complex, expert DBAs are likely to know the data well enough to give useful samples. Non-traditional users may not be familiar with database schemas in general. But as the trend for light-weight, Web-based information integration increases, our computationally unsophisticated users are likely to be well-informed about the target database they want to build. In either case, it is reasonable to

[2]We only consider joins via foreign key constraints.

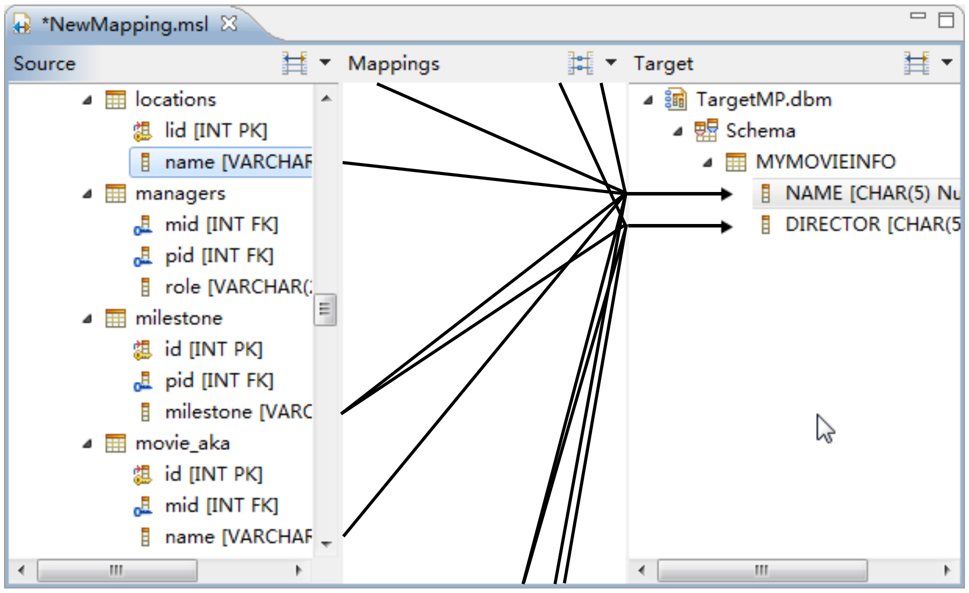
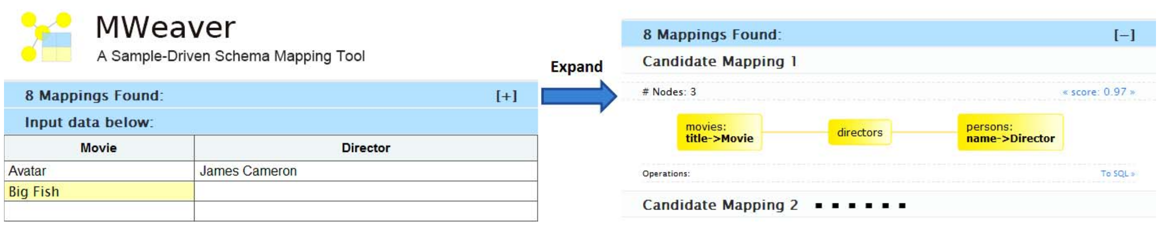


Figure 3.3: A Screenshot of IBM InfoSphere Data Architect

believe that the sample-driven approach will be suitable for a large group of mapping scenarios.

In this chapter, we design and prototype a sample-driven mapping system, MWeaver, which facilitates schema mapping tasks for end-users. We also conduct a detailed user study that compares users' behavior with MWeaver and with state-of-the-art mapping tools. We show that by reducing the cognitive burden for the user, and providing a familiar spreadsheet-style interface, MWeaver allows the average user to perform a practical mapping task in about *1/5th* the time needed for the traditional match-driven approach.



Movie	Director
Avatar	James Cameron
Big Fish	

Figure 3.4: A Screenshot of MWeaver. Left: The Input Spreadsheet. Right: The Expanded List of Candidate Mappings.

Technical Challenges MWeaver renders schema mapping easier from the user's perspective, but actually building the runtime system presents two substantial technical challenges. First, it must obtain the desired mapping using just the user-provided samples. Doing so can entail locating each piece of sample throughout the source database, and then deriving all possible mappings that those pieces together suggest. Second, these mappings must be computed quickly enough that the user can obtain "interactive-speed" feedback, allowing her to review the current system status before continuing to provide more samples or stopping if the system has generated the desired mapping.

Our Contributions Our work makes the following contributions:

- We propose a sample-driven approach to facilitate schema mapping tasks for end-users, and present a prototype system, MWeaver.
- We develop an efficient sample search algorithm and show that it can obtain provably correct results at interactive speeds.
- We present the results of a detailed user study that demonstrates, among other things, that a typical MWeaver user can obtain schema mappings in 1/5th of the time required by state-of-the-art mapping tools.

This chapter is organized as follows. We first provide an overview of MWeaver in Section 3.2. The sample search algorithm is described in Section 3.3. Section 3.6 describes how we iterative prune the candidate mappings. In Section 5.8, we present user studies that demonstrate the usability of our system, as well as performance experiments that demonstrate the efficiency of our algorithm. Finally, we conclude in Section 3.8.

3.2 System Overview

In this chapter, we propose MWeaver, a sample-driven schema mapping tool that constructs schema mappings based on user-input sample instances. By assuming the user-input samples are approximately present[3] in a source database instance that we have access to, the major advantage of MWeaver is that it isolates the user from the possible complexity of such a source database and its schema. MWeaver takes as input the source instance and a partially filled spreadsheet, and produces as output the schema mappings that map the source to a target containing that spreadsheet. We assume the schema mappings are Project-Join queries over relational database. While selection, aggregation and user-defined functions would largely strengthen the expressive power of the mappings, we do not study them in this work because they may produce information loss that is non-recoverable on the target side.

Since we do not expect end-users to be able to specify foreign key constraints [54], we assume in this chapter that the target schema comprises one or more table "views", each of which has joined all the information the user wants to see at one time. Since these views are independent, they can be constructed one at a time. Without loss of generality, we can assume the target schema is a single table.

User Interface: The primary UI component of MWeaver is a spreadsheet that conforms to the target schema. We call it the ***Input Spreadsheet***. On the left of Fig 3.4 shows an input spreadsheet in which the user is filling data. The user may adjust the input spreadsheet by adding/dropping/renaming columns to

[3]We will detail this notion of "approximation" in Section 3.3.1

meet her mental model of the target. The bar directly under the logo provides information about the current mapping generation status. By default it only displays the number of mappings currently found. If the user wishes to know more about the mappings, she may expand the information bar by clicking the “plus” on the right. This will trigger a ***Mapping List***, which visualizes each mapping with details.

In the mapping list, each mapping is visualized as an undirected tree. Each node in the tree is labeled with the source relation involved in the mapping, together with the correspondences between the target columns and the source attributes. Each edge in the tree represents how these source relations are joined in the mapping.

User Input: After the structure of the input spreadsheet is fixed, the user can input data in any cell in the spreadsheet. Formally, a user input is $Input(i, j, c)$, which updates the content of the cell on the i-th row and the j-th column of the input spreadsheet to c. We call the content in each non-empty cell of the input spreadsheet a ***sample***. We do not consider empty cells.

Interaction Model: The system interacts with the user by maintaining a set of mappings upon each user input. We call such mappings ***candidate mappings***. The interaction starts with the user filling out the first row of the input spreadsheet. We require the first row to be fully populated in order to establish a general impression of the complete desired mapping. After this, MWeaver constructs the initial set of candidate mappings from the samples in the first row. Formally, we name this process ***Sample Search***.

Afterwards, the user may continue to provide sample instances in any cell below the first row. Whenever one cell is updated, MWeaver uses all the samples (i.e., non-empty cells) from that row to prune the set of candidate mappings. We call this process ***Sample Pruning***. Finally, the interaction terminates when there is only one mapping left. As long as the user input correctly reflects her knowledge and her knowledge is consistent with the source database, the remaining mapping must be the desired mapping. In other words, as the number of candidate mappings decreases, the average mapping quality increases w.r.t. the number of user-input sample. This finally produces a single best mapping which meets the user requirement. We will elaborate on the sample search and pruning techniques in Section 3.3 and Section 3.6, respectively.

3.3 Sample Search

3.3.1 Problem Formalization

We consider a ***source database*** D_S with ***schema*** S_S that has n ***relations*** $R_1, ..., R_n$ and a target database with ***target schema*** S_T comprising a single ***target relation*** R. A ***schema mapping*** M is a project-join query that maps S_S to S_T. For each R_i, $i \in [n]$[4], we denote its schema by $S(R_i)$ and its instance by $I(R_i)$. $S(R_i)$ is the set of all the attributes in R_i. Similarly, R has a schema $S(R) = \{A_1, ..., A_m\}$, where m is the ***size*** of the target and A_j ($j \in [m]$) represents the j-th attribute in R. $t[A]$ stands for the projection of tuple t on attribute A.

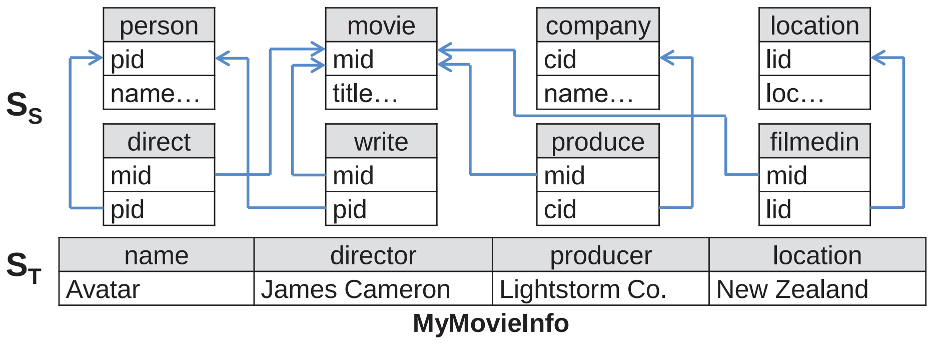
The user types in samples in the input spreadsheet under the target schema. Each sample E is a string. We denote the first row of samples by $t_E = (E_1, ..., E_m)$,

[4]Throughout the chapter, we denote $\{1..n\}$ by $[n]$.

and call it a ***sample tuple***. Our goal of sample search is to find all the schema mappings that transform the source database to a target “containing” the sample tuple.

Because the user-input may not have an exact match in the source, as we use them to generate the schema mappings, we forgive inaccurate samples by allowing them to be “noisily contained” by some database instance. Formally, we define this “noisily contain” relationship by a binary operator $\succeq$, which returns a boolean value based on the desired error model. Having this operator, we say $t[A]$ contains sample E iff $t[A] \succeq E$. Similarly, we say t contains E iff $\exists A$ s.t. $t[A] \succeq E$. Furthermore, given $t_E = (E_1, ..., E_m)$, we say t contains t_E, iff $\forall i \in [m], t[A_i] \succeq E_i$. Finally, we say a target database D_T contains t_E iff $\exists t \in D_T$ s.t. t contains t_E. Having this concept of containment, we define sample search as follows.

Definition III.2 (Sample Search)**.** Given a source database D_S and a sample tuple $t_E = (E_1, ..., E_m)$, sample search finds all *schema mappings*[5] M such that $M(D_S)$ contains t_E. Each such mapping is called a *valid schema mapping*.



name	director	producer	location
Avatar	James Cameron	Lightstorm Co.	New Zealand

Figure 3.5: The Source Schema and the Target Relation with Samples

[5]We restrict the search space with certain constraints that we will detail in Section 3.3.4

Example III.3 (The Running Example)**.** Let D_S be part of the Yahoo Movie Database with its schema S_S partially shown in Figure 3.5. Let S_T has $R =$ $\mathbf{MyMovieInfo}$ with $S(R) = \{\mathbf{name}, \mathbf{director}, \mathbf{producer}, \mathbf{location}\}$, where $\mathbf{name}$ indicates the movie title, $\mathbf{director}$ represents the name of the movie director, $\mathbf{producer}$ identifies the company which produces the movie and $\mathbf{location}$ specifies the filming location. Suppose the user enters a sample tuple $(\texttt{Avatar}, \texttt{James Cameron}, \texttt{Lightstorm Co.}, \texttt{New Zealand})$. The sample search aims to find all the schema mappings that produce a target database that contains the sample tuple. We will use this as a running example throughout this section.

3.3.2 The Challenge and The Opportunity

Intuitively, one way to solve the sample search problem is to model the whole source database as a graph with each tuple represented by a vertex and each foreign key reference by an edge. Then the problem is equivalent to finding all the subgraphs such that each user-input sample is contained by a vertex of the subgraph. However, the fan-out of vertices in such a graph can be very large (e.g., a director may have directed dozens of movies, and one movie genre may even contain thousands of movies). As a result, searching such a graph may be very inefficient [20].

An alternative approach is to first generate a group of mappings that will *possibly* yield a target database that contains the sample tuple, and then execute each of the mappings on the source database to see if it is actually a valid mapping. Such “lucky” mappings could be constructed by joining sample-containing relations in the source database via various foreign key relationships. Unfortunately, the number of such “lucky” mappings grows exponentially with respect to both the number of joins allowed and the size of the target. Because a “lucky” mapping has to be executed before one knows whether its result contains the sample tuple, this approach may require exponential rounds of database accesses. This can be verified in similar database keyword search scenarios [50, 5].

In fact, the sample search problem is NP-hard, because it essentially deals with the problem of searching a graph for all sub-graphs that satisfy certain properties [59]. However, the definition of sample search implies that, if a mapping is invalid, then any mapping that structurally contains it must be invalid. This inspires us to construct valid mappings from smaller ones to larger ones. Since generating smaller valid mappings is relatively cheap, as long as we can efficiently build larger valid mappings on top of the smaller ones, we can potentially meet practical interactive requirement.

3.3.3 Our Solution

In this chapter, we propose a **tuple path weaving** algorithm TPW to solve the sample search problem. We first create schema mappings for each pair of samples (*pairwise mappings*), and then check their validity within the limit of acceptable noise. The execution returns a set of *pairwise tuple paths*, which are essentially instance-level support for the mappings. A mapping is valid iff such supporting set is non-empty. After that, we “weave” these pairwise tuple paths purely in memory to generate larger and larger paths, which finally cover all the samples. Lastly, we extract the valid mappings from these “complete” tuple paths and return them with ranking.

Informally, TPW functions in the following five major steps.

1. **Find sample occurrences** in the source database.
2. For each pair of sample occurrences, **generate pairwise mapping paths** by searching their possible connections in the source schema.
3. For each pairwise mapping, **create a set of pairwise tuple paths**. Do so by translating the mapping into an approximate search query[6], then executing it in the source database. This will produce all pairwise tuple paths that support the mapping. Mappings with no support will be pruned.
4. From these pairwise paths, build all **complete tuple paths** in a bottom-up manner.
5. **Rank** mappings extracted from the complete tuple paths.

By pruning invalid mappings in an earlier stage, TPW avoids exponential rounds of database accesses. Moreover, since instance-level exploration is done in step 2, there is no overhead from traversing the database with a potentially large tuple fan-out. Finally, it is reasonable to expect "weaving" to be efficient, because as tuple paths grow larger, their number decreases dramatically, which we have verified by experiments and will describe in Section 3.7.3.

3.3.4 Definitions

Before we dive into a detailed description of TPW, we need some definitions. Hereafter, for any graph or tree structure g, we use $V(g)$ to denote all its

[6]We use standard full-text search techniques for such approximate search in our implementation.

vertices, $E(g)$ to denote all its edges, and $T(g)$ to denote all its terminal vertices (vertices of degree one).

Schema Graph To create pairwise mappings, we need to search for possible join paths between the sample-containing relations. This requires modeling the source schema as a graph. Because a "null" value can not contain any samples, we only consider inner join here. Since inner join is symmetric, we omit the direction of the foreign key to primary key relationship hereafter.

Definition III.4 (Schema Graph)**.** The *schema graph* G is an undirected graph that defines relation joinability according to the foreign key to primary key relationships in S_S. It has a vertex R_i for each relation $R_i \in S_S$ and an edge (R_i, R_j) for each foreign key to primary key relationship from R_i to R_j in S_S.

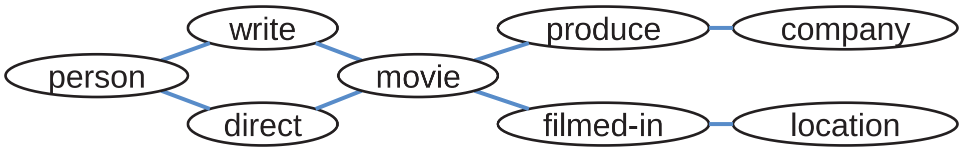


Figure 3.6: The Schema Graph of the Running Example

Figure 3.6 shows the schema graph of Example 1.

Relation Path Each possible join path among the sample-containing relations specifies a unique mapping structure. We extract this structure by a path of relations.

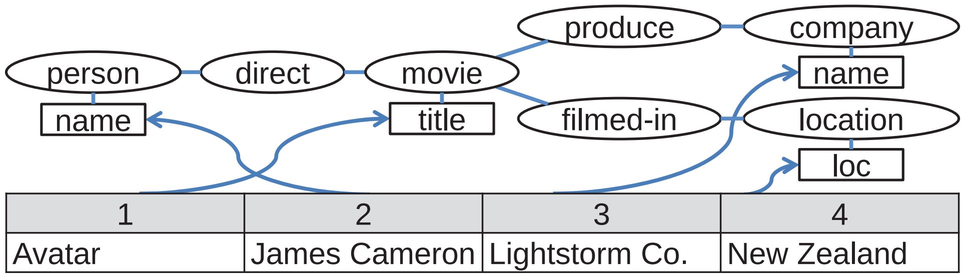
Definition III.5 (Relation Path)**.** A *relation path* p is an undirected tree such that: (1) $\forall u \in V(p)$, u has a corresponding relation $R^u \in V(G)$ and (2) $\forall (u, v) \in E(p)$, $(R^u, R^v) \in E(G)$.

Note that the same relation can appear multiple times in a relation path, as long as the second condition is satisfied. This potentially means the size of the relation tree is not upper bounded by the size of the schema graph.

Mapping Path Having the join structure defined, a mapping also needs to specify which attributes in which relations are projected to the target relation. We capture this information by a *projection map*, which maps a subset of the target attributes to attributes belonging to vertices on the relation path. Intuitively, each terminal vertex must have at least one projection, or it is redundant. Recall that the target relation has size m, we have the following definition.

Definition III.6 (Mapping Path)**.** Given a relation path p, let N be a subset of $[m]$ and $A(p) = \bigcup_{u \in V(p)} S(R^u)$. A *mapping path* is p augmented with a projection map $pm : N \to A(p)$, such that $\forall v \in T(p)$, $\exists i \in N$ and $a \in S(R^v)$, s.t. $pm(i) = a$.

The definition says that a mapping path is essentially a relation path whose terminal vertices have some attributes projected to the target. Attributes in non-terminal vertices may also be projected.



1	2	3	4
Avatar	James Cameron	Lightstorm Co.	New Zealand

Figure 3.7: One Desired Mapping Path for the Running Example

A mapping path is equivalent to a schema mapping in that it can be translated to a SQL query that maps the source database to the target relation. The projection can be fully determined from the projection map while the joining of relations is implied by the structure of the relation path. Hereafter, we use mapping path and schema mapping interchangeably.

Informally, we say a mapping path is ***valid*** iff the corresponding schema mapping is valid. A mapping path can be ***partial***, when N is a proper subset of $[m]$. We define the ***size*** of the mapping path to be the size of N. Specifically, we call a mapping path with size two a ***pairwise mapping path*** and a mapping path with size m a ***complete mapping path***. Our goal mapping paths must be complete. Figure 3.7 exhibits a complete mapping path that is one of the possible answers to the running example.[7]

Since the size of a relation path is unbounded by the schema, the corresponding schema mapping size is also unbounded. This may lead to an infinite number of mappings, most of which make little practical sense. We will restrict the family of mappings we explore by a join number constraint, which is detailed in Section 3.3.5.

Tuple Path In general, a mapping path may or may not be valid. Because a mapping path is merely a *schema-level object*, and does not guarantee the samples can be connected in the source database instance following its path. Indeed, a mapping path is valid iff there is a corresponding *instance-level support*. Such a support should comprise source database tuples that connect samples via the mapping path. The formal definition of such a support is given below.

Definition III.7 (Tuple Path)**.** A tuple path r is an instantiated mapping path such that: (1) for each vertex $u \in V(r)$, there is an associated tuple $t^u \in I(R^u)$

[7]We also show the sample tuple for clarity.

and (2) for each $(u, v) \in E(r)$, t^u and t^v are directly connected in the source database by the foreign key to primary key relationship between R^u and R^v.

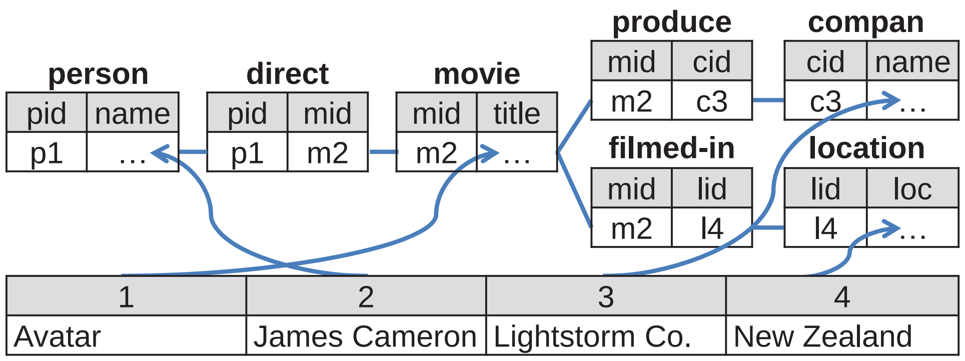


Figure 3.8: One Tuple Path Supporting the Desired Mapping

We define the ***size*** of a tuple path in the same way we did for a mapping path. We call a tuple path with size two a ***pairwise tuple path*** and one with size m a ***complete tuple path***. Figure 3.8 depicts a complete tuple path that instantiates the mapping path in Figure 3.7.[8]

One mapping path may be instantiated to any number of tuple paths. Formally, we say a mapping path is ***valid*** iff it can be instantiated to at least one tuple path. And our goal of sample search is to find those valid complete mapping paths.

3.3.5 TPW Algorithm

In this section, we elaborate the five steps (see Section 3.3.3) of the TPW algorithm. Details can be found in Appendix 3.4.

[8]We present only the primary keys, foreign keys and the projected attributes due to space.

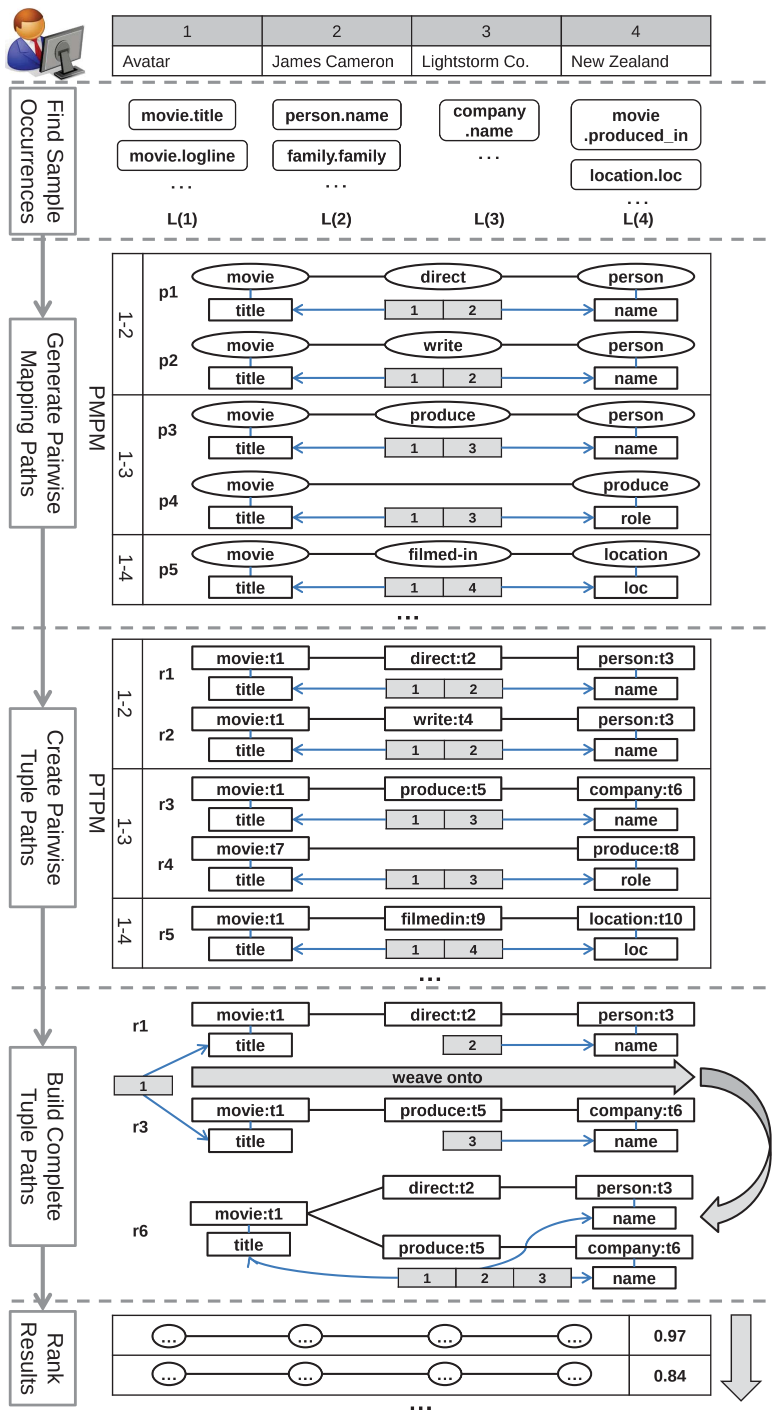


Figure 3.9: The Tuple Path Weaving Algorithm TPW

Find Sample Occurrences

A mapping path can be arbitrary. However, if a source attribute does not contain any samples, the mapping path that projects that attribute can not be valid. Therefore, we first narrow our search space by focusing only on the source attributes that contain at least one sample. Formally, we construct a *location map* L, where $L(i)$ ($i \in [m]$) is the set of all the source attributes containing E_i.

Example III.8. In the running example, we first search for sample `Avatar`. We have $L(1) = \{\textbf{movie}.\textbf{title}, \textbf{movie}.\textbf{logline}\}$, because these are all the attributes that contain `Avatar`. Similarly, $L(2) = \{\textbf{person}.\textbf{name}, \textbf{family}.\textbf{family}\}$ since they contain `James Cameron`. A more complete L is shown in Figure 3.9.

Pairwise Mapping Path Generation

Next, we generate all the pairwise mapping paths from schema graph G and location map L. The size of such pairwise mapping paths can be arbitrarily large, because the number of joins in these mapping paths is unbounded by the source schema [50]. Fortunately, we realize in practice, the mappings that project two attributes from two relations that are joined via many intermediate relations who have no attribute projected are very rare. Therefore, we set up a threshold value $PMNJ$ (*Pairwise Maximal Number of Joins*) to restrict the family of mapping paths we explore. Specifically, we say a mapping path satisfies the $PMNJ$ constraint iff the largest number of joins between each pair of projected attributes on the path is no larger than $PMNJ$. And we only aim to generate the mapping paths that satisfy the $PMNJ$ constraints. In this running example, we will set $PMNJ = 2$, since it is enough to generate the complete mappings of interest.

The pairwise mapping paths are generated as follows. First, for each $j \in [m]$

and each attribute $A^j \in L(j)$, we issue a breadth-first search from the vertex R^j that contains A^j in the schema graph G with depth limited by $PMNJ$. Whenever we encounter a vertex R^k that contains $A^k \in L(k)$ with $k > j$, we construct a relation path backward from R^k to R^j and augment it with a projection map built from A^j and A^k. The created pairwise mapping paths are stored in a pairwise mapping path map $PMPM$ with key (j, k).

Example III.9. In the running example, we first generate the pairwise mapping path linking `Avatar` and `James Cameron`. We start by picking $\mathbf{movie.title}$ from $L(1)$, $\mathbf{person.name}$ from $L(2)$ and searching for the relation paths between them. By a breadth-first search from $\mathbf{movie}$ in G, we reach $\mathbf{Person}$ via two paths: $\mathbf{movie\text{-}direct\text{-}person}$ and $\mathbf{movie\text{-}write\text{-}person}$. We augment them with projection information from the two attributes and respectively obtain the mapping paths **p1** and **p2**, and store them in $PMPM$ with key $(1, 2)$, as shown in Figure 3.9.

Pairwise Tuple Path Creation

Once all the pairwise mapping paths are generated, we produce tuple paths that instantiate them. To do this, we retrieve each pairwise mapping from $PMPM$, translate it into an approximate search query with the keywords constraints derived from the samples and execute it. If the result set is empty, we prune the corresponding pairwise mapping because any larger mapping containing it must be invalid. Otherwise, we construct a pairwise tuple path from each returned tuple, associate it with the corresponding mapping and store it in a pairwise tuple path map $PTPM$. Since foreign-key connection is normally sparse in real-life datasets, we expect these tuple paths to fit in memory. Subsequent operations can also be completed in memory, because the number of the tuple paths decreases dramatically w.r.t. its size, as we will see in Section 3.7.3.

Example III.10. In Figure 3.9, suppose the source tuple **t1** contains `Avatar`, **t3** contains `James Cameron` and **t2** links **t1** and **t3** by foreign keys. Because these tuples successfully support mapping path **p1**, we create a tuple path **r1** from them, associate it with **p1** and store it in $PTPM$ with key $(1, 2)$. **r2** - **r5** are created similarly.

Complete Tuple Path Construction

Since all the necessary information for constructing the complete mapping paths is stored in the pairwise tuple paths, we can then perform the "weaving" in memory. "Weaving" essentially merges a pairwise tuple path onto an existing ***base tuple path*** to produce a new larger tuple path. We use the word "weave" to describe the process of traversing the pairwise tuple path from one end to the other and gradually merging its vertices onto the base tuple path. The "weaving" terminates once a vertex fails to merge.

If all vertices on the pairwise tuple path can be successfully merged onto the base tuple path, the latter will preserve its structure. Otherwise, the "unwoven" part of the the pairwise tuple path will enrich the structure of the base tuple path by adding a "tail" to it. Either way, the cost of "weaving" is upper-bounded by the size of the pairwise tuple path, which is in turn bounded by $PMNJ$.

Since various larger tuple paths may share smaller sub-paths, we "weave" the tuple paths in a bottom-up manner. Specifically, we organize tuple paths by ***levels***. Level n contains all the tuple paths of size n, $n \in \{2..m\}$. For each n from 2 to $m-1$, for each base tuple path in level n, we "weave" a pairwise tuple path onto it to create a tuple path in level $n+1$. This terminates when all the complete tuple paths are "woven" and stored in level m.

The following example illustrates how the "weaving" works. A detailed description can be found in Algorithm 6 in Appendix 3.4.

Example III.11. We start by constructing tuple paths of size three from a base tuple path of size two, say, **r1** in Figure 3.9. Then, we enumerate all the possible pairwise tuple paths that may be woven onto **r1**. Those are pairwise tuple paths whose projection map has exactly one key in common with that of **r1**. For instance, it is possible to weave **r3** or **r4** onto **r1** since their keys intersect on key 1.

Next, we attempt to merge the two paths by fusing the two vertices pointed by the common key in both paths. If the tuples associated with the two vertices are identical, the two vertices are fused and the two paths are merged successfully. Otherwise, the merge fails and returns no new path. Here, **r1** and **r3** successfully merge by fusing the first vertex (since they agree on tuple **t1**). However, **r1** and **r4** fail to merge (because **r4** has tuple **t7**).

Finally, we issue a synchronized search from the fused vertex along the pairwise tuple path to be woven. For each following vertex v on the path, we search in the base tuple path for the next adjacent vertex u such that $t^u = t^v$. If such a vertex does not exist, we stop and return the current tuple path. Otherwise we fuse u and v and proceed to the next vertex. This continues until the whole pairwise tuple path is traversed. In this example, there is no vertex which is adjacent to **movie:t1** in **r1** and has a tuple **t5** as in **r3**. So the process terminates and the new tuple path **r6** is returned.

After all tuple paths of size three are generated, we begin to construct tuple paths of size four (the complete tuple paths) from those ones with size three. It is straightforward that **r5** can be woven onto **r6**, producing the complete tuple path shown in Figure 3.8. Another complete tuple path is constructed by weaving **r3** and **r5** onto **r2**.

Ranking

We extract all the complete mapping paths from the complete tuple paths and rank them before returning them to the user. For each complete tuple path, we define its score to be a weighted sum of two scores: a matching score which indicates how well the samples match the actual data in the tuple, and a complexity score which is the number of joins in the tuple path. The score of a complete mapping path is the average score of all its corresponding tuple paths.

3.3.6 Soundness and Completeness

The TPW algorithm is sound and complete in that every complete tuple path generated corresponds to a valid schema mapping, and every valid schema mapping satisfying the PMNJ constraint is discovered by the algorithm. The soundness is straightforward. The completeness is because, for any valid schema mapping that generates a tuple t containing the sample tuple, we can always construct the corresponding complete tuple path from the source tuples that contribute to t, which are completely recorded when generating the pairwise tuple paths. The proof can be found in Appendix 3.5.

3.4 TPW Algorithms

3.4.1 Find Sample Occurrences

Given a sample tuple and a source database, Algorithm 1 locates in the source all the occurrences of each sample in that tuple by scanning all the source attributes (loops at line 3 and 5). Whenever a source attribute contains a sample (line 6), it is registered to the corresponding location map (line 7 and 11). The check on line 6 is done by a standard full-text search on an individual column which has a pre-computed inverted-index. Line 7 and 11 nest the location map by relation so we can easily generate relation path hereafter.

Algorithm 1 LocateSample

```
Require: A sample tuple $(E_1, ..., E_m)$, source database $D_S$ with schema $S_S$
Ensure: A location map $L = \{L_1, ..., L_m\}$
 1: Initialize $L$
 2: for $i \in \{1..m\}$ do
 3:     for relation $R^i \in D_S$ do
 4:         $L_{R^i} \leftarrow \emptyset$
 5:         for attribute $A \in R^i$ do
 6:             if $\exists t \in R^i$ s.t. $t[A] \succeq E_i$ then
 7:                 $L_{R^i} \leftarrow L_{R^i} \cup \{A\}$
 8:             end if
 9:         end for
10:         if $L_{R^i} \neq \emptyset$ then
11:             $L_i(R^i) \leftarrow L_{R^i}$
12:         end if
13:     end for
14: end for
```

3.4.2 Pairwise Mapping Path Generation

Algorithm 2 constructs all the pairwise mapping paths that satisfy the $PMNJ$ constraint by issuing a breadth-first search on the schema graph G. To perform the breadth-first search, we define a structure *relation node*, which has three properties: $relation$ which points to the corresponding relation in the source database, $dist$ which records the number of joins from this node to the origin and $parent$ which stores the previous node during the search. We also define *Path Vertex*, which composes a relation path. It has a property $relation$ which stores the corresponding relation, and $neighbor$ which stores all its neighbors. A relation path maintains a $lookup$ table to manage vertex membership. Also, we compute a map in advance to prepare for mapping path creation.

We first construct relation paths from the schema graph G and the location map L, as described in Algorithm 2. For each sample E_i (line 2) and each relation R^i that contains it (line 3), we issue a breadth-first search from the relation by calling $Grow()$ (See Algorithm 3). $Grow()$ returns every relation path whose length is no larger than $PMNJ$ and connects R^i with another relation R^j containing E_j ($j > i$). Finally we construct all the pairwise mapping paths by appending registered attributes to each returned relation path by invoking $Create()$ (See Algorithm 4).

Algorithm 2 GeneratePairwiseMapingPath

```
Require: The location map $L$, the schema graph $G$
Ensure: $PMPM$, a map from (i,j) to a set of pairwise mapping paths, where $i, j \in \{1..m\} \wedge i < j$
 1: Initialize $PMPM$
 2: for $i \in \{1..m\}$ do
 3:     for relation $R^i \in Keys(L_i)$ do
 4:         Initialize relation node $r$
 5:         $r.relation \leftarrow R^i$
 6:         $r.dist \leftarrow 0$
 7:         $r.parent \leftarrow null$
 8:         $rs$ = Grow($L$, $r$, $i$, $G$, $PMNJ$)
 9:         Create($rs$, $PMPM$)
10:     end for
11: end for
```

Algorithm 3 Grow

```
Require: The location map $L$, a relation node $r$ containing $E_i$, the index $i$, the schema graph $G$, the
    maximal number of joins $PMNJ$
Ensure: All the relation paths from $r$ to $s$ within $PMNJ$ joins, where $s$ is a relation node containing $E_j$
    and $j > i$
 1: Initialize a relation path set $rs$
 2: Initialize a queue of relation node $Q$
 3: $Q.push(r)$
 4: while Q is not empty do
 5:     $c \leftarrow Q.pop()$
 6:     for $j \in \{i + 1..m\}$ do
 7:         if $c \in Keys(L_j)$ then
 8:             Create relation path $p$ backward from j to i.
 9:             $rs.add(p)$
10:         end if
11:     end for
12:     if $c.dist < PMNJ$ then
13:         for $R'$ that is adjacent to $c.relation$ in $G$ do
14:             Initialize relation node $n$
15:             $n.relation \leftarrow R'$
16:             $n.dist \leftarrow c.dist + 1$
17:             $n.parent \leftarrow c$
18:             $Q.push(n)$
19:         end for
20:     end if
21: end while
```

3.4.3 Pairwise Tuple path Creation

For each pairwise mapping path in $PMPM(i, j)$, we then search for all the pairwise tuple paths that instantiate it, and store them in $PTPM(i, j)$. This is done by the following steps. (1) We translate the mapping path into a SQL query, with the join conditions defined by the path structure. (2) For each relation on the mapping path, we project all of its primary keys. (3) We expand the query by full-text search constraints derived from the sample tuple. (4) We execute the full-text search query. (5) For each tuple in the result set: (i) for each relation on the mapping path, we generate a universal tuple id for the provenance tuple in that relation from the schema of that relation and the values of all the projected primary keys. (ii) We collect such tuple ids for all the relations on the mapping path, align them in the same structure as in the

Algorithm 4 Create

```
Require: A relation path set $rs$, the pairwise mapping path map $PMPM$
Ensure: $PMPM$
 1: for $r \in rs$ do
 2:     Let $i, j \in DOM(r.pm)$ and $i < j$
 3:     for $a_i \in R^{r.pm(i)}$ do
 4:         for $a_j \in R^{r.pm(j)}$ do
 5:             Initialize a mapping path $p$ from $r$
 6:             Append $a_i$ to $p.pm(i)$
 7:             Append $a_j$ to $p.pm(j)$
 8:             $PMPM(i, j) \leftarrow p$
 9:         end for
10:     end for
11: end for
```

mapping path and store them in $PTPM$.

3.4.4 Complete Tuple Path Construction

After all the pairwise tuple paths are generated, we no longer need to access the database since the tuple paths contain complete information to derive the candidate mappings. Specifically, we compute the complete valid mapping paths by "weaving" the tuple paths in a bottom-up manner, as described in Algorithm 5. At each level (line 3), we retrieve all the tuple paths generated from the previous level (line 4) and try to weave each pairwise tuple path onto these base tuple paths (line 5-18). The weaving is described in Algorithm 6, which simultaneously traversed two tuple paths (line 11, 12), compare corresponding vertices by checking the identities of the tuples associated with them (line 13) and update path structure on successful merges (line 14-16).

3.5 Proofs

Here we prove that the tuple path weaving TPW algorithm that solves the sample search problem is sound and complete.

Algorithm 5 GenerateCompleteTuplePath

```
Require: The pairwise tuple path map $PTPM$, the size of the target schema $m$
Ensure: The complete tuple path map $CTPM$
 1: Initialize tuple path set $pl$, $npl$
 2: $pl \leftarrow PTPM$
 3: for $i \in \{2..m\}$ do
 4:     for $tp \in pl$ do
 5:         for $ptp \in PTPM$ do
 6:             Initialize a set $ck$
 7:             $ck \leftarrow DOM(tp.pm) \cap DOM(ptp.pm)$
 8:             if $ck.size = 1$ then
 9:                 Let $k$ be the only element in $ck$
10:                 Initialize new tuple path $ntp$, $nptp$
11:                 $ntp \leftarrow tp$
12:                 $nptp \leftarrow ptp$
13:                 $ntp \leftarrow Weave(ntp, nptp, k)$
14:                 if $ntp \neq null$ then
15:                     $npl.add(ntp)$
16:                 end if
17:             end if
18:         end for
19:     end for
20:     $pl \leftarrow npl$
21: end for
22: $CTPM \leftarrow pl$
```

3.5.1 Soundness

Theorem III.12 (Soundness)**.** *Any valid complete mapping path generated by* TPW *corresponds to a valid schema mapping.*

We first give some definitions.

Definition III.13 (Valid Schema Mapping)**.** Given a sample tuple $(E_1, ..., E_m)$, we say a schema mapping M is valid on $N \subseteq [m]$ iff $\exists t \in M(D_S)$ s.t. $\forall i \in N, t[A_i] \succeq E_i$.

Definition III.14 (Tuple Path Projection)**.** Given a tuple path p with projection map pm, its projection t_p is a tuple such that $\forall i \in DOM(pm), t_p[i] = t^{u_a}[a]$, where $a = pm(i)$ and u_a is the vertex containing a in p.

Definition III.15 (Valid Tuple Path)**.** Given a sample tuple $(E_1, ..., E_m)$ and a tuple path p with projection map pm, p is valid iff $\forall i \in DOM(pm), t_p[i] \succeq E_i$.

Algorithm 6 Weave

```
Require: A tuple path $tp$ of size $n$, A pairwise tuple path $ptp$
Ensure: A tuple path $ntp$ of size $n+1$
 1: Initialize a set $visited$ for visited path vertex
 2: $u \leftarrow tp.map(k)$
 3: $v \leftarrow ptp.map(k)$
 4: if $u.tuple\_id \neq v.tuple\_id$ then
 5:     $ntp \leftarrow null$
 6:     return
 7: end if
 8: $visited.add(u)$
 9: $ptp.lookup.remove(v)$
10: while $ptp.lookup$ is not empty do
11:     Let $v'$ be the next vertex of $v$ in $ptp$
12:     for $u' \in u.neighbor \wedge u' \notin visited$ do
13:         if $u'.tuple\_id \neq v'.tuple\_id$ then
14:             $u.neighbor.add(v')$                    ▷ Update path structure
15:             $v'.neighbor.remove(v)$
16:             $v.neighbor.add(u)$
17:             return
18:         end if
19:         $u \leftarrow u'$
20:         $v \leftarrow v'$
21:         $visited.add(u)$
22:         $ptp.lookup.rmove(v)$
23:     end for
24: end while
```

Hereafter, given a mapping path or tuple path p, we call the schema mapping translated from it its *corresponding schema mapping* and denote it by M_p. Securely, if a tuple path p is valid, then M_p must be valid on $DOM(p.pm)$ since t_p exists in $M_p(D_S)$ and $\forall i \in DOM(p.pm)$, $t_p[A_i] \succeq E_i$.

Then, we have the following lemma.

Lemma III.16. *Let p be a tuple path of size n, q be a pairwise tuple path and $DOM(p.pm) \cap DOM(q.pm) = \{k\}$. Let $r = Weave(p, q, k)$ and assume $r \neq null$. If p is valid and q is valid, then r is also valid.*

The proof to Lemma III.16 is straightforward. According to Algorithm 6, since $r \neq null$, p and q must have been successfully merged. Thus $t_r = t_p \cup t_q$ and $\forall i \in DOM(r.pm), t_r[A_i] \succeq E_i$.

Finally, we give the proof to Theorem III.12.

Soundness. According to the definition, a mapping path is valid iff there is at least one tuple path that instantiates it. So it is equivalent to prove that any valid complete tuple path generated by TPW corresponds to a valid schema mapping. Since we generate pairwise tuple paths by executing full-text search queries in the source database, every pairwise tuple path p must be valid. Also, according to Lemma III.16, if each tuple path or size n is valid, then each tuple path of size $n+1$ must also be valid. According to mathematical induction, every complete tuple path must be valid. For each complete tuple path p, M_p defines the corresponding valid schema mapping. □

3.5.2 Completeness

Theorem III.17 (Completeness)**.** *Any valid schema mapping whose corresponding mapping path satisfies the PMNJ constraint must be discovered by* TPW.

Hereafter, we assume every mapping path satisfies the PMNJ constraint for simplicity. We first have the following lemma.

Lemma III.18. *If all the valid tuple paths of size n are discovered by* TPW*, then all the valid tuple paths of size $n+1$ must also be discovered.*

Proof. Suppose we have a valid tuple path r of size $n+1$. We decompose it into two parts as the following. First we choose any of its terminal vertex, say u. According to Definition III.6, there must be a map index associated with it. We denote that index by i. Then we traverse r from u until we meet the first vertex v which has an attribute projected by another map index j. We split r on v together with its map split on j into two tuple paths: a pairwise tuple path p connecting u and v, and a tuple path q of size n containing the rest of r including v. Because r is valid, it is straightforward that p and q are both valid. According to the procedure we generate the pairwise tuple paths, each valid pairwise tuple path must be discovered by TPW. So p must be discovered. According to the hypothesis, q must also be discovered. Let $r' = Weave(q, p, j)$. Since TPW is deterministic, we must have $r' = r$. In other words, r must also be discovered. □

This leads to the proof to Theorem III.17.

Completeness. Because a schema mapping is valid iff there exists a corresponding valid tuple path, the proof is equivalent to: TPW is able to discover any valid tuple path. According to the procedure we generate the pairwise tuple paths, all the valid pairwise tuple paths must be discovered. According to Lemma III.18 and mathematical induction, all the complete valid tuple paths must also be discovered. □

3.6 Sample Pruning

After the initial set of valid candidate mappings is generated, the user may continue to enter additional samples to prune the candidate set. We call this process sample pruning. Given that our sample search returns a limited number of candidate mappings, the pruning can be processed in a straightforward manner as follows.

Pruning by Attribute Suppose the user enters a new sample E_i in column i on another row in the input spreadsheet. We record all the source attributes that contain E_i. Any mapping path that does not map i to any of these attributes is then discarded.

Pruning by Mapping Structure After the user enters a new sample E_i in column i, whenever there is more than one sample on the same row, we execute an approximate search query for each candidate mapping with the keywords constraints derived from these samples. We discard a mapping if the search returns zero result.

Example III.19. In the running example, suppose the user continues to enter `Big Fish` on the first column of the second row, and we find that the attribute $\mathbf{Movie}.\mathbf{logline}$ does not contain `Big Fish`. As a result, any mapping that maps the first column to $\mathbf{Movie}.\mathbf{logline}$ will be discarded. Similarly, if the user continues to enter `Tim Burton` on the second column of the second row, we will issue a search query with `Big Fish` and `Tim Burton` on each candidate mapping. Any mapping that joins $\mathbf{Movie}$ and $\mathbf{Person}$ via $\mathbf{Writer}$ will be discarded, because the writer of `Big Fish` is not `Tim Burton` and, as a result, the query will return empty set.

3.7 Evaluation

The critical test for our sample-driven mapping tool is whether it truly renders schema mapping tasks feasible for end-users. In this section we demonstrate a positive result in two ways. First, we show that our sample-driven tool reduces user effort for completing the mapping task when compared to both a standard match-driven system and a state-of-the-art QBE-like mapping tool. Specifically, we present a user study that compares the usability of the three

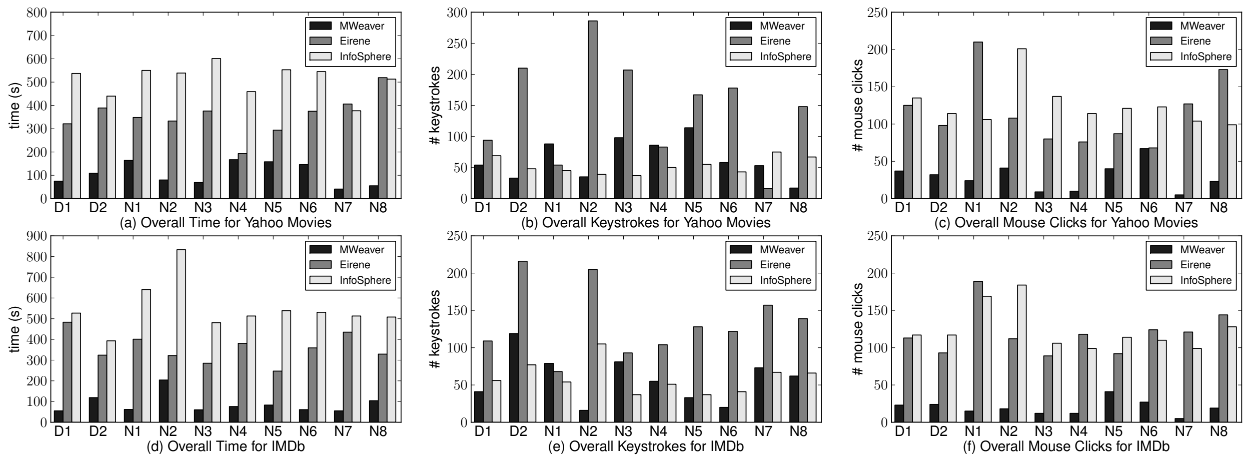


Figure 3.10: The overall time, keystrokes and mouse clicks for completing the mapping task on Yahoo Movies and IMDb. D1 and D2 are database experts. N1-N8 are non-technical users.

tools and show that our sample-driven tool enables an average user to complete a schema mapping task in just *1/5-1/4th* the overall time required by the other tools. Second, we show, with a synthetic mapping task workload, that the sample-driven approach is able to find the goal mapping with just about *two rows* of samples.

The sample search described in Section 3.3.1 is intuitively expensive since the search space is the whole source database instance. However, it relies on fast cooperation between the user and the system, so any serious computational delay could render the tool unusable. We conduct a performance experiment and show that the *Tuple Path Weaving* algorithm, TPW, is efficient enough to underpin our sample-driven tool. Indeed, we show that TPW is able to find the candidate mappings in seconds, in a 500MB sized database. In contrast, a naive approach constructed in a traditional keyword search manner yields runtime up to hundreds of seconds, far more than a user of a sample-driven tool is likely to tolerate.

3.7.1 Implementation and Environment

Our implementation of a sample-driven tool, MWeaver, has a UI written in HTML and javascript that communicates via AJAX to a backend engine written in Java Servlet. We use two datasets in our experiments: Yahoo Movies and IMDb. The Yahoo Movies dataset is 500MB in size and contains 43 relations and 131 attributes. The IMDb dataset is 2GB in size and contains 19 relations and 57 attributes. Both datasets are stored in MYSQL 5.

We use the full-text search engine in MySQL to implement the approximate search query. We set PMNJ to two, which is sufficient for our goal mappings. All the experiments were run on a desktop machine with an Intel Core i7 860 @ 2.80GHz and 8GB RAM.

3.7.2 Usability

We compare the usability of MWeaver against two tools. The first is IBM InfoSphere Data Architect (Version: 7.5.3.0), which is a commercialized version of the Clio project, and serves as a typical representative of the state-of-the-art match-driven tools. The second tool, Eirene [10], is a schema mapping tool recently developed by Alexe, et al. to help users design schema mappings through a QBE-like interface. We design the following user study scenario to be simple so that it can be carried out with non-technical subjects.

Suppose a user is browsing her favorite movies on the Yahoo Movies/IMDb website, and finds two pages of particular interests: the movie information page, which lists various properties of a movie, and the personal information page, which displays the biography of the contributors (e.g., the writers and the actors). However, the user finds these pages overwhelming since they contain every piece of related information spanning from a one-hundred-line movie description to an actor's achievements in forty years. In fact, all she wants is just a small subset of all these messy attributes.

Assume the user is only interested in the release date, the production company and the director of the movie, we formalize the mapping task as follows. The source schema S_S includes the complete Yahoo Movies/IMDb Database schema, and the target schema S_T contains only one relation comprising the following attributes: **Movie**: the title of the movie, **ReleaseDate**: the release date of the movie, **ProductionCompany**: the company which produces the movie and **Director**: the director of the movie. The user is asked to develop a schema mapping that transforms the data under S_S to the new database with schema S_T, for both Yahoo Movies and IMDb. The goal mappings are depicted by mapping paths in Figure 3.11.

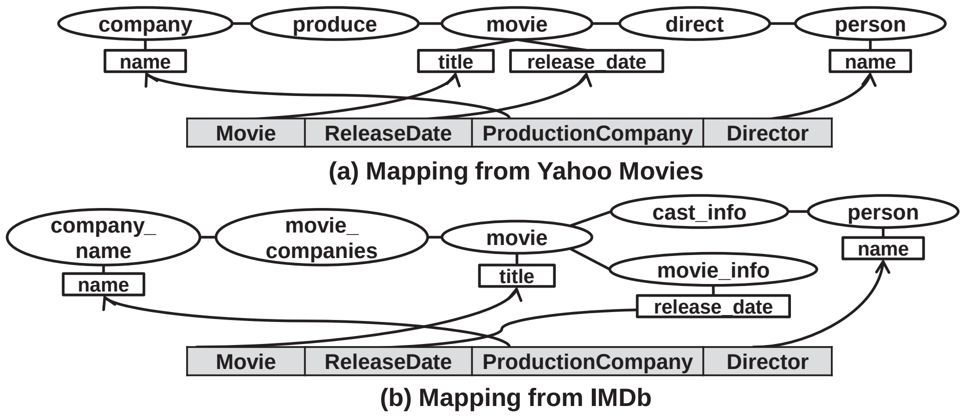


Figure 3.11: Task Schema Mappings: (a) Yahoo Movies, (b) IMDb.

We recruited eight non-technical subjects and two database experts for comparison. All of them were asked to complete the schema mapping task using MWeaver, IBM InfoSphere Data Architect and Eirene[9]. We recorded the overall time, keystrokes and mouse clicks to complete the task for each user with each tool, on both datasets. Because the latter two tools require substantial knowledge about the source schema, we provided complete technical support when the users were using these two tools. The results are shown in Figure 3.10. There was no substantial performance difference between database experts and end-users, or Yahoo Movies and IMDb datasets.

The results demonstrate that, on average, creating the mapping with MWeaver only needs 1/5 the overall time that required by IBM InfoSphere Data Architect, and 1/4 the overall time required by Eirene[10]. This difference is partly explained by the reduction in number of keystrokes and mouse clicks. The rest is attributed to the (not directly measurable) cognitive burden on the user in reasoning with unfamiliar source schema in the other tools.

While Eirene also asks the users to enter examples as in traditional QBE systems to design the mapping, MWeaver saves around half of the keystrokes required by Eirene. This is because MWeaver requires only target sample entry aided by auto-completion, while Eirene requires the user to fully specify the examples under both related source and target schema. Finally, MWeaver only needs 1/5 mouse clicks required by the other two tools, since the UI of MWeaver is simply a spreadsheet, which the users may naturally navigate through using traditional spreadsheet hot keys.

At the end of the user study, we asked each user how much she was satisfied

[9]We randomized the order to counterbalance the learning effect.
[10]All the above differences are statistically significant with p-values < 0.0002 according to the Mann-Whitney test.

with each tool and recorded a satisfaction score scaled from one (very dissatisfied) to five (very satisfied). MWeaver has an average score of 4.7, InfoSphere 2.7 and Eirene 3.45.

We also found that MWeaver only requires a few user-input samples to derive the goal mapping. However, the scale of the user study prevented us from collecting statistically significant data. Therefore, we focused on the Yahoo Movies dataset and constructed a synthetic mapping task workload containing tasks similar to the one used in the user study. Specifically, we defined three sets of task mappings. All the mappings in the same task set share the same relation path. The relation path has two, three and four joins for the three task sets. Each set contains four mappings, which vary in the target schema size from three to six.

For each mapping in each task set, we simulated user-input by repeatedly randomly sampling instances from a synthetic target database and fed them into MWeaver until the mapping is discovered. Each task was repeated for one hundred times and the average number of samples required is shown in Table 3.1. Recall that one row in the target has m samples, the results show that it only takes the user approximately two rows of samples to obtain the goal mapping.

Size of S_T (m)	3	4	5	6
Task Set 1	7.24	9.35	10.80	14.98
Task Set 2	5.08	8.50	11.55	16.18
Task Set 3	6.97	9.27	11.71	13.67

Table 3.1: The Average Number of Samples to Generate the Goal Mapping.

Finally, for each of the three task sets, we examined the number of candidate

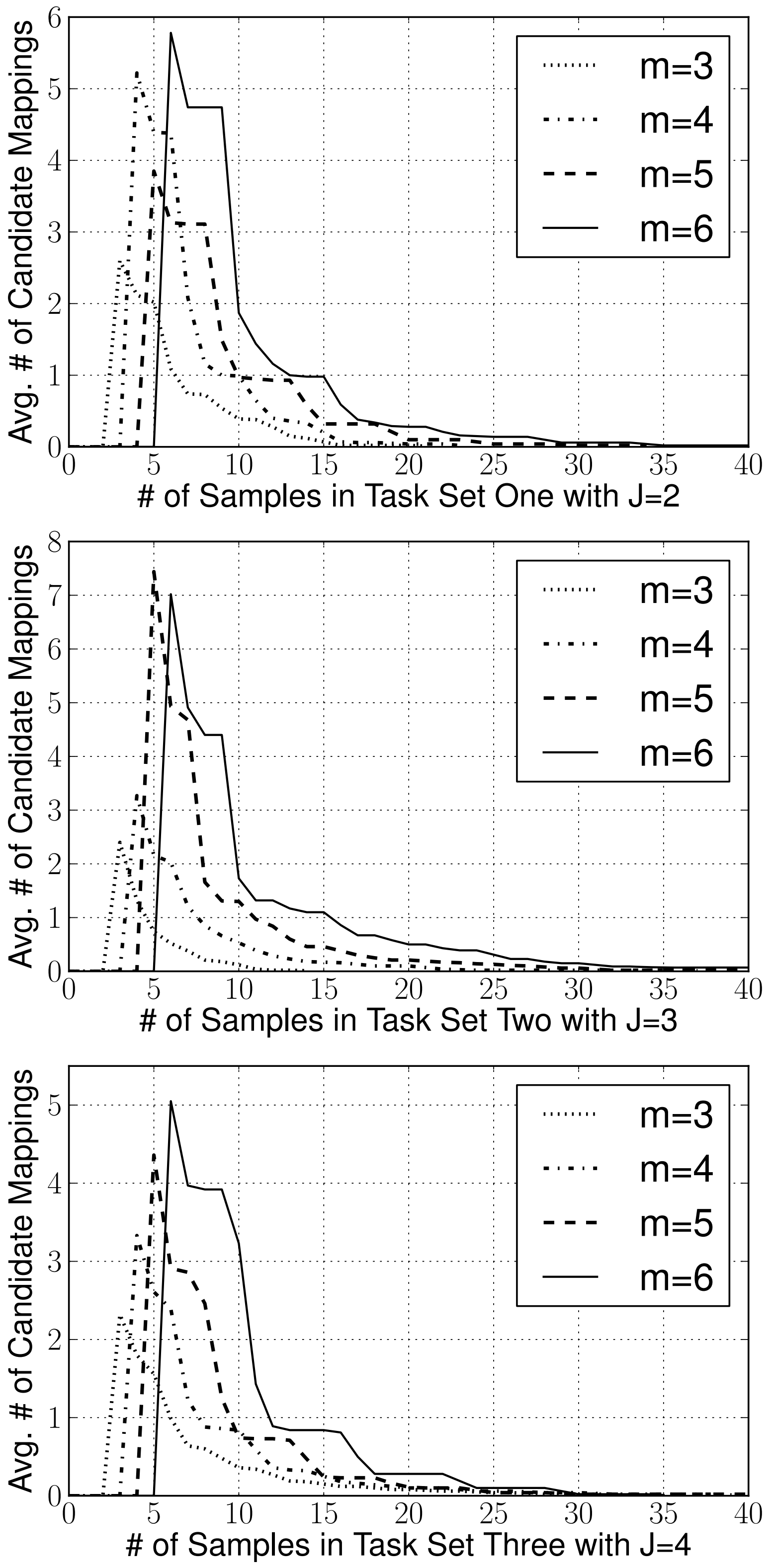


Figure 3.12: Average Number of Candidate Mappings w.r.t. the Number of Simulated Samples. J: number of joins in each mapping. m: the target schema size.

mappings (valid complete mapping paths) as the number of user-input samples increased. From the results shown in Figure 3.12, we observe that the number of candidate mappings drops dramatically as the number of user-input samples increases. Although in the worst case, the system may need about $8m$ samples to discover the goal mapping, the average is only about $2m$, where m is the target schema size.

3.7.3 Performance

We conducted performance experiments with the same set-up introduced above to demonstrate that TPW is efficient enough to meet interactive requirements.

We first measured the average response time for both searching and pruning to provide an overall sense of our system performance. The results shown in Table 3.2 demonstrates that MWeaver is able to respond to a user-input sample within 1s for searching and 50ms for pruning. In practice, the computation for the previous input is largely paralleled with the next user data entry so that the absolute waiting time observed by the user is very small.

Task Set	Size of S_T	3	4	5	6
1	Searching (ms)	534.35	655.03	639.49	577.25
	Pruning (ms)	34.27	24.46	35.13	58.54
2	Searching (ms)	177.98	363.32	407.69	450.91
	Pruning (ms)	27.23	40.63	58.24	62.20
3	Searching (ms)	305.89	442.78	761.69	817.38
	Pruning (ms)	32.53	24.46	40.24	51.58

Table 3.2: The Average Response Time for Searching and Pruning.

Next, we demonstrate that TPW is much more efficient compared to the naive approach derived directly from the schema-based keyword search techniques [50, 5]. To do so, we developed a naive algorithm which enumerated all the complete mapping paths (no matter valid or not) in the same way as the equivalent "candidate networks" are generated in [50], and validated them by executing an approximate search query translated from each of them. We performed both algorithms on the same workload described above. Table 3.3 shows the average overall time to complete the sample search for both algorithms. While TPW completed the search within 5 seconds on average, the time required by the naive algorithm grew dramatically. The naive algorithm failed beyond size 5 because the enumerated mapping paths exhausted the memory.

Task Set	Size of S_T	3	4	5	6
1	TPW (ms)	3735.48	3775.22	3008.52	3695.28
	Naive (ms)	35891.43	734319.25	–	–
2	TPW (ms)	578.47	1354.05	2043.77	2804.33
	Naive (ms)	1273.62	41976.94	–	–
3	TPW (ms)	1044.49	1674.66	3885.44	4727.86
	Naive (ms)	11644.93	388723.31	–	–

Table 3.3: The Average Search Time for TPW and the Naive Algorithm.

From this experiment, we also see that, even if the sample search problem is NP-hard, TPW is typically able to solve it in near-linear time. The intuition behind this huge improvement is that, invalid mapping paths are pruned away much earlier when smaller tuple paths are being processed in MWeaver. To offer a clearer insight into the performance experiment, we measured the total number of tuple paths (with various size) processed in TPW and the number of potentially valid complete mapping paths generated by the naive algorithm. Both numbers are compared with the exact number of valid mapping paths given a sample tuple. The result is shown in Table 3.4. Although the number of tuple path processed in our approach increased near-exponentially, it was still many fewer than the number of complete mapping paths that were generated and needed to be validated by the naive algorithm. In addition, the tuple paths were quickly processed in memory in TPW. In contrast, in the naive algorithm, the complete mapping paths had to be validated via expensive database accesses. This explains the performance difference observed in Table 3.3.

Task Set	Size of S_T	3	4	5	6
1	# Valid MP	2.67	5.05	4.52	6.00
	# TP Woven	15.46	207.40	719.67	3403.20
	# Naive MP	964.38	163634.45	–	–
2	# Valid MP	2.69	2.55	6.61	6.16
	# TP Woven	5.66	39.6	530.16	2008.39
	# Naive MP	35.31	967.25	–	–
3	# Valid MP	2.19	3.45	4.53	6.85
	# TP Woven	4.38	72.69	640.49	4149.37
	# Naive MP	318.36	10582.93	–	–

Table 3.4: Comparison between TPW and the Naive Algorithm. (MP=Mapping Path, TP=Tuple Path)

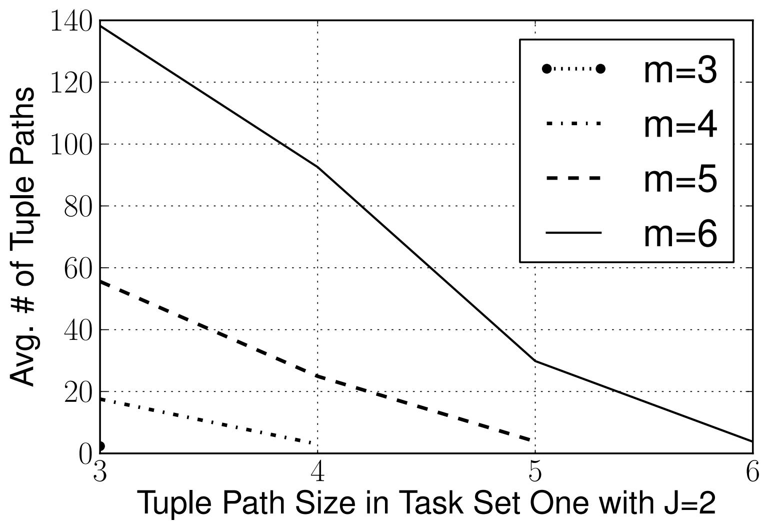
Our final experiment examined the average total number of tuple paths generated at each level in the TPW algorithm. The results are shown in Figure 3.13. We observed that the number of valid tuple paths decreases dramatically as the algorithm approached the full size of the target schema. This is because in a real-life dataset, the distributions of samples in different source attributes are relatively independent, making specific combinations unlikely as the size of the combination increases.

3.8 Conclusions

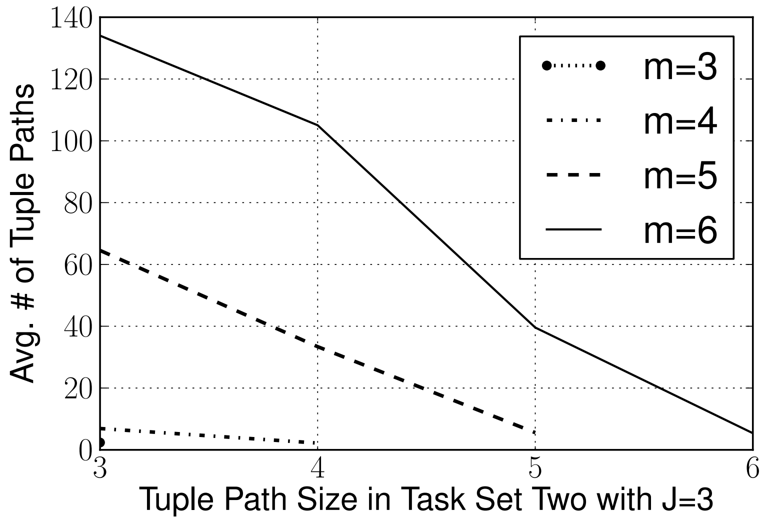
In this chapter, we have proposed a sample-driven schema mapping approach to facilitate the data integration tasks for end-users. While it is hard for end-users to understand the precise semantics of schemas and mappings, providing sample instances is much easier for them; we exploit this to design and prototype MWeaver, our sample-driven schema mapping tool, which renders the schema mapping tasks for end-users much more feasible.

We have studied the sample search problem in such a sample-driven approach, and proposed an algorithm to efficiently generate candidate mappings from user-input samples. Through user studies and simulated experiments on real-world datasets, we have demonstrated that MWeaver is more usable than existing schema mapping tools and our solution to the sample search problem is efficient enough to meet interactive requirements.

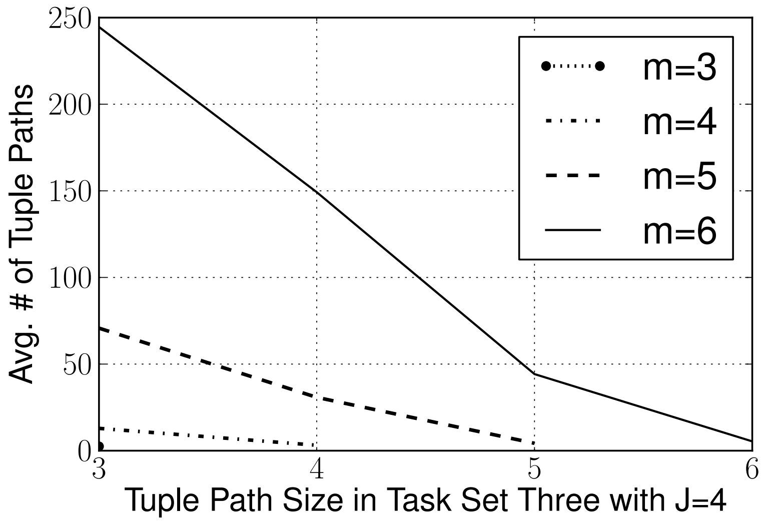
Our approach relies on the user-input to be roughly present in the source instance. In case the user-input is totally irrelevant to the source, it will invalidate previously generated correct mappings. We are studying on how to provide features that will automatically suggest relevant data and warn the user about irrelevant one. In this chapter, we primarily dealt with samples of string values. If the source contains many numerical attributes, a numerical sample may be contained by multiple source attributes, which will in turn degrade system performance. Also, since the number of tuple paths may grow rapidly w.r.t. the source database size, it is desirable to provide some insights into the scalability of our approach. Finally, we currently only support mappings in the form of project-join queries, which is a subset of GAV mappings. It would be interesting to study how to extend our approach to LAV and GLAV mappings. We plan to address these issues in the future work.



(a)



(b)



(c)

Figure 3.13: Average Number of Tuple Paths Generated at Each Level in TPW. J: number of joins in each mapping. m: the target schema size.

CHAPTER IV

Example-Driven Selection Condition Specification

In chapter III, we introduced a user-friendly sample-driven approach to deduce the desired schema mappings. However, we only supported mappings in the form of project-join queries. While these assumptions may be sufficient for simple user cases, they may not hold for general real-life scenarios. For instance, a natural requirement for an online market investigator would be to select several categories of products from certain time windows. Similar selection requirements may arise in various tasks spanning from database view creation, query construction to schema mapping.

While to directly specify the selection criteria is hard to end-users, they have no problem providing some examples of what they wish to select. Based on this intuition, we propose an approach to derive selection conditions via user-input examples. On the other hand, examples provide limited information which may not lead to the exact selection condition desired by the user. As a result, we further augment our approach for the user to easily revise the example-driven selection conditions.

In this chapter, we formalize the problem of example-driven selection condition specification and fold the above ideas into a two-phase solution. In the first phase, we automatically derive an initial selection condition according to a few user-input examples. In the second phase, we direct the user to an expressive representation of the initial condition, on which the user is free to revise the condition via an algebra consisting of direct manipulation operators. We show via a simulated user study and synthetic experiments that, we are able to derive high-quality initial selection conditions from just a few user-input examples, and our algorithms are efficient and scalable.

4.1 Introduction

In chapter III, we introduced a user-friendly sample-driven approach to deduce the desired schema mappings. As a starting point, we assumed the target is a single flat relation and the mapping is in the form of a project-join SQL query. While these assumptions may be sufficient for simple user cases, they may not hold for general real-life scenarios. For instance, a natural requirement for an online market investigator would be to select several categories of products from certain time windows. Similarly, when integrating multiple protein datasets, a biologist would like to select only the proteins related to certain pathways. These kind of selection requirements may come from not only schema mapping tasks, but also tasks of database view creation and query construction.

To offer user-friendly selection in these scenarios is a non-trivial problem, since the selection logic may be complicated in itself such that the end-user is not able to directly specify it in formal languages, even if she precisely knows the selection logic. This is illustrated by the following simplified user case.

Example 1. Imagine an end-user, Jane, who wishes to buy a used car. She has an access to an used car database, but is not interested in every car in the repository. Say she wants to find all the new cars no earlier than 2010 with a sedan body style. Also, because she is a big fan of Japanese cars, she would like to consider older Japanese cars too. Say the selection she has in mind is “sedan no earlier than 2010 or Japanese sedan no earlier than 2005”. Of course, a DBA is able to express this condition in formal boolean logic: $BodyStyle = \text{Sedan} \wedge (Year \geq 2010 \vee Country = \text{Japan} \wedge Year \geq 2005)$. However, the end-user is unlikely to be able to make this precise expression.

Although the user may specify the selection criteria via a traditional forms-based interface, the interface may not be sufficient in cases where the complexity of the desired selection logic exceeds the expressive power of such an interface. For instance, suppose Jane has access to some kind of forms-based query interface on top of the user-car dataset. If the form has a dropdown list for “Year no Earlier Than” and another dropdown list for “Body Style”, she has no trouble specifying “sedan no earlier than 2010”. But, what if there is no corresponding filter for “Body Style” (as in car.com)? Does Jane just give up the selection criteria on boy style? Also, how to specify the disjunction of general sedan and Japanese sedan? Traditional forms-based interface falls short under such scenarios.

On the other hand, without the aid from an abstract selection logic, the user would have to manually select all desired data points, which is impractical if such data points are many. For example, there may be thousands of used cars which satisfy Jane's selection condition, but she is unable to manually label them all. To enable the end-users to easily specify such kind of selection conditions, we need a selection framework which offers sufficient expressiveness and facilitates complex selection logic specification with as little manual burden as possible.

Example-Driven Selection Condition Specification. While the end-users are unable to expand the expressiveness of a given forms-based interface or directly spell out the selection condition in formal logic, she is most likely able to provide *examples* of what she wants to select. Even if the user is at the very beginning phase of specifying the selection condition and has difficulty specifying the condition from scratch, she may have no problem providing some first-impression examples. By using these examples, we could derive an initial estimate of the user-desired selection condition.

For instance, in the running example, even if Jane is not able to write the selection condition in formal boolean logic on the first spot, she would be able to provide examples such as "Ford Focus 2010" and/or "Honda Accord 2005". From these examples we could generate an initial estimate of the user-desired condition, such as "Ford cars newer than 2010 or Honda cars newer than 2005".

However, although the initial estimate may be close to the desired condition, it may not be an exact match. While an intuitive solution would be to allow the user to continue with more examples to refine the current estimate, we note that our estimate does not necessarily converge to the user-desired condition with respect to the number of user-input examples. For instance, even if the user have exhaustively entered every car in the dataset that is either a sedan newer than 2010 or a Japanese sedan newer than 2005, our best estimate may be "any sedan newer than 2005", which is still more general than the user-desired condition. Consequently, we further introduce an expressive representation of the selection condition, on which the user may easily refine the initial estimate via an algebra consisting of direct manipulation operators.

We fold these ideas into an approach named *example-driven selection condition specification*. The approach works in two phases. During the first phase, when the user is not confidant about directly specifying the selection condition from scratch, we ask the user to provide some example data points she wants to see in the output. Based on these examples, we automatically derive an initial selection condition which will select the user-input examples in additional to other data points the user likely desires. During the second phase, the user is free to revise the initial condition via an expressive representation using a direct manipulation algebra until she reaches the desired condition.

Contributions and Chapter Outline. We summarize the main contributions of this chapter as follows:

- We formalize the problem of selection condition specification under the example-driven scenario, with fair expressiveness assumptions.
- We introduce a representation model to visualize each valid solution to the specification problem by a set of manipulatable example groups, which we call *R-units*.
- We give an algorithm for automatically deriving an initial representation of

the selection condition based on the user-input examples.

- We design and implement a direct-manipulation algebra based on the representation, and prove it to be sound and complete.
- We demonstrate that we are able to derive high-quality initial selection conditions from a few user-input examples from a simulated user study, and show that our algorithms are efficient and scalable.

The rest of this chapter is organized as follows. We formalize the example-driven selection condition specification problem in Section 4.2. Section 4.3 introduces our representation mechanism, which represent each valid solution to the specification problem as a set of example groups. In Section 4.4, we describe our algorithms to give birth to an initial representation based on user-input examples. Section 4.5 introduces our direct-manipulation algebra, using which the user can easily customize the selection condition by modifying each example group in the representation. We present our experimental results in Section 4.6 before concluding this chapter in Section 4.7.

4.2 Problem Formalization

Before introducing our representation, we first formalize the example-driven selection condition specification problem. Literally, the problem means to derive a user specification of selection conditions from user-input examples. However, in order to formalize the problem, we have to first define the family of conditions that we allow the user to specify.

This is not a straightforward task. On one hand, the family of selection conditions should be expressive enough. Otherwise, the user could simply fulfill the task via forms-based interfaces or explicitly write them down in formal logic. On the other hand, our choice of the condition family should not be too expressive, in such a way that would overwhelm the user in the specification process. In this chapter, we consider it essential for the selection condition to be in the form of column-value comparison expressions (e.g., $Make = \mathrm{Honda}$) and any sequences of conjunctions or disjunctions of such expressions. We do not consider conditions containing column-column comparison, sub-queries or user-defined functions, because these are both semantically hard to comprehend and practically hard to specify for an end-user. As a starting point, we also assume no negation.

Furthermore, we assume every column is categoric rather than numeric. To make this assumption practical, we can always categorize numeric values into a set of categories with reasonable granularity. For instance, in the used car scenario, we can group every five years into one category, so we have time intervals such as $[2000 - 2005)$, $[2005 - 2010)$ and $[2010 - Present)$. When the user enters a value for the year, we can implicitly match it to the corresponding category without ambiguity. Having this categoric assumption, we further assume that each column-value comparison expression in our selection condition is an equality predicate. This is because, greater than (>), less than (<), greater than or equal to ($\geq$) and less than or equal to ($\leq$) can be rewritten (approximately, according the categorization granularity) in the form of disjunction of such equality predicates. For instance, $Year > 2005$ can be rewritten as $Year = [2005 - 2010) \vee Year = [2010 - Present)$.

Given these assumptions, we define a normal form for our selection condition family.

Definition 1 (Relational Disjunctive Normal Form (RDNF)). Given a relational schema $\mathrm{S} = (A_1, A_2, ..., A_N)$, where A_i is an attribute, a relational disjunctive normal form (or RDNF for short), is a disjunction of conjunctive clauses, each of which has the form $\mathrm{P}_1 \wedge \mathrm{P}_2 \wedge ... \wedge \mathrm{P}_N$, where $\mathrm{P}_i = \textbf{True}|(A_i = \mathrm{V}_i), i = 1..N$, and V_i is some value in the domain of A_i. We call each such conjunctive clause a *relational conjunction* (RC), and say it has a predicate on A_i, if $\mathrm{P}_i = (A_i = \mathrm{V}_i)$. We denote the relational disjunctive normal form and the relation conjunction by RDNF_S and RC_S, respectively, and say they have schema S.

We note that, any selection condition in relational disjunctive normal form is expressive enough in the sense that, any sequence of conjunctions or disjunctions of column-value equality predicates can be expressed in RDNF.

Theorem 4.2.1 (RDNF Expressiveness). Let a predicate $\mathrm{P} = \textbf{True}|(A_i = \mathrm{V}^\mathrm{P}), i = 1..N$. Let a sentence be recursively defined as $\mathrm{S} = \mathrm{P}|(\mathrm{S}_1 \wedge \mathrm{S}_2)|(\mathrm{S}_1 \vee \mathrm{S}_2)$. Any S can be expressed in the relational disjunctive normal form.

Proof: We prove it by mathematical induction. Given a sentence S, we define its order to be the number of $\wedge$ or $\vee$ in it, and denote it by $\mathrm{O(S)}$. The base case is trivial, where $\mathrm{O(S)} = 0$ (or $\mathrm{S} = \mathrm{P}$). For the inductive step, where $\mathrm{O(S)} = n, n > 0$, we pick the $\wedge$ or $\vee$ of the highest precedence, and split the sentence into S_1 and S_2. According to the definition, $\mathrm{O}(\mathrm{S}_1) + \mathrm{O}(\mathrm{S}_2) = n - 1$. Since $\forall \mathrm{S}, \mathrm{O(S)} \geq 0$, we must have $0 \leq \mathrm{O}(\mathrm{S}_1), \mathrm{O}(\mathrm{S}_2) \leq n - 1$. According to the inductive hypothesis, S_1 and S_2 can be expressed in RDNF.

Let $\mathrm{S}_1 = \mathrm{Q}_1 \vee \mathrm{Q}_1 \vee ... \vee \mathrm{Q}_K$, where $\mathrm{Q} = \mathrm{P}^i_1 \wedge \mathrm{P}^i_2 \wedge ... \wedge \mathrm{P}^i_N, i = 1..K$. Let

$\mathrm{S}_2 = \mathrm{Q'}_1 \vee \mathrm{Q'}_1 \vee ... \vee \mathrm{Q'}_M$, where $\mathrm{Q'} = \mathrm{P'}^j_1 \wedge \mathrm{P'}^j_2 \wedge ... \wedge \mathrm{P'}^j_N, j = 1..M$. If S is split on $\vee$, the inductive step is trivially proved by expressing S as $\mathrm{Q}_1 \vee \mathrm{Q}_1 \vee ... \vee \mathrm{Q}_K \vee \mathrm{Q'}_1 \vee \mathrm{Q'}_1 \vee ... \vee \mathrm{Q'}_M$. If S is split on $\wedge$, according to the distribution rule, $\mathrm{S} = \bigvee_{i=1..K,j=1..M} \mathrm{Q}_i \wedge \mathrm{Q'}_j = \bigvee_{i=1..K,j=1..M} \bigwedge_{l=1..N} \mathrm{P}^i_l \wedge \mathrm{P'}^j_l$. Let $\mathrm{P''} = \mathrm{P}^i_l \wedge \mathrm{P'}^j_l$. If $\mathrm{P}^i_l = \mathrm{P'}^j_l = \mathbf{True}$, $\mathrm{P''} = \mathbf{True}$. If only $\mathrm{P}^i_l = \mathbf{True}$, $\mathrm{P''} = \mathrm{P'}^j_l$, and vice versa. Otherwise, $\mathrm{P}^i_l = (A_l = \mathrm{V})$ and $\mathrm{P'}^j_l = (A_l = \mathrm{V'})$. If $\mathrm{V} = \mathrm{V'}$, $\mathrm{P''} = \mathrm{P}^i_l = \mathrm{P'}^j_l$. If $\mathrm{V} \neq \mathrm{V'}$, $\mathrm{P''} = \mathbf{False}$, thus $\mathrm{Q}_i \wedge \mathrm{Q'}_j = \mathbf{False}$ and can be omitted. In any case, S is still in RDNF. According to induction, S of any order can be expressed in RDNF. $\square$

Given a selection condition C in $\mathrm{RDNF_S}$ and a tuple t with schema S, we can view C as a boolean function and use it to evaluate t. We denote the evaluation by $\mathrm{C(t)}$. We say t *satisfies* C if and only if $\mathrm{C(t)} = \mathbf{True}$. Similarly, given a relation $\mathbf{R}$ with schema S, we say $\mathbf{R}$ *satisfies* C, or C is *valid* on $\mathbf{R}$, if and only if $\forall \mathrm{t} \in \mathbf{R}$, $\mathrm{C(t)} = \mathbf{True}$. Based on this, the selection operation can be defined as: $\sigma_{\mathrm{C}}(\mathbf{R}) \triangleq \{\mathrm{t} | \mathrm{t} \in \mathbf{R} \wedge \mathrm{C(t)}\}$. Obviously, $\sigma_{\mathrm{C}}(\mathbf{R}) = \mathbf{R}$, if $\mathbf{R}$ satisfies C. Having these notions, we define our problem as follows.

Definition 2 (Example-Driven Selection Condition Specification). Given a relation $\mathbf{R}$ which consists of user-input example tuples and has schema S, the example-driven selection condition specification problem is to *specify* a selection condition C in $\mathrm{RDNF_S}$, which is valid on $\mathbf{R}$.

Note that, the problem can be further divided into two subproblems. The first is to find *a* selection condition that is valid on the user-input examples, and the second is to *specify* such a selection condition. These two subproblems are twisted in such a way that, because there may be multiple valid selection conditions, the specification is also responsible to pick the *desired* one. However, only the user knows the ground truth. The best we can do is to find all the candidate valid conditions given the current set of user-input examples, and provide a reasonable mechanism for the user to browse through these candidates and choose the one she wants. In the following section, we describe how we represent each valid selection condition in conjunction with the user-input examples, and how we enable the user to navigate through various valid conditions.

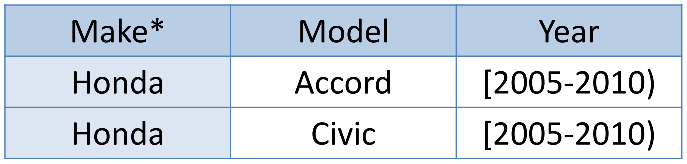
4.3 R-unit Representation

Given a relation $\mathbf{R}$ of user-input examples and a selection condition C in $\mathrm{RDNF}_{\mathrm{S}(\mathbf{R})}$ (not necessarily valid on $\mathbf{R}$), here we describe how we represent C with the examples.

Since C is in RDNF, we can decompose it as $\mathrm{C} = \bigvee_{i=1}^{M} \mathrm{C}_i$, where C_i is in the form of a relational conjunction $\bigwedge_{j=1}^{N} \mathrm{P}_j^i$, and $\mathrm{P}_j^i = \mathbf{True} | (A_j = \mathrm{V}_j^i)$. If we apply both sides of the first equation in a selection operation on $\mathbf{R}$, we have $\sigma_{\mathrm{C}}(\mathbf{R}) = \sigma_{\bigvee_{i=1}^{M} \mathrm{C}_i}(\mathbf{R}) = \bigcup_{i=1}^{M} \sigma_{\mathrm{C}_i}(\mathbf{R})$. This suggests that, we can represent a condition C by a set of relations, each of which corresponds to the result of applying a selection with condition C_i on $\mathbf{R}$. We call each such result a *disjunct relation* and denote it by $\mathbf{R}_i$.

While $\mathbf{R}_i$ provides an intuition of the corresponding C_i by presenting to the user the example tuples that satisfy C_i, it does not necessarily determine C_i. This is because, first, any generalization of C_i must be satisfied by $\mathbf{R}_i$. It is also possible that the artifact of the user-input $\mathbf{R}$ makes $\mathbf{R}_i$ satisfy a more strict version of C_i, just by chance.

Example 2 (Disjunct Relation). Suppose $\mathrm{C}_1 = (Make = \mathrm{Honda})^1$, the corresponding $\mathbf{R}_1$ is shown in Figure 4.1. We see that, the two example tuples are not enough to determine C_1. For instance, if we loosen C_1 to $\mathrm{C}'_1 = \mathbf{True}$, $\mathbf{R}_1$ trivially satisfies C'_1. On the other hand, even if we strengthen C_1 to $\mathrm{C}''_1 = (Make = \mathrm{Honda} \wedge Year = [2005, 2010))$, $\mathbf{R}_1$ still satisfies C''_1, since the example tuples happen to agree on the attribute $Year$.



Make*	Model	Year
Honda	Accord	[2005-2010)
Honda	Civic	[2005-2010)

Figure 4.1: An Example Disjunct Relation (and an R-unit)

To resolve this ambiguity, we augment each disjunct relation with a boolean tuple of the same schema. The tuple has a value **True** on attribute A_j, if and only if the corresponding P_j in C_i is an equality predicate (not **True**). We call the augmented disjunct relation an **R-unit**, and give the formal definition as follows.

Definition 3 (R-unit). Given a relation $\mathbf{R}$ of user-input example tuples and a selection condition C_i in RC, where $\mathrm{C}_i = \bigwedge_{j=1}^{N} \mathrm{P}^i_j$ and $\mathrm{P}^i_j = \mathbf{True} | (A_j = \mathrm{V}^i_j)$, an *R-unit* is a disjunct relation $\mathbf{R}_i$ augmented with a boolean tuple t_i, where $\mathbf{R}_i = \sigma_{\mathrm{C}_i}(\mathbf{R})$ and $\mathrm{t}_i[A_j] = \mathbf{True}$ iff $\mathrm{P}^i_j = (A_j = \mathrm{V}^i_j)$. We denote the R-unit by $\mathbf{R}^*_i$.

In practice, we represent the boolean tuple by associating a flag with each attribute in the r-unit's schema. For example, in Figure 4.1, we append a *star* beside $Make$ in the table header to denote that the boolean tuple is **True** on

[1] Hereafter we omit Ps that are **True**, if the context is clear.

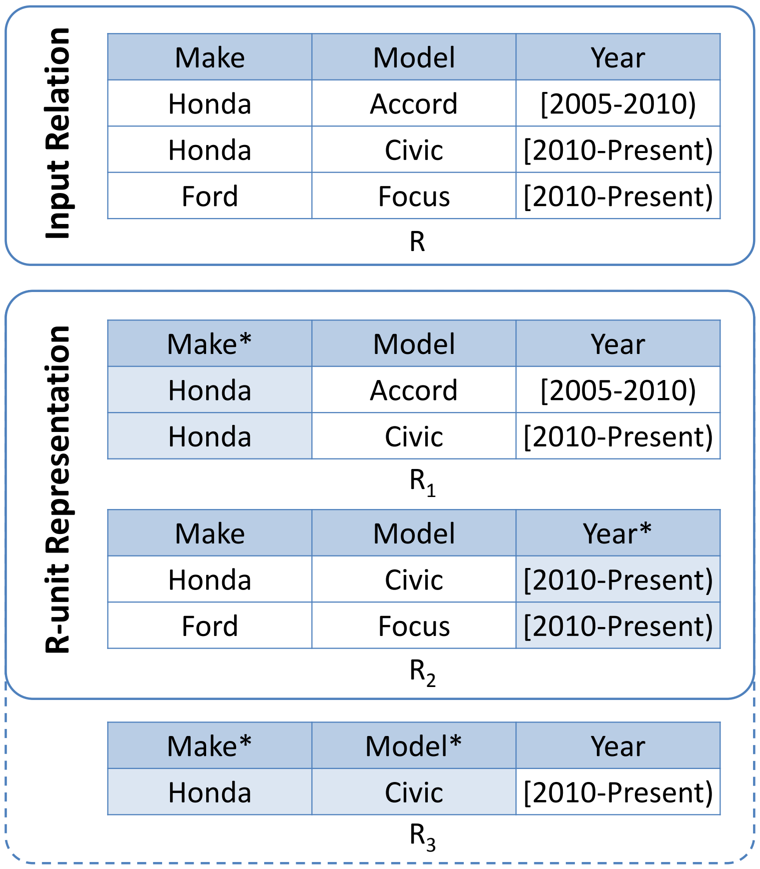
that attribute (or, in other words, the attribute is associated with a non-trivial predicate).

Given an R-unit $\mathbf{R}_i^* = (\mathbf{R}_i, \mathrm{t}_i)$, it uniquely determines a selection condition C_i in RC. To construct C_i from $\mathbf{R}_i$ is straightforward. For $j = 1..N$, if $\mathrm{t}_i[A_j] = \mathbf{False}$, $\mathrm{P}_j^i = \mathbf{True}$. Otherwise, $\mathrm{P}_j^i = (A_j = \mathrm{V}_j^i)$, where $\mathrm{V}_j^i = \mathrm{t}_i[A_j], \mathrm{t} \in \mathbf{R}_i$. ($\mathrm{t}$ can be any tuple from $\mathbf{R}_i$, since all tuples must agree on the value of attribute A_j according to the definition.) On the other direction, it is straightforward that C_i uniquely determines $\mathbf{R}_i^*$ given $\mathbf{R}$. Hereafter, we use $\mathbf{R}_{\mathrm{C}_i}$ to denote the R-unit corresponding to a given C_i in RC. We use $\mathrm{Pr}(\mathbf{R}_{\mathrm{C}_i})$ to denote the set of all attributes on which t_i is true.

Now we can use a set of of R-units to represent the overall selection condition C. Intuitively, for each C_i, we represent it with a corresponding R-units $\mathbf{R}_i^*$. When we display the set of R-units in a whole to the user, the union of these R-units represents the disjunction of C_i, which is C.

Example 3 (The R-unit Representation). Suppose we have user-input example tuples as shown in $\mathbf{R}$ in Figure 4.2, and the desired selection condition is $\mathrm{C} = (Make = \mathrm{Honda}) \vee (Year = [2010 - \mathrm{Present}])$, an R-unit representation is shown in the bottom solid box in Figure 4.2. Note that the Civic tuple appears in both R-units. This is essential since it implies that the tuple is selected not only because it is a Honda car, but also because its made after 2010. In other words, we cannot remove the tuple from either R-unit.

Although each R-unit precisely represents a selection condition in RC, not any combination of the R-units form a “good” overall representation. Here we propose several properties that a good representation should satisfy.


Input Relation
R-unit Representation

Make	Model	Year
Honda	Accord	[2005-2010)
Honda	Civic	[2010-Present)
Ford	Focus	[2010-Present)

R

Make*	Model	Year
Honda	Accord	[2005-2010)
Honda	Civic	[2010-Present)

R_1

Make	Model	Year*
Honda	Civic	[2010-Present)
Ford	Focus	[2010-Present)

R_2

Make*	Model*	Year
Honda	Civic	[2010-Present)

R_3

Figure 4.2: An R-unit Representation

First of all, the R-unit representation must be *complete*, in a sense that each user-input example tuple in $\mathbf{R}$ must appear in at least one R-unit $\mathbf{R}_i^*$. Since an R-unit cannot contain tuples not existing in $\mathbf{R}$, the completeness is equivalent to say $\mathbf{R} = \bigcup_{i=1}^{M} \mathbf{R}_i$.

Second, the R-unit representation must be *non-redundant*. There are two notions of redundancy here. First, a R-unit $\mathbf{R}_i^*$ may be redundant in a given set of R-units, if each example tuple in $\mathbf{R}_i^*$ is covered by some other R-unit in the set. For instance, if we add a new R-unit $\mathbf{R}_3^*$ in the representation in Example 3, as shown in the dashed box in Figure 4.2, it is obvious that $\mathbf{R}_3^*$ is redundant, because any tuple in it is already covered by $\mathbf{R}_1^*$ or $\mathbf{R}_2^*$.

The redundancy may occur not only at relation level, but also at attribute level, due to data dependencies. Consider, a *functional dependency* (or FD for short) from $Model$ to $Make$. Because the value of $Model$ uniquely determines the value of $Make$, a selection condition of form $Model = \mathrm{V}_1 \wedge Make = \mathrm{V}_2$ makes little sense. In other words, the predicate $Make = \mathrm{V}_2$ is redundant in this case. Reflected in the representation, we can always turn off the flag associated with the determined attribute in the corresponding R-unit, without changing the content in the R-unit.

In an example-driven scenario, we should not assume an end-user to comprehend, recognize and resolve such redundancies in the logic level. Instead, we should automatically avoid these redundancies in the representation for the user. We also note that, we do not try to make an R-unit representation *minimal*. This is because a cost function would be completely user-specific in the example-driven scenario. Rather than "guess" what the optimal representation is, we would like to invite the user to explore and define her best choice.

We say a R-unit representation is *well-formed*, if it is complete and non-redundant. It is straightforward that a well-formed R-unit representation uniquely specifies a valid selection condition on $\mathbf{R}$. In this chapter, our goal is to provide the user with a well-formed R-unit representation, which the user is free to modify. To do this, we have two challenges: 1) How do we derive an initial well-formed representation, given a set of user-input examples? and 2) How do we allow the user to manipulate this representation, while guaranteeing it is well-formed?

In Section 4.4, we formalize the representation birthing problem and give an algorithm to derive the initial R-unit representation. In Section 4.5, we propose an R-unit algebra, with which the user may navigate through various well-formed R-unit representation.

4.4 Birthing a Representation

Here we study the problem of giving birth to an initial R-unit representation, given a relation $\mathbf{R}$ of user-input example tuples. We first note that, the set of all possible R-units in the representation is finite given $\mathbf{R}$ is finite. This is because, given an attribute A_j, it can only take a finite set of distinct values given $\mathbf{R}$. Consequently, C_i (in RC) can only have finite variations, which map to a finite set of $\mathbf{R}_{\mathrm{C}_i}$. We name all these R-unites *candidate* R-units, and denote the set of such R-units by $\mathbf{D}$. Formally, the representation birthing problem can be defined as the following.

Definition 4 (R-unit Representation Birthing Problem). Given a relation $\mathbf{R}$ of user-input example tuples, the R-unit representation birthing problem is to find a non-redundant set of candidate R-units $\mathbf{B} \subseteq \mathbf{D}$, such that $\bigcup_{\mathbf{R}^* \in \mathbf{B}} \mathbf{R}^* = \mathbf{R}$.

We can further divide the representation birthing problem into two subproblems: 1) how to generate the complete set of candidate R-units $\mathbf{D}$ and 2) how to select a subset of the candidate set, which can cover $\mathbf{R}$. In this section, we first describe our solutions to these two subproblems. We then introduce an optimization which does not require the generation of all the candidate R-units.

4.4.1 Candidate Generation

Here we examine the problem of generating all the candidate R-units in $\mathbf{D}$. We first give a more formal definition of the complete candidate R-units set $\mathbf{D}$.

To do this, we adopt the idea of *attribute partition* introduced in the literature of minimal FD induction [51]. Specifically, given a set of attributes $\overline{X}$, we can group tuples in $\mathbf{R}$ which have identical values on each $A \in \overline{X}$. This results in an *attribute partition* of $\mathbf{R}$, which we denote as $\mathbf{P}_{\overline{X}}$. We note that, any R-unit $\mathbf{R}_{\mathrm{C}_i}$ with $\Pr(\mathbf{R}_{\mathrm{C}_i}) = \overline{X}$ can only have some partition group in $\mathbf{P}_{\overline{X}}$ as its disjunct relation.

For simplicity, hereafter we refer these R-units, their disjunct relations and the corresponding partition groups interchangeably (since $\overline{X}$ uniquely determines the boolean tuple for each $\mathbf{R}_{\mathrm{C}_i}$). We call $\mathbf{P}_{\overline{X}}$ an R-unit partition, and denote $\mathbf{P}_{\overline{X}} = \{\mathbf{R}^i_{\overline{X}}\}$, $i = 1..L_{\overline{X}}$, where each $\mathbf{R}^i_{\overline{X}}$ is both an R-unit and a partition group. As a special case, $\mathbf{P}_\emptyset = \{\mathbf{R}\}$. Obviously, $\mathbf{D} = \bigcup_{\overline{X} \in \mathcal{P}(\mathrm{S}(\mathbf{R}))\setminus\emptyset} \mathbf{P}_{\overline{X}}$, where $\mathcal{P}$ stands for powerset and $\emptyset$ is removed from the powerset to omit the trivial case of covering with $\mathbf{R}$.

Given the user-input example relation $\mathbf{R}$, to generate all the candidate R-units is an expensive task, because we have to partition $\mathbf{R}$ on each non-empty subset of $\mathrm{S}_{\mathbf{R}}$, where we have $2^N - 1$ such subsets. Fortunately, instead of computing $\mathbf{P}_{\overline{X}}$ from scratch, Y. Huhtala et al. [51] suggests an efficient way of computing $\mathbf{P}_{\overline{X}}$ by taking a *partition product* of $\mathbf{P}_{\overline{X}\setminus A}$ and $\mathbf{P}_A$. Having this operation, we can construct partitions from those on small set of attributes to those on larger set of attributes. To do this, we first borrow the concept of *attribute lattice* from [51], and define an important data structure we call *R-lattice*.

Definition 5 (R-lattice). Given a relation $\mathbf{R}$ of user-input example tuples, an *R-lattice* is a lattice $\mathbf{L}$ on $\mathrm{S}_{\mathbf{R}} = (A_1, A_2, ..., A_N)$. For each subset $\overline{X}$ of $\mathrm{S}_{\mathbf{R}}$, there is a corresponding lattice node $\mathbf{P}_{\overline{X}}$, which is an R-unit partition of $\mathbf{R}$ on $\overline{X}$. Given

$\overline{X}$, for any $A \in \mathrm{S}_{\mathbf{R}} \backslash \overline{X}$, there is an edge from $\mathbf{P}_{\overline{X}}$ to $\mathbf{P}_{\overline{X} \cup A}$. We group lattice nodes into *levels*, and denote them by $\mathbf{L}_i, i = 0..N$, where $\mathbf{L}_i = \{\mathbf{P}_{\overline{X}} | \left|\overline{X}\right| = i\}$.

ID	Make	Model	Year
1	Honda	Accord	[2005-2010)
2	Honda	Civic	[2005-2010)
3	Honda	Civic	[2010-Present)
4	Ford	Focus	[2010-Present)
5	Ford	Fusion	[2010-Present)

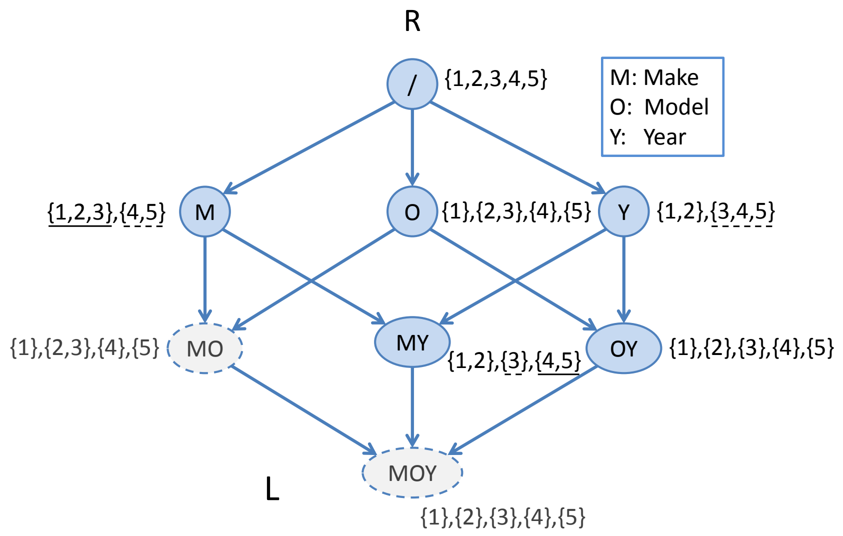


Figure 4.3: An R-lattice in the Running Example

For instance, Figure 4.3 shows a user-input relation $\mathbf{R}$ and its corresponding R-lattice $\mathbf{L}$, with each of its nodes labeled with the corresponding set of attributes and appended with the corresponding R-unit partition (presented as comma-separated sets of tuple IDs).

We say $\mathbf{P}_{\overline{X}}$ is a *parent* of $\mathbf{P}_{\overline{Y}}$, and $\mathbf{P}_{\overline{Y}}$ is a *child* of $\mathbf{P}_{\overline{X}}$, if there is an edge in the lattice from $\mathbf{P}_{\overline{X}}$ to $\mathbf{P}_{\overline{Y}}$ (e.g., in Figure 4.3, $\mathbf{P}_M$[2] is a parent of $\mathbf{P}_{MY}$). These definitions naturally extend to *ancestor* and *descendant*. Specifically, given a set of partitions $\mathbf{Q} = \{\mathbf{P}_{\overline{X}}\}$, we say $\mathbf{P}_{\overline{Y}}$ is a *pure descendant* of $\mathbf{Q}$, if $\overline{Y} \subset \bigcup\{\overline{X} | \mathbf{P}_{\overline{X}} \in \mathbf{Q}\}$ (e.g., $\mathbf{P}_{MY}$ is a pure descendant of $\{\mathbf{P}_M, \mathbf{P}_Y\}$, while $\mathbf{P}_{MO}$ or

[2]For simplicity, we abbreviate $\mathbf{P}_{\{Make\}}$ by $\mathbf{P}_M$, $\mathbf{P}_{\{Model\}}$ by $\mathbf{P}_O$, $\mathbf{P}_{\{Make,Year\}}$ by $\mathbf{P}_{MY}$, so on and so forth.

$\mathbf{P}_{MOY}$ is not).

Given an R-lattice, we can generate the complete set of candidate R-units $\mathbf{D}$ by traversing the lattice in a top-down manner. At $\mathbf{L}_1$, we partition $\mathbf{R}$ on each single attribute, and add all the R-units in these partitions to $\mathbf{D}$. At $\mathbf{L}_i, i = 2..N$, for each $\mathbf{P}_{\overline{X}}$ in $\mathbf{L}_i$, we choose any $A \in \overline{X}$, and compute $\mathbf{P}_{\overline{X}}$ by taking the partition product of $\mathbf{P}_{\overline{X} \setminus A}$ and $\mathbf{P}_A$, where $\mathbf{P}_{\overline{X} \setminus A}$ is already computed at $\mathbf{L}_{i-1}$ and $\mathbf{P}_A$ at $\mathbf{L}_1$. We collect all the R-units from the computed product and add them to $\mathbf{D}$. Upon completion, $\mathbf{D}$ must contain the complete set of candidate R-units (e.g., all sets of tuple IDs shown in the lattice in Figure 4.3).

4.4.2 R-unit Covering

Having the set of all candidate R-units generated, we have to choose a non-redundant subset of the full candidate R-unit set to cover $\mathbf{R}$. This problem is very similar to the set covering problem. In fact, we could directly apply a traditional set covering algorithm to solve the R-unit covering problem. Since the set covering problem is known to be NP-hard, it is usually solved by a greedy algorithm. Here we start with the same greedy algorithm as a naive solution to out R-unit covering problem.

The naive greedy algorithm in our scenario works as follows. We start by marking all the example tuples in $\mathbf{R}$ as uncovered. At each greedy search step, we iterate over all the candidate R-units in $\mathbf{D}$, and choose the “best” R-unit that covers the most uncovered example tuples. The tuples covered by the best choice are marked as covered at the end of each step. We repeat the process until all example tuples in $\mathbf{R}$ are covered, and return the set of selected

R-units.

While the naive greedy algorithm is simple and intuitive, we note that, it may be unsuitable for our R-unit covering problem, because of the following reasons. First, the naive algorithm may be very inefficient, because the size of the candidate pool can be very large (exponential to the size of $\mathbf{R}$). On the other hand, not all of these candidate R-units need to be considered for the best choice at each step. Moreover, the naive algorithm does not guarantee the result R-units in the cover to be non-redundant. We now describe our algorithm, **Smart R-cover**, which improves the efficiency and guarantees the result cover to be non-redundant.

Search Space Reduction

Although the number of all candidate R-units can be potentially large, not all of them need to be considered during the covering process. In fact, when we search for the R-unit that covers the most uncovered tuples, most likely we will find the “best” R-unit from $\mathbf{L}_1$, instead of the deeper part of the lattice. This is because, intuitively, R-unit partitions at the deeper part of the lattice are in general finer than those at a higher level. Consequently, R-units at the deeper part of the lattice tend to cover less example tuples. This suggests a possibility to reduce our search space for such best R-units. To illustrate this kind of search space reduction, we first give a formal definition of such best R-units.

Definition 6 (Best Choice). During any step in the covering process, given a set of R-units $\mathbf{T}$, we say an R-unit $\mathbf{R}^*$ is the *best choice* in $\mathbf{T}$, if $\mathbf{R}^* \in \mathbf{T}$ and $\mathbf{R}^*$

covers the most uncovered example tuples among the R-units in $\mathbf{T}$.

Note that, we may have multiple best choices in a given set, if they cover the same number of uncovered tuples. These best choices may from the same partition, or different partitions. We will discuss how we choose a single R-unit from the set of best choices shortly. Given the notion of best choices, we have the following theorem.

Theorem 4.4.1 (Best Choice). At any step in the covering process, let $\mathbf{E}$ be the set of best choices in $\bigcup\{\mathbf{P}_{\overline{X}}|\mathbf{P}_{\overline{X}} \in \mathbf{L}_1\}$, and $\overline{Y} = \bigcup\{\overline{X}|\mathbf{P}_{\overline{X}} \in \mathbf{L}_1 \wedge \exists \mathbf{R}^* \in \mathbf{E}, s.t. \mathbf{R}^* \in \mathbf{P}_{\overline{X}}\}$. Let $\mathbf{E}'$ be the set of best choices in $\mathbf{D}$, we have $\mathbf{E}' \supseteq \mathbf{E}$ and $\mathbf{E}' \backslash \mathbf{E}$ can only contain R-units in $\mathbf{P}_{\overline{Z}}$, where $\overline{Z} \subseteq \overline{Y}$ and $\left|\overline{Z}\right| > 1$.

Theorem 4.4.1 makes two statements. First, the local best choices in $\mathbf{L}_1$ of the lattice must be a subset of the global best choices in $\mathbf{D}$. Second, if there are more global best choices, they must be from the *pure descendant* partitions of the the set of first-level partitions where the local best choices come from.

As a special case, where the best choices at $\mathbf{L}_1$ are from only one partition, we have the following corollary of Theorem 4.4.1.

Corollary 4.4.1 (Best Choice). At any step in the covering process, let $\mathbf{E}$ be the set of best choices in $\bigcup\{\mathbf{P}_{\overline{X}}|\mathbf{P}_{\overline{X}} \in \mathbf{L}_1\}$. If $\exists \mathbf{P} \in \mathbf{L}_1, s.t. \mathbf{E} \subseteq \mathbf{P}$, $\mathbf{E}$ must also be the set of best choices in $\mathbf{D}$.

Corollary 4.4.1 says, if all the local best choices at $\mathbf{L}_1$ of the lattice belong to the the same partition, they must also be the complete set of global best choices.

Theorem 4.4.1 and Corollary 4.4.1 suggests a much more efficient way to search for the best choices for our greedy algorithm. At each step during the covering process, we can start by searching the local best choices at $\mathbf{L}_1$ of the lattice. These local best choices must be part of the global best choices. If all these best choices appear in the same partition, they must be the only global best choices, and we just pick one from them. Otherwise, say the best choices appear in several partitions, we only need to check those pure descendants of these first-level partitions for additional global best choices, and choose one from the union of these additional best choices and the local best choices.

So far we have discussed how to efficiently find the set of global best choices, but have been silent about how to pick a unique R-unit from these best choices. Since all the best choices cover the same amount of uncovered tuples, we based our preference on the number of tuples they cover that are already covered by previous selected R-units (i.e., the overlapping with the existing coverage). Intuitively, we prefer the best choice with a less overlapping, because the corresponding selection condition is more selective, and the result covering is less likely to be redundant. Moreover, given two best choices with the same amount of overlapping, preference will be given to the one at the deeper level of the lattice, since the corresponding selection condition is more specific.

Example 4 (Efficient R-unit Covering with Best Choices). Suppose we have a user-input example relation $\mathbf{R}$ as shown in the top part in Figure 4.3, here we illustrate how we search for the best choices. We start by searching for the best choices in $\mathbf{L}_1$, and get two of them: $\{1, 2, 3\}$ from $\mathbf{P}_M$ and $\{3, 4, 5\}$ from $\mathbf{P}_Y$. Since they are from two different partitions, it is possible that we have more best choices in the pure descendant ($\mathbf{P}_{MY}$ in this case) of these two partitions. We examine $\mathbf{P}_{MY}$ and find no more best choices, so the two local bet choices are also global. Since this is the first step, both R-units have no overlapping with the existing coverage. Thus we randomly pick one form them, say $\{1, 2, 3\}$, as the first covering R-unit.

Now the remaining set of tuples to be covered are $\{4, 5\}$. We repeat the greedy search step and find two best choices in $\mathbf{L}_1$: $\{4, 5\}$ from $\mathbf{P}_M$ and $\{3, 4, 5\}$ from $\mathbf{P}_Y$. Again, we dive into the pure descendant $\mathbf{P}_{MY}$ to check for more best choices. In this time, we find one more global best choice $\{4, 5\}$ from $\mathbf{P}_{MY}$. $\{3, 4, 5\}$ overlaps with the existing coverage by one tuple, so we omit it. Between $\{4, 5\}$ from $\mathbf{P}_M$ and $\{4, 5\}$ from $\mathbf{P}_{MY}$, we choose the latter since the corresponding selection condition is more specific. Now all tuples in $\mathbf{R}$ are covered and our process terminates. We highlight the columns on which the select R-units have predicates with different color for different R-units in Figure 4.3.

Avoiding Redundancies

In the previous section, we investigate an efficient R-unit covering algorith-m based on traditional greedy set covering. However, the cover of R-units generated by our algorithm is not guaranteed to be non-redundant. Here we describe how we further improve our algorithm to guarantee the result R-unit cover to be non-redundant.

As we have discussed in Section 4.3, there are two notions of redundancies: redundancies in the attribute (predicate) level and redundancies in the relation

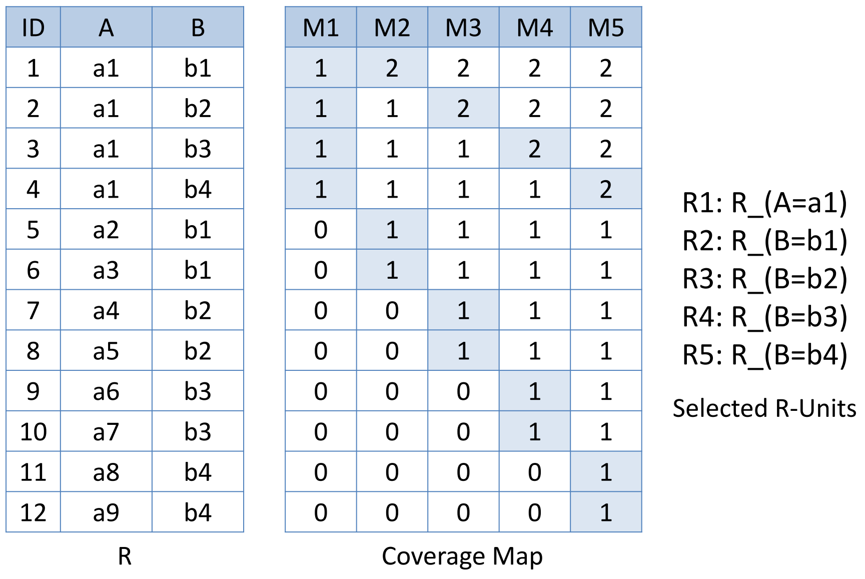
(R-unit) level. We start with the first kind of redundancies.

FD Redundancy The redundancies in the attribute level attributes to data dependencies. In this chapter, we focus on functional dependencies (FDs). For instance, in our running example we have a FD $Model \rightarrow Make$. Given this FD, an R-unit $\mathbf{R}_{(Make=\text{Honda} \wedge Model=\text{Accord})}$ makes little sense, since we know Accord is always made by Honda. As a result, the predicate $Make = \text{Honda}$ is redundant in this case. We can always remove it and replace the above R-unit with $\mathbf{R}_{(Model=\text{Accord})}$.

In fact, if we interpret this in our lattice vocabulary, we realize that, if we have a FD $\overline{X} \rightarrow A$, then for any $\overline{Y} \supseteq \overline{X}$, $\mathbf{P}_{\overline{Y}} = \mathbf{P}_{\overline{Y} \cup A}$. In other words, each R-unit in $\mathbf{P}_{\overline{Y} \cup A}$ is redundant, because we can replace it with an R-unit in $\mathbf{P}_{\overline{Y}}$ which has the same coverage and no predicate on A. Consequently, we can remove the whole $\mathbf{P}_{\overline{Y} \cup A}$ from the lattice before our covering procedure, without affecting the rest part of our algorithm. For example, since we have $Model \rightarrow Make$, we can safely remove $\mathbf{P}_{MO}$ and $\mathbf{P}_{MOY}$ from the lattice in Figure 4.3 (presented in circles with dashed outline).

As a special case, if $\overline{X}$ is a key, we have $\overline{X} \rightarrow A$ for any A in $\mathrm{S}(R) \backslash \overline{X}$. As a result, for any $\overline{X} \subset \overline{Y} \subseteq \mathrm{S}(R)$, we can safely remove $\mathbf{P}_{\overline{Y}}$ from the lattice at the beginning of our algorithm.

R-unit Redundancy The second type of redundancy occurs at the R-unit level. To recall, given the R-unit cover set $\mathbf{B}$, we say an R-unit from $\mathbf{B}$ is redundant, if each of its tuples are covered by some other R-units in $\mathbf{B}$. We say the set $\mathbf{B}$ is redundant, if it contains at least one redundant R-unit. The following example demonstrates how our algorithm generates a redundant R-unit cover set.



ID	A	B
1	a1	b1
2	a1	b2
3	a1	b3
4	a1	b4
5	a2	b1
6	a3	b1
7	a4	b2
8	a5	b2
9	a6	b3
10	a7	b3
11	a8	b4
12	a9	b4

R

M1	M2	M3	M4	M5
1	2	2	2	2
1	1	2	2	2
1	1	1	2	2
1	1	1	1	2
0	1	1	1	1
0	1	1	1	1
0	0	1	1	1
0	0	1	1	1
0	0	0	1	1
0	0	0	1	1
0	0	0	0	1
0	0	0	0	1

Coverage Map

R1: R_(A=a1)
R2: R_(B=b1)
R3: R_(B=b2)
R4: R_(B=b3)
R5: R_(B=b4)

Selected R-Units

Figure 4.4: A Example for R-unit Redundancy

Example 5. Consider a relation $\mathbf{R}$ as shown in the left part of Figure 4.4. According to our algorithm, the unique best choice at step one is $\mathbf{R}_{A=\mathrm{a}_1}$ (which covers 4 tuples). We select it as $\mathbf{R}_1$. In the subsequent steps, $\mathbf{R}_{B=\mathrm{b_i}}, i = 1..4$ are equally good, so we pick one at a time, and denote them as $\mathbf{R}_2 - \mathbf{R}_5$. We see that, even if $\mathbf{R}_1$ is redundant (since it is covered by $\mathbf{R}_2 - \mathbf{R}_5$), it was still selected at first, because it was the only global best choice at that time.

In order to eliminate such redundancies, we maintain a data structure called *cover map*, which is shown in the middle part of Figure 4.4. Intuitively, it records how many times a tuple in $\mathbf{R}$ is covered by R-units in the cover set $\mathbf{B}$. At any time, for any R-unit in $\mathbf{B}$, if all of its tuples are covered at least twice according to the cover map, we remove it from $\mathbf{B}$ and update the cover map accordingly.

Given a cover map $\mathbf{M}$ and a tuple $\mathrm{t} \in \mathbf{R}$, we use $\mathbf{M}(\mathrm{t})$ to denote the corresponding value in $\mathbf{M}$ for that tuple t. For a set of tuples $\mathbf{R}'$, we define

$\mathbf{M}(\mathbf{R}') \triangleq \{\mathbf{M}(\mathrm{t})|\mathrm{t} \in \mathbf{R}'\}$. We note, when we are going to add a new R-unit $\mathbf{R}^*$ to $\mathbf{B}$ at the end of each step, only the subset of $\mathbf{R}^*$ that overlaps existing coverage will possibly make an existing R-unit cover redundant. Specifically, we only need to check $\mathbf{M}(\mathbf{R}^* \cap \bigcup \mathbf{B})$. Only when there is a switch from 1 to 2 in that part of the map, there can be a new redundant R-unit.

This leads to our algorithm as follows. At the end of each step, we iterate each entry in $\mathbf{M}(\mathbf{R}^* \cap \bigcup \mathbf{B})$, and increase them by one. If $\mathbf{M}(\mathrm{t}) = 2$ (after the increment), we retrieve the R-unit $\mathbf{R}'$ in $\mathbf{B}$ that covers this tuple. If $\{\mathrm{t}|\mathbf{M}(\mathrm{t}) = 1, \mathrm{t} \in \mathbf{R}'\} = \emptyset$, we remove $\mathbf{R}'$ from $\mathbf{B}$ and decrease each entry in $\mathbf{M}(\mathbf{R}')$ by one.

4.4.3 Smart R-Cover Algorithm

Here we describe our algorithm for the representation birthing problem, which is shown in Algorithm 7.

Our algorithm takes as input the user-input relation $\mathbf{R}$, and the set of functional dependencies FD. We first populate the full lattice, by adopting a top-down approach using partition product introduced in [51] (line 2). We then prune all the lattice nodes whose R-units must be FD-redundant (line 3). We continue with the R-unit covering procedure.

We first initialize the cover set $\mathbf{B}$ and the cover map $\mathbf{M}$ (line 4-5), and directly use $\mathbf{R}$ to record the set of uncovered tuples. While $\mathbf{R}$ is non-empty, we repeat the following greedy search step. We first find the set of best choices in the first-level partition that cover m uncovered tuples (line 8), and record their corresponding set of attributes (set of single column in this case) in $\mathbf{S}_1$ (line 9). We randomly pick one of the best choices as the current best choice (line 10). We then check if there are any additional best choices down in the lattice (line 11- 24). At each deeper level, we only consider partitions all of whose parents have best choices (line 15). We collect all the best choices in these partitions (line 16) and pick the one with the least overlapping (line 23). At the end of each step, we prune redundant R-units in $\mathbb{B}$ (line 25- 36), as we describe before. Finally, we update the cover set $\mathbb{B}$ (line 37) and the set of uncovered tuples $\mathbb{R}$ (line 38).

4.4.4 Optimized R-unit Covering

While smart R-cover saves computation by scanning much fewer lattice partitions, it still suffers from the cost of populating the full lattice. According to Theorem 4.4.1, we need to check the deeper lattice only in the presence of ties, which has a small probability. Intuitively, we can just scan the first level of the lattice for the best choices. Only when we have ties, we dynamically create the partition product of these tie R-units and check if the result qualifies for an extra best choice. By doing this, we only need to populate the first level of the lattice in the lattice population step. We apply this optimization to Smart R-cover and obtain the **Optimized R-cover** algorithm. We will gain a performance insight into these algorithms in Section 4.6.2.

4.5 R-unit Algebra

So far we have discussed how to give birth to an initial R-unit representation to the user. However, there is no guarantee that the selection condition implied by this initial representation is exactly the *user-desired* one. In this section, we describe how do we allow the user to easily explore the space of R-unit

Algorithm 7 Representation Birthing

```
procedure Birth($\mathbf{R}$, $\mathbf{FD}$)
    $\mathbf{L} \leftarrow$ PopulateLattice($\mathbf{R}$)
    $\mathbf{L} \leftarrow$ PruneByFD($\mathbf{L}$, $\mathbf{FD}$)
    $\mathbf{M} \leftarrow$ InitCoverMap($\mathbf{R}$)
    $\mathbf{B} \leftarrow \emptyset$                                          ▷ initialize cover set
    while $\mathbf{R} \neq \emptyset$ do
        $m \leftarrow \max_{\mathbf{R}^* \in \bigcup \mathbf{L}_1} |\mathbf{R}^* \cap \mathbf{R}|$                  ▷ find local best choices
        $\mathbf{E}_1 \leftarrow \{\mathbf{R}^* | \mathbf{R}^* \in \bigcup \mathbf{L}_1 \wedge |\mathbf{R}^* \cap \mathbf{R}| = m\}$
        $\mathbf{S}_1 \leftarrow \{\overline{X} | \overline{X} = \Pr(\mathbf{R}^*), \mathbf{R}^* \in \mathbf{E}_1\}$
        $\mathbf{R}' \leftarrow$ RandomPick($\mathbf{E}_1$)
        for $i = 2..|\mathbf{S}_1|$ do                                ▷ search for more best choices
            $\mathbf{E}_i \leftarrow \emptyset$
            $\mathbf{S}_i \leftarrow \emptyset$
            for $\mathbf{P}_{\overline{X}} \in \mathbf{L}_i$ do
                if $\forall A \in \overline{X}, \overline{X} \backslash A \in \mathbf{S}_{i-1}$ then
                    $\mathbf{T} \leftarrow \{\mathbf{R}^* | \mathbf{R}^* \in \mathbf{P}_{\overline{X}} \wedge |\mathbf{R}^* \cap \mathbf{R}| = m\}$
                    if $\mathbf{T} \neq \emptyset$ then
                        $\mathbf{E}_i \leftarrow \mathbf{E}_i \cup \mathbf{T}$
                        $\mathbf{S}_i \leftarrow \mathbf{S}_i \cup \overline{X}$
                    end if
                end if
            end for
            $\mathbf{R}' \leftarrow \arg\min_{\mathbf{R}^* \in \mathbf{E}_i} |\mathbf{R}^* \backslash \mathbf{R}|$
        end for
        for $\mathrm{t} \in \mathbf{R}'$ do                                    ▷ eliminate redundancy
            $\mathbf{M}(\mathrm{t})++$
            if $\mathrm{t} \in \mathbf{R}^* \backslash \mathbf{R} \wedge \mathbf{M}(\mathrm{t}) = 2$ then
                $\mathbf{R}'' \leftarrow \mathbf{R}^* | \mathbf{R}^* \in \mathbf{B} \wedge \mathrm{t} \in \mathbf{R}''$
                if $\{\mathrm{t} | \mathrm{t} \in \mathbf{R}'' \wedge \mathbf{M}(\mathrm{t}) = 1\} = \emptyset$ then
                    $\mathbf{B} \leftarrow \mathbf{B} \backslash \{\mathbf{R}''\}$
                    for $\mathrm{t} \in \mathbf{R}''$ do
                        $\mathbf{M}(\mathrm{t})--$
                    end for
                end if
            end if
        end for
        $\mathbf{B} \leftarrow \mathbf{B} \cup \{\mathbf{R}'\}$                                   ▷ update cover set
        $\mathbf{R} \leftarrow \mathbf{R} \backslash \mathbf{R}'$                                  ▷ update uncovered tuples
    end while
    return $\mathbf{B}$
end procedure
```

representation, in order to arrive at the the representation with the desired selection condition specification.

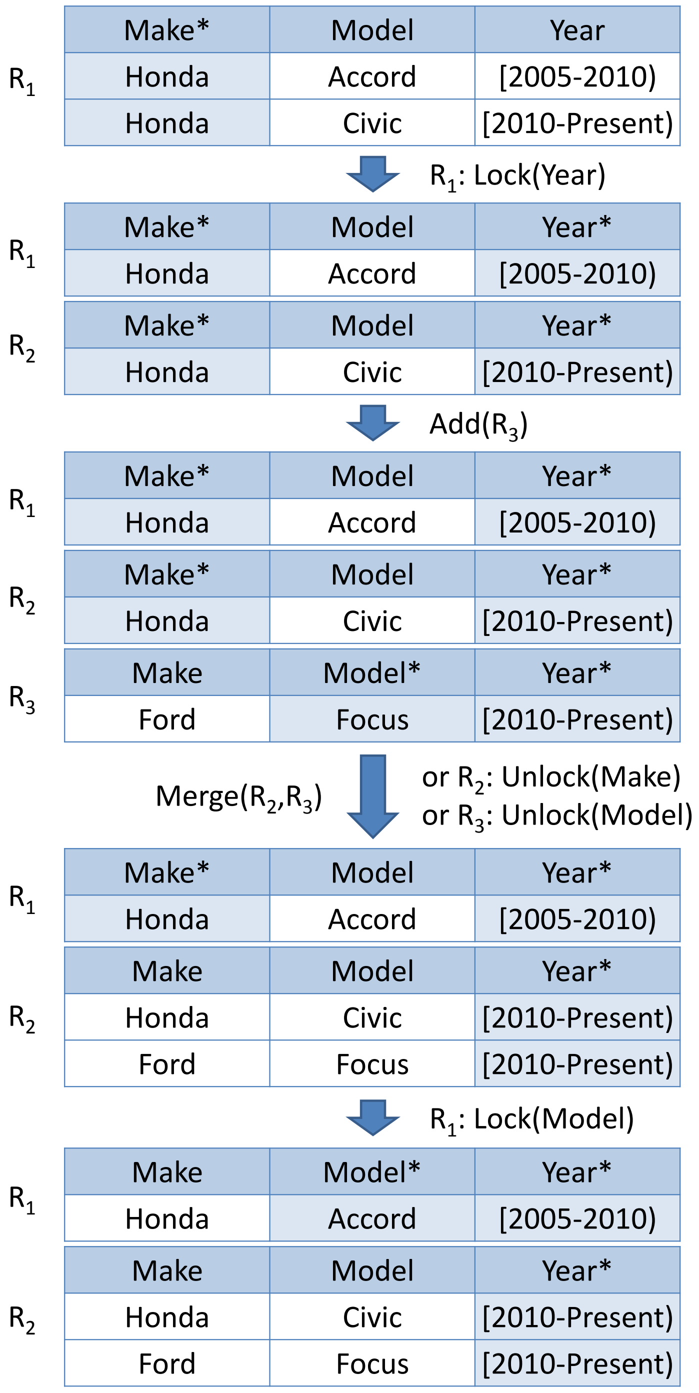
In this chapter, we propose a set of direct manipulation operators implemented with point-and-clicks or drag-and-drops for the user to easily navigate through the representation space. Collectively, we call it **R-unit Algebra**.

4.5.1 R-unit Algebra Operators

Our R-unit algebra consists of following operators: Add, Remove, Lock, Unlock and Merge, where Lock, Unlock and Merge restructure the R-unit representation of a given $\mathbf{R}$, while Add and Remove also augment or diminish the scope of $\mathbf{R}$. We describe each of them as follows.

Lock In Example 2, we see that how a disjunct relation can imply multiple conditions in a relational conjunction (RC) form. If the two tuples in Example 2 are all the user input, this disjunct relation can mean four selection conditions: (1) none, (2) $Make = \mathrm{Honda}$, (3) $Year = [2005, 2010)$ or (4) $Make = \mathrm{Honda} \wedge Year = [2005, 2010)$. Say, at some stage, we represent the second condition by a corresponding R-unit to the user, as shown in Figure 4.1. Now, what if the actual user-desired condition is the fourth one?

In the representation of an R-unit $\mathbf{R}^*$ with a predicate tuple t, we say a column A is locked, if and only if $\mathrm{t}[A] = \mathbf{True}$. The intuition is, whenever A is locked, it means we have a predicate on that attribute. Consequently, all the tuples in $\mathbf{R}^*$ must have the same value on A. For instance, in Figure 4.1, $Make$ is locked, meaning that we have a predicate $Make = \mathrm{Honda}$ in the condition corresponding to this R-unit. If the user actually requires an additional pred-



R_1

Make*	Model	Year
Honda	Accord	[2005-2010)
Honda	Civic	[2010-Present)

R_1: Lock(Year)

R_1

Make*	Model	Year*
Honda	Accord	[2005-2010)

R_2

Make*	Model	Year*
Honda	Civic	[2010-Present)

Add(R_3)

R_1

Make*	Model	Year*
Honda	Accord	[2005-2010)

R_2

Make*	Model	Year*
Honda	Civic	[2010-Present)

R_3

Make	Model*	Year*
Ford	Focus	[2010-Present)

Merge(R_2,R_3)
or R_2: Unlock(Make)
or R_3: Unlock(Model)

R_1

Make*	Model	Year*
Honda	Accord	[2005-2010)

R_2

Make	Model	Year*
Honda	Civic	[2010-Present)
Ford	Focus	[2010-Present)

R_1: Lock(Model)

R_1

Make	Model*	Year*
Honda	Accord	[2005-2010)

R_2

Make	Model	Year*
Honda	Civic	[2010-Present)
Ford	Focus	[2010-Present)

Figure 4.5: Examples of R-unit Algebra

icate on $Year$, she can just *lock* the $Year$ attribute. In practice, she can just click the corresponding cell in the table header to toggle the status of $Year$ from unlocked to locked. The new predicate $Year = [2005, 2010)$ will be automatically derived from the existing values in $\mathbf{R}^*$ and appended to the current condition.

Note that, the values on the attribute to be locked may not be the same. For instance, suppose we have different values under $Year$ (as shown in the first R-unit in Figure 4.5). In this case, since the R-unit can no longer satisfy the new predicate, it will be *split* into two smaller disjoint R-units, as shown in the second representation in Figure 4.5. We notice, the result R-units of the split may be redundant. In order to maintain the representation to be well-formed, we have to check each such R-unit using the same technique we have discussed in Section 4.4.2, and remove it from the representation if it is redundant.

Add/Remove We allow the user to directly add example tuples in our R-unit representation. Specifically, the user can either add a new tuple in an existing R-unit, or create a new R-unit containing only the new tuple. If the new tuple is added to an existing R-unit, values under the locked attributes will be copied from other tuples to the new tuple, which serves as both a validness guarantee and a way to save user efforts. If the new tuple is added as a new R-unit, we try to make this new R-unit as specific as possible by locking as many attributes as we can. For instance, in Figure 4.5, we add a R-unit containing a single new tuple for Ford Focus as $\mathbf{R}_3$. Due to the FD from $Model$ to $Make$, we automatically lock $Model$ and $Year$ for the user. Add will not cause any redundancy.

We also allow the user to directly remove tuples from any R-units in the representation. If the tuple to be removed is the only one in the R-unit, we delete the R-unit accordingly. We do not lock any additional attribute even if all the tuples in the R-unit have the same value on that attribute after the tuple removal, for the same reason we have described in Example 2. However, we do take care of redundancies here, because the R-unit may be redundant after the tuple removal.

Unlock Unlock is the reverse of Lock. By unlocking an attribute in an R-unit, we mean to remove the corresponding predicate from the selection condition represented by the R-unit. Unlock can be issued using point-and-clicks similar to Lock. In fact, all the user has to do is to click the corresponding cell in the table header, and the semantics will be clear according to the current lock status of that attribute. If it is previously locked, the click means to unlock it, and vice versa.

According to Definition 3, Unlock may *expand* the R-unit to include tuples from other R-units. For instance, in Figure 4.5, if we unlock $Make$ in $\mathbf{R}_2$ in the third representation, $\mathbf{R}_2$ will expand to the $\mathbf{R}_2$ in the fourth representation, where it absorbs old $\mathbf{R}_3$. As a result, the old $\mathbf{R}_3$ is redundant and need to be removed from the representation.

Similarly, we can achieve the same result by unlocking $Model$ in $\mathbf{R}_3$. Indeed, both of these two operations results in a *merge* of $\mathbf{R}_2$ and $\mathbf{R}_3$ in the third representation, which we describe as follows.

Merge Merge helps the user to directly combine two R-units, which implies the generation of a selection condition (in RC) that is more general than the one represented by both R-units. In practice, this operation can be easily specified by a drag-and-drop gesture: the user may drag one R-unit and drop it on another R-unit to specify such a merge.

If we merge $\mathbf{R}_1$ and $\mathbf{R}_2$, we discard all locks except the common locks on $\mathrm{Pr}(\mathbf{R}_1) \cap \mathrm{Pr}(\mathbf{R}_2)$. Furthermore, for each attribute A on which we have a common lock, we retain it only when tuples from $\mathbf{R}_1$ and $\mathbf{R}_2$ have the same value on A. For instance, by merging $\mathbf{R}_2$ and $\mathbf{R}_3$ in the third representation, only the lock on $Year$ is preserved.

Again, Merge may cause other R-units to be redundant since it may largely generalize the corresponding selection condition. We have to check the redundancy of other R-units and prune the redundant ones accordingly.

4.5.2 Expressive Power Analysis

Although our R-unit algebra is very simple, here we show that the algebra is very expressive, in a sense that it is sound and complete on the space of all well-formed R-unit representation given $\mathbf{R}$ (i.e, we do not consider Add or Remove, since they will alter the scope of $\mathbf{R}$).

Soundness The soundness says, given a well-formed R-unit representation, after applying any restructuring operator in the R-unit algebra, the result representation is still well-formed. The proof is straightforward. Recall the definition in Section 4.3, a well-formed R-unit representation is complete and non-redundant. Thus we only need to prove, the result of applying any restructuring operator in the R-unit algebra on a well-formed R-unit representation is still complete and non-redundant.

For Lock, the split of the old R-unit does not change the coverage, so completeness is guaranteed. For Unlock and Merge, the result R-unit can only contain more tuples than the operand R-unit(s), so the result representation must also be complete. Since we explicitly prune possible redundancy during the execution of each operator, the non-redundancy is also guaranteed.

Completeness The completeness says, given any pair of well-formed R-unit representation, one as the start point and the other as the end point, there is at least one sequence of the restructuring operators from the algebra, which can transform the start point representation to the end point. Here we show that, this can be done with only Lock and Unlock.

To construct the sequence of operators, we separate the transformation into two phases. In the first phase, we transform the start point to a *fully-partitioned* representation, where each R-unit only contains a single tuple and has locks on all the attributes (we omit FDs here for simplicity). In the second phase, we transform the fully-partitioned representation to the end point. To construct the operator sequence for the first phase is straightforward. We only need to lock every unlocked attribute in every R-unit in the start point representation. We thus focus on the second phase in the following.

Given the end point representation, we first trim each R-unit in it. Specifically, given any R-unit, we preserve the tuples in it that is *only* covered by this R-unit (i.e., where $\mathrm{M}(\mathrm{t}) = 1$), and trim the other tuples. Since the end point representation is non-redundant, the trimmed R-units must be non-empty. For each trimmed R-unit, we randomly pick one tuple we call *the seed*. Obviously, such seeds must appear in the fully-partitioned representation.

Here is how we construct the sequence of operators for the second phase. Given the fully-partitioned representation, we first allocate those R-units which contain the set of seeds from the end point representation. We call these R-units *seed R-units*. We can then *grow* each of these seed R-unit to the corresponding R-unit that contains the seed in the end point representation. In fact, if the corresponding R-unit in the end point has locks on the attribute set $\overline{X}$, we just need to unlock each attribute in $\mathrm{S}(\mathbf{R})\backslash\overline{X}$ in the seed R-unit. The composition of the operations from both phases completes the proof.

As a final remark, the operator sequence constructed in the above proof is tediously long. In practice, the actual sequence a user specifies could be much shorter. Although it would be interesting to examine an optimal sequence here, the problem is beyond the scope of this chapter.

4.6 Evaluation

The critical question to our two-phase solution of example-driven selection condition specification is that, whether some initial examples would be able to generate an initial selection condition that is close enough to the ground truth. In this section, we demonstrate a positive result from a simulated user study.

Moreover, in order to gain a performance insight into our R-cover algorithm, we conduct a synthetic experiment and show that, our algorithm is much more efficient compared with the naive algorithm described in Section 4.4.2, and effectively eliminates redundancies.

We use two datasets in our experiments: Yahoo Movies and IMDb. The Yahoo Movies dataset is 500MB in size and contains 43 relations and 131 attributes. The IMDb dataset is 2GB in size and contains 19 relations and 57 attributes. Both datasets are stored in MYSQL 5. All the experiments were run on a desktop machine with an Intel Core i7 860 @ 2.80GHz and 8GB RAM.

4.6.1 Birthing with Simulate Tasks

In this section, we show that, given a ground truth selection condition, and just a few examples from the result of the ground truth selection, we are able to derive an initial selection condition that is close enough to the ground truth. Specifically, we design the ground truth to be three simple selection conditions shown as follows:

T1 $Title = \text{“Harry Potter”} \vee Director = \text{“Mike Newell”}$

T2 $Title = \text{“Harry Potter”}$

$\vee(Director = \text{“Mike Newell”} \wedge Genre = \text{“Action”})$

T3 $(Title = \text{“Harry Potter”} \wedge Director = \text{“David Yates”})$

$\vee(Director = \text{“Mike Newell”} \wedge Genre = \text{“Action”})$

Note that, we deliberately design some overlap between the disjunct relations in each task to make them non-trivial. The complexity of these tasks are monotonically increased.

For each of the three ground truth, we simulate the user input as follows. First, we create a database view of all the attributes involved in these tasks (a view with attributes $Title$, $Director$ and $Genre$) by joining and projecting corresponding relations and attributes in the underlying movie database. After this, we apply the ground truth selection condition on the view to generate a *user-input candidate set* **U**. We then simulate the user input by randomly sampling a number of tuples from **U**, where we vary the number from one to the full size of **U**.

For each simulated user input, we feed it to our birthing algorithm, retrieve the output selection condition and compare it with the ground truth. In order to measure the difference, we introduce a metric we name *operation distance*. Intuitively, it stands for the minimal number of R-unit algebra operations (Lock-/Unlock) needed to refine the initial output condition to the ground truth. For instance, if the output is identical to the ground truth, the distance is zero. If the output is **T1** but the ground truth is **T2**, the distance is one since the user needs to further lock an attribute $Genre$ in the second disjunct relation in **T1**.

For each task and each sample size from one to the size of **U**, we randomly sample **U** for one hundred times and use the sample to generate the initial condition. The average operation distance between these conditions and the ground truth condition is shown in the top of Figure 4.6. As can be seen from the figure, the distance drops dramatically as more simulated input are provided. On average, the user only needs about one extra operation to derive the desired condition after providing five examples. For task one and two, our birthing algorithm almost directly infer the desired condition with more than ten examples. Notice that, however, task three does not converge to the ground truth even with complete input. This is because, “Mike Newell” also directs “Harry Potter” which has a $Genre$ of “Action”. Thus the R-unit with $Title = \text{“Harry Potter”}$ is always a better choice than the first R-unit in **T3** according to our greedy heuristics. We are exploring this problem and planning to refine our algorithm using selectivity information in the future work.

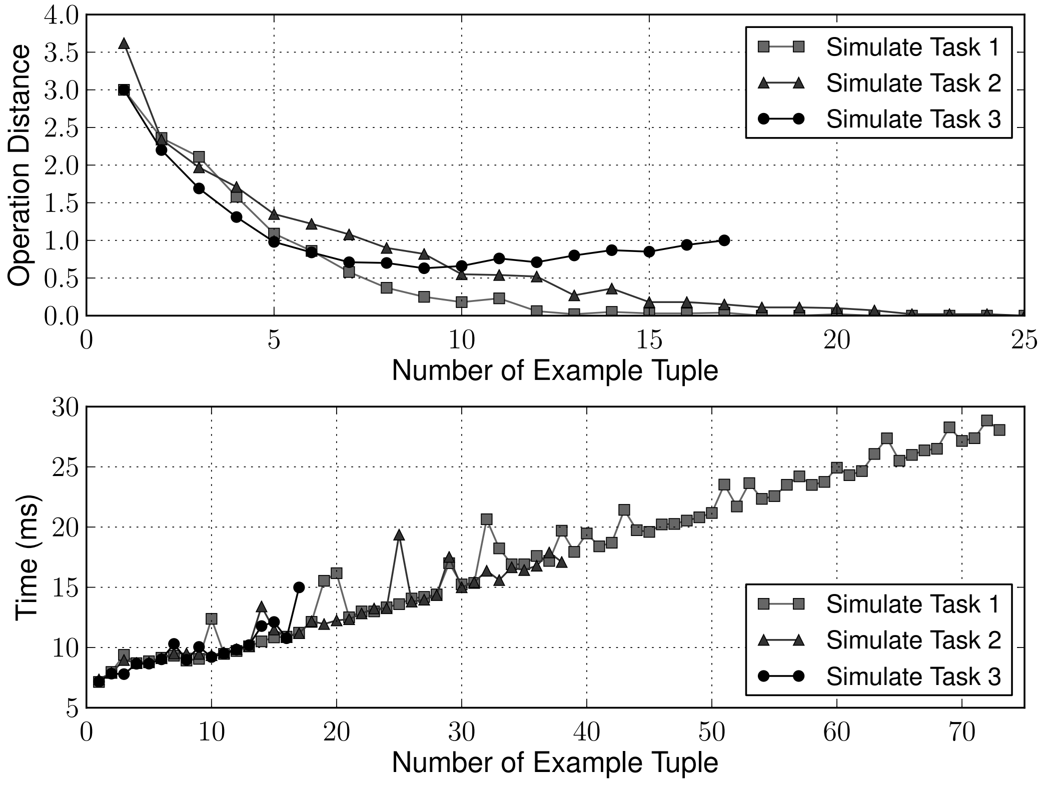


Figure 4.6: The average operation distance and execution time for birthing the initial conditions with various sample size.

The execution time for birthing with simulated input is shown in the bottom of Figure 4.6. Since the set of simulate task set is relatively selective, the sample size is small and our algorithm is able to achieve near-linear performance. All experiments are completed within 100ms, which is sufficient for practical human computer interaction.

4.6.2 Birthing with Synthetic Data

The previous experiments are all based on domain-specific data and tasks with limited input size. In order to obtain a general performance insight into our R-cover algorithm, we further experiment it with a synthetic dataset. Specifically, we generate the input relation with a set of numeric tuples. For each attribute in each tuple, we randomly pick a number from an integer pool. As a starting point, we set the pool size to be the rounded square root of the total number of tuples. We run Optimized R-cover, Smart R-cover and the naive covering algorithms on this synthetic dataset, and compare the total execution time.

We start by fixing the number of column to be five and varying the row size from 10^i, 2×10^i to 5×10^i, $i = 2..4$. As can be seen from Figure 4.7, both our algorithms are almost one order of magnitude faster than the naive algorithm. This is because instead of traversing the whole lattice, most of the time (if no ties are present) our algorithms only need to traverse the first level of the lattice. In addition, on average, Optimized R-cover is around 30% faster than Smart R-cover, due to the fact that the former only needs to populate the first level of the lattice, while the latter has to compute all partitions throughout the lattice.

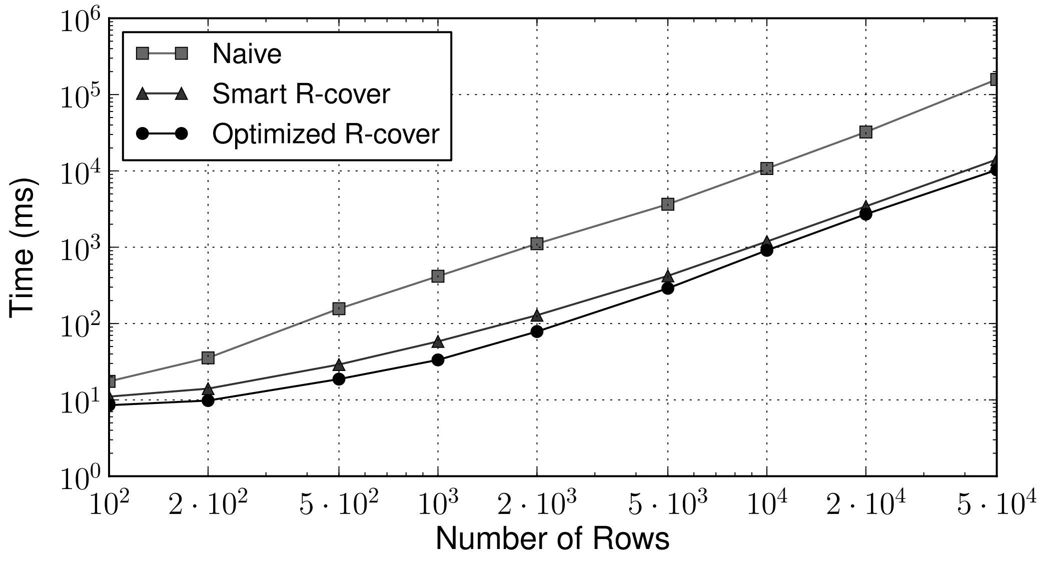


Figure 4.7: The overall execution time for birthing on a relation with five columns and varying row number.

Since the column number is fixed at five in the previous experiment, the save on lattice population is not clearly seen. In order to magnify the performance difference between the optimized R-cover and the smart R-cover, we further fix the row number at one thousand (with a corresponding pool size of 32) and vary the column size from one to ten. The result is shown in Figure 4.8. As expected, while the one-order-of-magnitude difference remains between the naive algorithm and Smart R-cover, the difference between Optimized R-cover and Smart R-cover increases exponentially with respect to the column number. This is because given a column number c, Smart R-cover needs to populate 2^c lattice nodes, while Optimized R-cover only needs to populate c nodes on the first level.

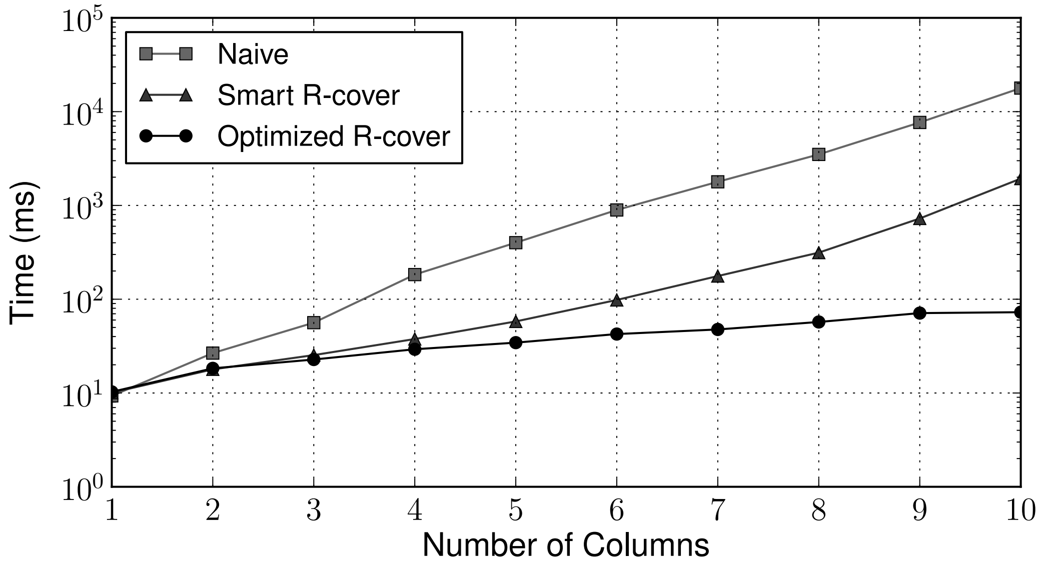


Figure 4.8: The overall execution time for birthing on a relation with one thousand columns and varying column number.

4.7 Conclusions

In this chapter we studied how to make it easy for end-users to specify s-election conditions. We started by formalizing the example-driven selection condition specification problem, with a general assumption of the family of conditions to be specified. We then proposed a two-phase solution based on a representation of such selection conditions. In the birthing phase, we automatically derive an initial representation of the estimated user-desired condition. During the refining phase, we designed an algebra for the user to easily modify the initial representation until she reaches the desired selection condition.

We folded these ideas into a prototype system, and evaluated via a simulated user study and synthetic experiments. Our results show that, our system is able to generate high-quality initial estimates of the desired condition, and our algorithms for deriving the initial estimates are efficient and scalable.

CHAPTER V

Incremental Information Integration

While information integration could itself be hard to end-users, as we have seen in chapter III, it becomes even more challenging when the end-user has to maintain it. Specifically, when either new increments of source data are available for processing, or when new data sources become available with their own integration rules, there is a need to regenerate the integrated data, in an incremental way, by exploiting only the data that is new, and only the new integration rules (in the case of new data sources). This is important because the end-users would otherwise need to reintegrate everything again from scratch, which places a heavy operational burden on them.

In this chapter we present an incremental integration framework for complex nested data that applies to both new source data and new integration rules. Our method utilizes HIL [48], a recent declarative integration language. A salient feature of HIL, which makes it appealing for incremental integration, is that it logically decomposes a complex integration task into simple rules that are based on indexes. We give a method for the incremental evaluation of such rules that consists of two phases: (1) an incremental chase with the new rules and data, to create a flat, index-based representation of the target data, followed by (2) a de-reference procedure that traverses all the indexes to construct the desired nested representation of the target data. We study which data structures should be maintained to facilitate incremental integration, and give a novel de-reference algorithm that is based on generating queries with joins. We evaluate the resulting method on a real-world financial integration scenario based on SEC filings, and show significant performance benefits.

5.1 Introduction

Information integration has long been an important problem in the industry and, also, has received substantial attention from the research community. Integration usually involves heterogeneous sets of structured and unstructured data items that must be cleansed, normalized, linked and mapped into a set of target entities. There are now large public data sources that are constantly updated with new data and that can be combined with other external or internal (enterprise) data sources. For instance, it is not uncommon these days to see projects or start-ups integrating data extracted from blogs and microblogs (e.g., Twitter), from public data sources such as the U.S. Census or the U.S. Securities and Exchange Commission (SEC), or from knowledge bases like Wikipedia, DBLP, or IMDb. The end-goal of most of these efforts is to provide integrated data that can be used for subsequent data analysis.

A well-known challenge occurs when either new increments of source data (over the same source schema) become available, or when entirely new data sources (with new schemas) must be added into the incremental process. In the latter case, new integration rules must be written to account for the new type of data. Furthermore, the target schema itself may need to be adjusted to accomodate possibly new attributes. In both cases, there is a need to regenerate the integrated data, in an incremental way, without re-evaluating everything from scratch. This is important for efficiency reasons but also in cases where the previous source data or the previous integration rules are no longer available.

In this chapter we present a declarative incremental integration framework for complex nested data that makes use of HIL [48], a recent declarative language for entity integration. HIL expresses the integration logic as rules that transform source “raw” records into target “entities”. An important feature of HIL, which makes it particularly appealing for incremental integration, is the ability to logically decompose a complex integration task into simple rules that are based on indexes. Consequently, subsequent information integration steps will often be directed to only a few indexes (the ones that are relevant to the new increments of data). Furthermore, HIL has other appealing properties, such as type inference and record polymorphism [83], which in turn enables to automatically refine the target schema in response to the new integration rules.

Our incremental integration framework applies seamlessly to new increments of data and to additions to the integration logic. At one extreme of the spectrum, we cover what is called *incremental view maintenance* [25], which creates a new version of the target in response to changes in the input data (without any changes in the logic itself). At the other end of spectrum, we cover what is known as *view adaptation* [44], which creates a new version of the target in response to changes in the integration logic itself (without any changes in the source data).

To illustrate how our framework works, we present a simplified version of a real integration flow in the financial domain based on SEC filings. The complete version of this scenario will be used later for experimental evaluation.

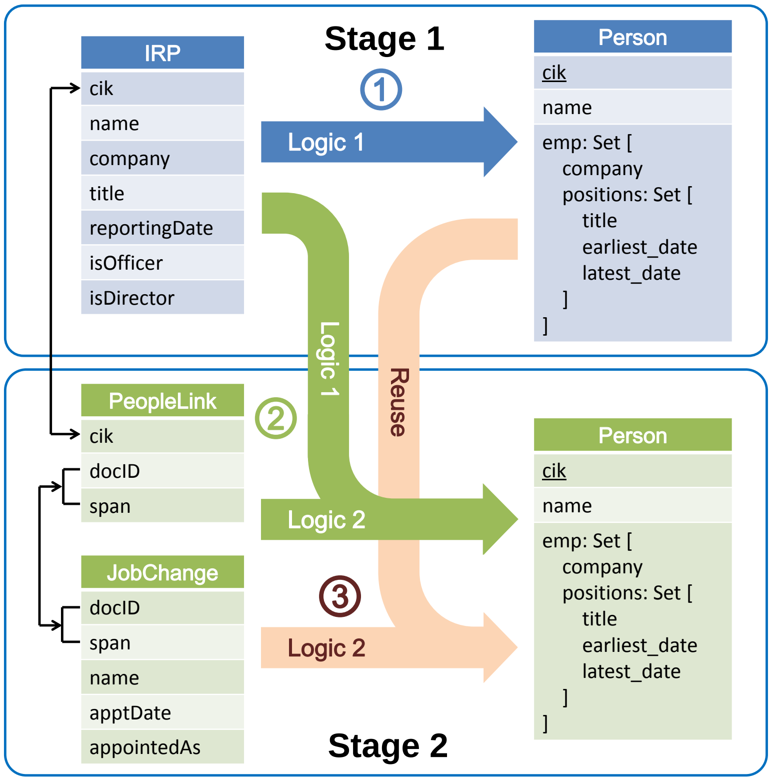


Figure 5.1: SEC Scenario: Alternative Integration Strategies

Example 6. Public financial institutions and individuals connected to these institutions are required to file periodic reports with the US SEC agency, disclosing financial results of companies, information about directors and board members of those companies, as well as their stock transactions and holdings. These fiilings represent a rich set of sources of information that form the basis for subsequent integration and data analysis [23].

We divide our example into two stages, as shown in Figure 5.1. The first involves a (non-incremental) mapping from one of the SEC sources (IRP, which stands for insider reports for people) into a target Person entity type. We represent the set of integration rules that specifies this transformation as *Logic 1*. We will give details later about how this logic is expressed in HIL. The role of *Logic 1* is to create, for each person, a nested representation of all the information known about that person from all the insider reports. This information includes the SEC id (called cik), the name, as well as more complex structures such as the list of employment records with all the known companies for which the person worked, with all the positions held within each company, and with an understanding of the time-span for each position (based on all the records in IRP).

The second stage corresponds to adding a new data source (JobChange, containing historical records mentioning when a person was appointed into a position or left a company) into the integration flow. The goal here will be to integrate the new information into the existing Person entity data in an incremental fashion. The new information may result in additional companies in the employment history, or additional positions, or perhaps changes in the earliest or latest date for a position, as more is learned from the new data source. One additional complication in this scenario is that the JobChange records are extracted from text documents and do not include the actual cik of a person. Thus, to link a JobChange record to a known Person entity, we assume a separate entity resolution step that produces a set PeopleLink of links between JobChange and Person.

The fields docID and span, in this example, form an identifier for each JobChange record. We represent the set of rules to map the new information from JobChange (and PeopleLink) into Person as *Logic 2* (to be shown later as well).

The default strategy to combine the data from the two sources is to take the union of *Logic 1* and *Logic 2* and evaluate it as if it were a single set of rules on top of both IRP and JobChange. This strategy (shown with number 2 in the figure) is also the standard strategy that is supported by the HIL compiler [48]. Without going into details of HIL compilation, the advantage is that the compiler has access to all the rules and all the data, and hence it can stage appropriately all the transformation steps as well as the fusion that needs to be done across the two types of data. The disadvantage of this strategy is that it pretty much re-executes all the integration steps for IRP and *Logic 1*.

The more challenging strategy (shown with number 3 in the figure) is to avoid touching both the previous source data IRP and the previous rules *Logic 1*, which may not even be available anymore, and to create a new Person entity data based only on *Logic 2* and the new data source, while *reusing* the result of previous integration for Person. An important condition that one must require is that the incremental strategy must be equivalent (in terms of the final result) to the non-incremental strategy.

The goal of this chapter, as suggested by the above example, is to develop the algorithms that allow for the incremental integration strategy, along with any auxilliary data structures that may need to be maintained in order to support this incremental strategy. Towards this goal, we develop an incremental evaluation strategy that is based on two phases: *chase* and *de-reference*.

The chase phase takes as input a set of integration rules and a data set, and generates a flat representation of the target data that uses *index references* to encode the hierarchical structure of the data. The chase makes essential use of HIL indexes, and is similar in spirit to the chase with second-order tgds in [39], which uses symbolic Skolem terms rather than index references. For our example, chasing with the integration rules of *Logic 1* will generate flat *Person* records where the emp component is represented, lazily, by a reference to an index (Employment) that is populated separately by other rules. The advantage is that the population of *Person* is decorrelated from the population of the Employment index. Furthermore, indexes are amenable for incremental evaluation, since a new chase stage with new integration rules or new source data will incrementally add into the relevant indexes. One of the key ideas of our approach is to materialize the flat result of the chase (containing “frozen" index references) as a data structure that can be reused and incrementally maintained.

The de-reference phase can then be used to transform the flat result of the chase into the desired nested target data. While a naive implementation of de-reference would walk the flat data structure and recursively “inline" all index references with the actual values stored at the indexes, we show that a more efficient alternative can be obtained by using an additional data structure that can also be incrementally maintained. This data structure, which we call *reference summary graph* (or *reference summary*), encodes all the possible types of references that may appear in a record. It enables us to develop an optimized de-reference algorithm that is based on generating a non-recursive set of queries with join operations. Even though the de-reference procedure itself is non-incremental, the efficiency of the generated queries allows us to achieve good overall performance.

Contributions and Chapter Outline. We summarize the main contributions of this chapter as follows:

- We describe a declarative incremental integration framework for nested data, which is based on simple integration rules that make use of indexes.
- Our incremental integration framework applies seamlessly to both new increments in the data and additions to the integration logic.
- We give an integration strategy that is based on two phases: chase and de-reference. The chase is incremental and produces flat, index-based representations of the target data that can be incrementally maintained.
- We give an optimized de-reference method by generating efficient, non-recursive, queries with join. The method is based on an auxilliary reference summary graph that can also be incrementally maintained.
- We evaluate our method, experimentally, on a financial integration scenario based on SEC filings. We show that as new types of data, and new types of rules are added to the integration process, the incremental recomputation of the target entities achieves significant performance benefit.

The rest of the chapter is organized as follows. Section 5.2 briefly introduces HIL and describes the integration logic for Example 6. We formalize the incremental integration problem in Section 5.3 and describe a naïve solution based on deep-union of nested data in Section 5.4. We develop the chase and de-reference solution in Sections 5.5 and 5.6, and give our optimized de-reference method in Section 5.7. We present our experimental results in Section 5.8 and conclude in Section 5.9.

5.2 Preliminaries

We now describe the declarative framework provided by HIL, the language we use in this chapter. The main ingredients of HIL are: (1) *entity types*, which define the logical objects that users create and manipulate, and (2) declarative *rules* that transform entities. Entity types are represented in HIL's data model as nested-sets of values and records. A special form of entity types are *indexes*, which allow grouping of sets of values under a key value and facilitate the generation and maintenance of nested entities. We will now use parts of the example in Figure 5.1 to illustrate how each of these HIL ingredients are used to express an integration flow.

5.2.1 Constructing Entities

We start by illustrating how we transform IRP to Person ($Logic1$ in Figure 5.1). First, we declare some of the relevant entity types:

```
IRP: set [cik: int, name: string, ?];
Person: set [cik: ?, name: ?, emp: set ?, ?];
Employment: fmap [cik: int]
               to set [company: string, positions: set ?];
Positions: fmap[cik:int, company:string]
           to set [title:string, earliest_date: ?];
PositionInfo: fmap[cik:int, company:string, title:string]
              to set [date: ?];
```

The data model of HIL allows for sets and records that can be arbitrarily nested. In the above, IRP and Person are both sets of records, whose types are only partially specified and will be inferred with more rules. Employment is declared to be an index (or finite map) that associates each key, in this case a record with a single cik value, to a set of record values containing the company name of the employer and a set of positions with that employer. Intuitively, this index will be used to associate the employment information of each Person identified by a cik. Positions and PositionInfo provide finer-grained information regarding the person's employment history, such as the titles the person has had in a specific company, or the dates that the person is known to have a given title in a given company.

To create Person, we use the following HIL rule:

```
rule m1:   insert into Person
           select [cik: i.cik, name: i.name
                   emp: Employment![cik: i.cik] ]
           from IRP i;
```

The semantics of the rule is one of containment. For every record in IRP, we require the existence of a Person entity with corresponding cik and name values. The emp field, however, is specified via an *index look-up* operation Employment![cik: i.cik], that is, emp will be assigned the value returned by probing the index Employment on the cik value. The population of the Employment index is specified using separate HIL rules such as:

```
rule m2:   insert into Employment![cik: i.cik]
           select [company: i.company,
                   positions: Positions![cik: i.cik, company: i.company] ]
           from IRP i   where i.isOfficer = true;
```

Here, for each IRP record, we construct an index entry on each *unique* cik value. Since there might be more than one record in IRP with the same cik, each index entry will be a set of records computed from each group of IRP records with the same cik. In effect, this rule incorporates a group by operation on IRP. Note that the rule populates the nested positions set using a look-up into a separate

Positions index. Following the same pattern as with Employment, the Positions index is defined in a separate rule m3 that groups IRP records by cik *and* company. Similarly, another rule m4 groups IRP records by an additional title field to populate PositionInfo. We note that HIL also allows UDFs (user-defined functions) in the rules. For instance, in order to populate the earliest_date in Positions, m3 applies an aggregation function minDate on the index look-up value from PositionInfo.

```
rule m3 :insert into Positions![cik: i.cik, company: i.company]
         select [title: normTitle(i.title),
                 earliest_date:
                    minDate(PosInfo![cik: i.cik,
                                     company: i.company,
                                     title: normTitle(i.title)])]
         from IRP i   where i.isOfficer = true;

rule m4: insert into PosInfo![cik: i.cik, company: i.company,
                              title: normTitle(i.title)]
         select [date: i.reportingDate]
         from IRP i   where i.isOfficer = true;
```

Indexes are a key construct in HIL entity integration flows. They allow to decorrelate a complex nested mapping into several smaller rules, where each rule is focused on a particular aspect of a complex entity (e.g., employment but not positions). The HIL compiler figures out the necessary lowere-level operations such as group-by, union, duplicate removal, and nesting of data, which ultimately assemble the complex entities. The type of logical *decorrelation* that HIL allows is key to declarative incremental integration, because it will offer an opportunity to incrementally enrich information at various levels in the entity hierarchy without affecting the rest of the target data.

5.2.2 Fusing Additional Entities

The next step in the entity integration flow in Figure 5.1 illustrates how to add JobChange and PeopleLink into the existing flow. JobChange records will not add new Person entities. Instead, each matched JobChange record (via PeopleLink) may result in the addition of new records to the employment list of a Person entity, or in the positions list. Two HIL rules specify how to add such records into the Employment and Positions indexes:

```
rule m5:  insert into Employment![cik: l.cik]
          select [ company: j.company,
                   positions: Positions![cik: l.cik, company: j.company] ]
          from JobChange j, PeopleLink l
          where j.docid = l.docid and j.span = l.span
            and  isOfficer (j.appointedAs) = true;

rule m6:  insert into Positions![cik:l.cik, company: j.company]
          select [title: normTitle(j.appointedAs)]
          from JobChange j, PeopleLink l
          where j.docid = l.docid and j.span = l.span
            and  isOfficer(j.appointedAs) = true;
```

When multiple rules populate the same target entity, as in the case of rules m5 and m2, the resulting entity will contain a union of the results of those rules. When the entities are indexes, as in this case, we union all the keys as well as the sets of records stored under each key. We note that no new target data structures (entities or indexes) are necessary. The new rules simply insert new data into the same indexes declared by the initial mapping phase. This same pattern will typically apply when fusing any new data source. The *incremental* evaluation of such rules will be our focus in the rest of the chapter.

5.3 Problem Formalization

Our integration logic is specified in a HIL script using entity declarations and entity population rules, as we just described in Section 5.2. We denote a HIL script as H. Let $\mathbf{I}$ and $\mathbf{O}$ denote the input and output entity data, respectively. Let P_H be an integration process derived from H. When that process takes as input a set of entities $\mathbf{I}$ and produces $\mathbf{O}$, we write $\mathrm{P}_H(\mathbf{I}) = \mathbf{O}$.

Definition 7 (Declarative Incremental Integration). Given input entity data $\mathbf{I}$, let H be an integration script such that $\mathrm{P}_H(\mathbf{I}) = \mathbf{O}$. Given an additional input $\mathbf{\Delta I}$, possibly on a different schema than $\mathbf{I}$, and an additional script ΔH that applies on $\mathbf{\Delta I}$, the declarative incremental integration problem is to find an integration process P', such that $\mathrm{P}'_{\Delta H}(\mathbf{O}, \mathbf{\Delta I}) = \mathrm{P}_{H \cup \Delta H}(\mathbf{I} + \mathbf{\Delta I})$.[1]

Thus, the incremental integration process P' has access to the new rules ΔH, the new input data $\mathbf{\Delta I}$ and the result $\mathbf{O}$ of the previous integration (with H and $\mathbf{I}$). We do not assume that we have access to H or $\mathbf{I}$. This is important, since it makes the incremental integration process completely independent of the original integration in terms of both the input data, and the rules that were applied originally. Also, note that the above definition covers the incremental view maintenance scenario, when the source data changes but the rules and the source schema remain fixed. In such case, we set ΔH to be the same as H.

The equality condition at the end of the definition is a natural correctness condition which states that the result of the incremental process P' must be the same as applying the non-incremental process P on the union of $\mathbf{I}$ and $\mathbf{\Delta I}$, based on the union of the rules in H and ΔH.

One can interpret the definition in the context of our running example. Stage one in the example corresponds to $\mathrm{P}_H(\mathbf{I}) = \mathbf{O}$, where H is the set {m1-m4} of HIL rules, $\mathbf{I}$ is an IRP dataset and $\mathbf{O}$ is the first version of Person. In stage two, we have an additional script Δ_H given by the set {m5, m6} of HIL rules, and new input data $\mathbf{\Delta I}$ for JobChange and PeopleLink. Rather than executing $\mathrm{P}_{H \cup \Delta H}(\mathbf{I} + \mathbf{\Delta I})$ to obtain the second version $\mathbf{O}'$ of Person, we would like to find an incremental

[1]We use the plus sign to denote, in an intuitive way, the union of all the facts in the two datasets.

method P' such that $\mathrm{P}'_{\Delta H}(\mathbf{O}, \mathbf{\Delta I})$ evaluates to the same $\mathbf{O}'$.

Given an integration script H, we assume in the above definition that P_H is some deterministic procedure that implements H. A concrete choice for P_H is the result of *compiling* H to lower-level code, which can then be executed on a specific run-time engine. In the current implementation of HIL [48], H is compiled to a *Jaql* [19] script, which in turn runs on Hadoop. In this chapter, we will use a more general, two-phased semantics for H that is based on chase and de-reference (Section 5.5). Correspondingly, we will devise a method P' that is the incremental version of the two-phased semantics.

5.4 A NAIVE Approach

The critical question in the incremental integration problem is, given a ΔH and a $\mathbf{\Delta I}$, how do we reuse $\mathbf{O}$ to generate $\mathbf{O}'$. In this section, we start by analyzing a naïve answer, which later elicits our improved solution.

A natural idea is to apply ΔH directly on $\mathbf{\Delta I}$, and then "merge" the result with $\mathbf{O}$. Formally, we write this as:

$$\mathbf{O}' = \mathrm{P}'_{\Delta H}(\mathbf{O} + \mathbf{\Delta I}) \triangleq \mathbf{O} + \mathrm{P}_{\Delta H}(\mathbf{\Delta I}) \tag{5.1}$$

In Equation 5.1, we use the second plus sign to represent the "merge" process. However, the semantics of this operation are not trivial because of the data contains nested sets of records. We illustrate the complexity of this merge with the following example.

Example 7 (Naïve Approach). To illustrate the "merge" operation, we continue with our running example, and focus on a specific person (a person with a specific `cik`) from the `Person` entity generated in the first stage. We denote this person by `p` and display it inside the `Person` entity as a JSON object below:

```
Person:[
p: {"cik": 555,
    "name": "Bob Smith",
    "emp":  [{  "company": "Nest Bank",
                "positions": [
                {   "title": "CEO",
                    "earliest_date": 2008-12-31 }]  }]
    }...]
```

Suppose we have new information regarding this specific person from `JobChange`, as shown below:[2]

```
JobChange:[
j1: { "cik": 555, "name": "Bob Smith", "company": "Nest Bank", "appointedAs": "CEO", "
  apptDate": 2004-12-31}
j2: { "cik": 555, "name": "Bob Smith", "company": "Nest Bank", "appointedAs": "CFO", "
  apptDate": 2002-12-31}
j3: { "cik": 555, "name": "Bob Smith", "company": "Save Bank", "appointedAs": "CFO", "
  apptDate": 1998-12-31}
...]
```

By applying $\mathrm{P}_{\Delta H}(\cdot)$ on `JobChange`, we obtain Δ`Person`, which contains the incremental part (δ`p`) of the person:

```
ΔPerson:[
δp:{"cik": 555,
    "name": "Bob Smith",
    "emp": [{   "company": "Nest Bank",
                "positions": [
                {   "title": "CFO",
                    "earliest_date": 2002-12-31 },
                {   "title": "CEO",
                    "earliest_date": 2004-12-31 }]},
            {   "company": "Save Bank",
                "positions": [
                {   "title": "CFO",
                    "earliest_date": 1998-12-31 }]}]
    }...]
```

Now we merge δ`p` in Δ`Person` with `p` in `Person`. A natural way of doing this, which relies on recursively merging the set-valued attributes of any two records with the same atomic-valued attributes, yields a new person `p'` in `Person'`:

[2] For simplicity, we have joined `PeopleLink` into `JobChange` to associate `cik` with `JobChange` entries.

```
 1  Person’[
 2  p’ :{"cik": 555,
 3      "name": "Bob Smith",
 4      "emp": [{   "company": "Nest Bank",
 5                  "positions": [
 6                  {   "title": "CFO",
 7                      "earliest_date": 2002-12-31 },
 8                  {   "title": "CEO",
 9                      "earliest_date": 2004-12-31 }]},
10              {   "company": "Save Bank",
11                  "positions": [
12                  {   "title": "CFO",
13                      "earliest_date": 1998-12-31 }]}]
14      }...]
```

Note that this “merge” operation is non-trivial. This is primarily because such a “merge” occurs at each level of the entity. Specifically, since `j3` contains employment history from a new company (`"Save Bank"`), corresponding employment information has to be added to the `"emp"` set in `p` (line 10-13 in `p’`). Moreover, because `j2` reveals that this person was once appointed as the `CFO` in `Nest Bank`, the `positions` set inside `emp` in `p` has to be enriched accordingly (line 6-7 in `p’`). Finally, for the same company `Nest Bank` and same position `CEO`, we need to merge the entry in `positions` of `p` (line 6-7) with that of δp (line 8-9), and update the `earliest_date` to `2004-12-31` in `p’` (line 9). This update is challenging, because it applies the incremental semantics (a binary `min`) of the aggregation function `minDate`. In case the function is holistic, the update will simply fail.

In summary, to perform such a “merge”, we need to traverse the structure of `p` and δp simultaneously (starting from `emp` in this case). If δp contains a new entry (entry with a new set of values for the atomic fields defined in the rules, such as `company:"Save Bank"`), we add it to the corresponding set in `p`. Otherwise, the entries from δp and `p` contain the same set of values for those atomic fields (`company:"Nest Bank"`), thus we recursively perform the “merge” for any index-lookup attributes (Positions in this case). Moreover, atomic values reduced by aggregate functions have to be merged according to the incremental semantics of such functions. In essence, this is a recursive union procedure. To distinguish it from traditional union on flat sets, we term it ***deep-union*** and denote it by $\uplus$.

We note that deep-union has been used in various contexts related to merging of nested data (e.g., [22]). It is also related to the partitioned-normal form [4] that requires that the atomic-valued attributes in a set of records functionally determines the set-valued attributes, at any level.

With deep union, we can rewrite Equation 5.1 as:

$$\mathbf{O}' = \mathrm{P}'_{\Delta H}(\mathbf{O} + \mathbf{\Delta I}) \triangleq \mathbf{O} \uplus \mathrm{P}_{\Delta H}(\mathbf{\Delta I}) \tag{5.2}$$

Given that our output entity $\mathbf{O}$ is materialized, this kind of deep-union is extremely inefficient. This is because one has to drill into the hierarchical structure in $\mathbf{O}$ and append or update deeply nested data. Moreover, merging $\mathbf{O}$ and $\mathrm{P}_{\Delta H}(\mathbf{\Delta I})$ with deep-union does not necessarily produce the same result as applying $\mathrm{P}_{H \cup \Delta H}$ on $\mathbf{I} + \mathbf{\Delta I}$. This is because HIL allows for a finer-grained merging of nested data based on indexes. For example, even if the name in δp is different (possibly due to data inconsistencies), as long as δp has the same cik, HIL requires it to be merged with p, since the employment field only depends on cik. Finally, if information is previously aggregated, deep-union needs to incrementally update these aggregated values, which can be extremely challenging. These observations motivate us to explore a different incremental integration strategy.

5.5 Chase and De-reference

One reason incremental integration is challenging is that, once the nesting structure of the output entities is fully materialized, it becomes hard to reuse its content for subsequent integration stages. This leads us to the following question: is there a way to generate partial integration results that have a relatively simple structure and can be easily assembled to produce the complete output? If so, subsequent integration stages may benefit from this by reusing these simple partial results instead of the fully nested output. In this section, we first propose a method for generating such simpler partial results, and then describe the way to assemble them.

5.5.1 Chase

We borrow the spirit of decorrelation and decomposition of rules from HIL into our incremental evaluation strategy. To be more specific, given an input $\mathbf{I}$ and a set of HIL rules H, we apply each HIL rule in H independently on $\mathbf{I}$ to generate a partial flat output, so that the atomic attributes in the select clause of the rule are populated with actual data from $\mathbf{I}$, while the index-referring attributes are encoded with corresponding index references. We call this process **chase** since it has some similarity to the well-know chase operation (particularly the chase with second-order tgds [39]). Formally, we call this process chase $\mathbf{I}$ with H, and denote it by $\mathrm{C}_H(\mathbf{I})$. The following example illustrates how chase works.

Example 8 (Chase). Without loss of generality, we illustrate how one can chase the specific person in example 7, by focusing on its relevant entry in IRP:

```
IRP:[
i:  { "cik": 555, "name": "Bob Smith", "company": "Nest Bank", "title": "CEO", "
  reportingDate": 2004-12-31}
...]
```

We first chase IRP with m1, and obtain a flat version of Person. To do so, for each entry (e.g., i), we project its atomic attributes selected in m1 (e.g., cik and name), while populating the nested set (e.g., emp) with an *index reference*, which is a record consisting of the name of the referred index and a lookup key. We denote the chase result by Person$_c$ and show it below. Note that the reference in emp (highlighted by *REF*) says the actual content of emp can be found in an index Employment$_c$ with the key {"cik":555}.

```
Personc:[
p1: {   "cik": 555,
        "name": "Bob Smith",
  REF:  "emp": {    "index": "Employmentc",
                    "key": {"cik": 555}     }   }...]
```

Employment$_c$ can be obtained in a similar manner by chasing IRP using m2. Since Employment is an index itself, the chase result is a set of key-value pair. The key is the index lookup key to its set-valued entry, and each element in the value set is populated in the same way as before:

```
Employmentc:[
e1: {"key": {"cik": 555},
     "values": [{
        "company": "Nest Bank",
  REF:  "positions":{   "index": "Positionsc",
                        "key": {"cik": 555,
                                "company": "Nest Bank"}
                    }   }]  }...]
```

We continue to chase IRP using m3, and obtain Positions$_c$. Since we have employed UDFs for aggregation, the name of such UDFs (e.g., function) must be provided as well:[3]

[3]Hereafter, we omit similar references with dots due to space.

```
Positions_c:[
po_1: {  "key": { "cik": 555,
                  "company": "Nest Bank" },
        "values": [{
            "title": "CEO",
      REF:  "earliest_date": {
                     "index": "PositionInfo_c",
                     "key": { "cik": 555,
                              "company": "Nest Bank",
                              "title": "CEO" },
                     "function": "minDate" } }]  }...]
```

Finally, we chase IRP with m4 in the same way, and produce $\mathsf{PositionInfo_c}$. We omit the detail due to space.

So far we have chased IRP using m1-m4 into a series of partial integration results:

$$\mathrm{C}_H(\mathsf{IRP}) = \{\mathsf{Person_c}, \mathsf{Employment_c}, \mathsf{Positions_c}, \mathsf{PositionInfo_c}\} \tag{5.3}$$

We emphasize that, in contrast to the fully nested integration result, these chase results are flat data structures. That is, for each entry in the top-level entity and in the value set of each index in the chase results, only atomic values and index references are recorded. Future integration stages can benefit from this fact, and will utilize these flat structures as partial results instead of using a fully nested structure, as we will explain in Section 5.6.

5.5.2 De-reference

Chase produces partial results that are de-correlated from each other by index references. In contrast, one can chase the pointers (or references) and inline them with the corresponding values to construct the complete result. We name this process **de-reference**. For example, if a $\mathsf{Person_c}$ has a reference to an index $\mathsf{Employment}$, then de-referencing means, first making sure that all the entries in Employment are de-referenced (recursively), then replacing the reference inside the $\mathsf{Person_c}$ with the value that is obtained by looking up the de-referenced entry in Employment with the given key. If needed, we must also apply the function that may be encoded with the reference.

Note that it is possible for different entries in $\mathsf{Person_c}$ to refer to different indexes. For instance, suppose we have another rule m1′, which is identical to m1 except that it uses a new index Employment1 to populate Person. In this case, there also exist records in $\mathsf{Person_c}$ that refer to Employment1 from the emp field. Then de-referencing will use the (de-referenced) value from Employment1 (not Employment).

Example 9 (De-reference). Here we de-reference the chase results from Example 8. We start from the deepest level of the recursion, where we de-reference $\mathsf{Positions_c}$ with $\mathsf{PositionInfo_c}$. According to the reference in the earliest_date field in $\mathsf{Positions_c}$, we look up for the corresponding entry in $\mathsf{PositionInfo_c}$, with the key consisting of three attributes (cik, company and title). The retrieved entry is a set of dates. Since we have a UDF minDate here, we further apply this function to the set of dates, and use the result to replace the value of earliest_date in $\mathsf{Positions_c}$.

We continue to de-reference $\mathsf{Employment_c}$ with the de-referenced $\mathsf{Positions_c}$. For the positions field in $\mathsf{Employment_c}$, we use its key to look for the corresponding entry in $\mathsf{Positions_c}$. Since no function is presented, we directly replace the positions field in $\mathsf{Employment_c}$ with the retrieved value (the set of positions). Finally, we proceed with $\mathsf{Person_c}$ in a similar manner.

By denoting the de-reference procedure by D, the above process can be written as:

$$\mathrm{D}(\mathsf{Person_c}, \mathsf{Employment_c}, \mathsf{Positions_c}, \mathsf{PositionInfo_c}) = \mathsf{Person} \tag{5.4}$$

While conceptually simple, the actual execution of such a de-reference procedure can be highly inefficient (esp. with a recursive implementation). In Section 5.7, we describe a non-recursive implementation of de-reference. There we will describe how we abstract the de-reference logic into a data structure (namely *reference summary*), which is stored together with $\mathrm{C}_H(\mathbf{I})$ and can be compiled into a set of optimized queries using join operations. We also show how the reference summary can be maintained in an incremental scenario, with significant performance benefits.

To end this section, we summarize the chase and de-reference procedure with the following equation:

$$\mathrm{D}(\mathrm{C}_H(\mathbf{I})) = \mathrm{P}_H(\mathbf{I}) = \mathbf{O} \tag{5.5}$$

Equation 5.5 suggests, first, a more systematic way to perform integration tasks. That is, we can always chase the input using individual HIL rules, produce partial and flat results that encode the result via index references, and then de-reference the chase results to construct the final output. Secondly, and of particular importance for incremental integration, the partial results that result after chasing have the advantage that they can be more easily and efficiently merged with new data resulting from new rules or new source data. Taking advantage of this, we outline our incremental solution in the next section.

5.6 Incremental Integration

We now resume our discussion on the declarative incremental integration problem. To recall, given an additional input $\mathbf{\Delta I}$ and an additional integration script ΔH, our baseline procedure integrates everything from scratch:

$$\mathbf{O}' = \mathrm{P}_{H\cup\Delta H}(\mathbf{I} + \mathbf{\Delta I}) \tag{5.6}$$

Given the previous definitions of chase and de-reference, we can decompose this procedure into the two phases:

$$\mathbf{O}' = \mathrm{D}(\mathrm{C}_{H\cup\Delta H}(\mathbf{I} + \mathbf{\Delta I})) \tag{5.7}$$

Since our chase is distributive (a property which we will explain shortly), we can rewrite Equation 5.7 as:

$$\mathbf{O}' = \mathrm{D}(\mathrm{C}_H(\mathbf{I}) + \mathrm{C}_{\Delta H}(\mathbf{\Delta I})) \tag{5.8}$$

Notice that here $\mathrm{C}_H(\mathbf{I})$ represents the result of chase over the previous input, which we assume is now materialized and can be reused. Equation 5.8 suggests that we can incrementally compute $\mathbf{O}'$ by computing $\mathrm{C}_H(\mathbf{I})$, the chase with the new logic over the new data increment, followed by a "merge" between the two chase results. Since computing $\mathrm{C}_H(\mathbf{I})$ follows the same chase procedure we discussed in Section refsec:chase, we concentrate here on the "merge" semantics.

Since the results of our chase are flat, the semantics of merge are almost the same as the usual set union. When merging top-level entities such as Person_c, merge behaves exactly as a set union. When entities are indexes (e.g., $\mathsf{Employment}_c$), the semantics is a little richer: for each key, we union the associated values from both indexes (each is a set of entries). The resulting set is then associated to the same key.

Example 10 (Merge Chase Results). We now illustrate the merge using our running example. We first compute $\mathrm{C}_{\Delta H}(\mathbf{\Delta I})$, where $\mathbf{\Delta I}$ is the JobChange including `j1-j3` described in Example 7. By chasing this JobChange with m6, we generate

$\Delta\mathsf{Positions}_c$:

```
ΔPositions_c:[
δpo₁: { "key": { "cik": 555,
                 "company": "Nest Bank" },
        "values": [{
            "title": "CEO",
      REF: "earliest_date": {
                    "index": "PositionInfo_c",
                    "key": { "cik": 555,
                             "company": "Nest Bank",
                             "title": "CEO" },
                    "function": "minDate" } },
        {   "title": "CFO",
      REF: "earliest_date": {...}  }]  }
δpo₂: { "key":  { "cik": 555,
                  "company": "Save Bank" },
        "values": [{
            "title": "CFO",
      REF: "earliest_date": {...}  }]  }...]
```

We then merge $\Delta\mathsf{Positions}_c$ with $\mathsf{Positions}_c$ (see Example 8). For each key, we retrieve the `values` sets from both indexes, union them and associate the result with the same key. For example, for the key `{"cik":555, "company":"Nest Bank"}`, since the `values` in po_1 is a proper subset of the `values` in $\delta\mathsf{po}_1$, the result is identical to that in $\delta\mathsf{po}_1$.

Similarly, chasing JobChange with m5 produces $\Delta\mathsf{Employment}_c$:

```
ΔEmployment_c:[
δe1:{"key": {"cik": 555},
     "values": [{
        "company": "Nest Bank",
  REF:  "positions":
            {   "index": "Positions_c",
                "key": {"cik": 555,
                        "company": "Nest Bank"} } },
   {   "company": "Save Bank",
  REF:  "positions": {...}  }]  }...]
```

Since for the key {"cik":555}, the values in e_1 is a proper subset of that in δe_1, merging Employment_c with $\Delta\text{Employment}_c$ results in a $\text{Employment}'_c$ that is equal to $\Delta\text{Employment}_c$. Since ΔH does not modify Person or PositionInfo, Person_c and PositionInfo_c remain the same after the merge.

We call this merge operation **c-union** and denote in the rest of the chapter as $\sqcup$. Although simple, the semantics of c-union is important to our incremental solution with chase and de-reference, because it guarantees the *distributivity* of our chase operation. That is, chasing $\mathbf{I} + \mathbf{\Delta I}$ produces the same result as taking a c-union of the results of chasing $\mathbf{I}$ and $\mathbf{\Delta I}$ independently. Formally, we express this as:

$$\mathrm{C}_{H\cup\Delta H}(\mathbf{I} + \mathbf{\Delta I}) = \mathrm{C}_{H\cup\Delta H}(\mathbf{I}) \sqcup \mathrm{C}_{H\cup\Delta H}(\mathbf{\Delta I}) \tag{5.9}$$

Furthermore, because $\mathbf{I}$ is independent of ΔH and $\mathbf{\Delta I}$ is independent of H, we have:

$$\mathrm{C}_{H\cup\Delta H}(\mathbf{I} + \mathbf{\Delta I}) = \mathrm{C}_{H}(\mathbf{I}) \sqcup \mathrm{C}_{\Delta H}(\mathbf{\Delta I}) \tag{5.10}$$

By substituting equation 5.10 into Equation 5.7, we obtain:.

$$\mathbf{O}' = \mathrm{D}(\mathrm{C}_H(\mathbf{I}) \sqcup \mathrm{C}_{\Delta H}(\mathbf{\Delta I})) \tag{5.11}$$

Equation 5.11 completes our definition of the incremental chase and de-reference solution.

After doing the the c-union between the old and new chase results, we de-reference the results. Recall, however, that the naïve de-reference procedure can be highly inefficient. We now describe an optimized de-reference algorithm based on efficient join queries.

5.7 Optimizing De-Reference

In order to optimize the de-reference procedure, we translate it into a list of optimized non-recursive queries. We first give an intuition about what these queries look like.

Example 11 (De-Reference with Optimized Queries). We now show how to express the de-reference task in Example 9 as a sequence of optimized queries. We choose a bottom-up approach to avoid any recursion. We omitted PositionInfo_c in Example 9 and, thus, we start here by de-referencing Employment_c with an already de-referenced Positions_c.

Intuitively, the de-reference process approximates a join operation, where we join Employment_c with Positions_c by matching the key in the `positions` field of Employment_c with the top-level key in Positions_c. We then project the `values` set from Positions_c to the `positions` field in Employment_c, with the other fields of Employment_c intact.

However, we note that, the key we use in the join condition from Employment$_c$ is embedded in the values field, which contains a set of entries. As a result, we have to "flatten" Employment$_c$ before joining it with Positions$_c$. Here "flatten" means, for each entry in Employment$_c$, we natural join the values set with the top-level key and replace the entry with the result set. Since the semantics of such a "flatten" operation is similar to the Unnest operator in traditional nested relational algebra [**?**, 80, 75], we borrow the same notation here and denote the flatten process as:

$$\mathsf{Employment_c} \leftarrow \mathrm{Unnest}_{key}(\mathsf{Employment_c}) \tag{5.12}$$

We then specify the join query as follows:

$$\begin{aligned}&\mathsf{Employment_c} \leftarrow \mathsf{Employment_c} \bowtie \mathsf{Positions_c}\\ &\qquad |\mathsf{Employment_c}.positions.key = \mathsf{Positions_c}.key\end{aligned} \tag{5.13}$$

After this, we have to recover the original nested structure for Employment$_c$, by "re-grouping" the entries under the same key. Similarly, we borrow the Nest operator from nested relational algebra and write this process as:

$$\mathsf{Employment_c} \leftarrow \mathrm{Nest}_{key}(\mathsf{Employment_c}) \tag{5.14}$$

This completes the de-reference process for Employment$_c$. We then move up to Person$_c$ and de-reference it with the de-referenced Employment$_c$. Since Person$_c$ is the top-level entity with a flat structure, we no longer need to unnest and nest it.

We only need the following join query:

$$\mathsf{Person_c} \leftarrow \mathsf{Person_c} \bowtie \mathsf{Employment_c}$$
$$|\mathsf{Person_c}.emp.key = \mathsf{Employment_c}.key \quad (5.15)$$

We also note, these join queries can be easily extended to handle UDFs, regardless of the property of these functions (whether aggregative, whether holistic, etc.). Given a UDF, we simply apply it on the `values` set joined from the referred index before projecting it to the output.

Example 11 depicts the types of non-recursive join queries that efficiently inline all index references. To derive these queries, it is necessary to maintain some information on the kind of references inside each entity. We store this information in a data structure called *reference summary* and note that we only need this structure at compile time.

5.7.1 Reference Summary

A **reference summary** is a graph of the reference relationships in a decomposed nested entity. We start by defining its nodes and edges.

Definition 8 (Entity Node). An entity node $\mathbf{N}$ is an object with a name and a set of attributes A. We denote the name by $\mathbf{N}.name$ and the set by $\mathbf{N}.attrs$, where $\mathbf{N}.attrs = \{A_1, A_2, ..., A_{M_N}\}$. We refer to an attribute A_i by $\mathbf{N}.attrs.A_i$.

Given an integration script H, each HIL rule naturally translates into an entity node. For instance, we create an entity node $\mathbf{N}_{\mathsf{m1}}$ for `m1` and assign it the name `Person` and the attribute set `{name, cik, emp}`. Similarly, $\mathbf{N}_{\mathsf{m2}}$ has name `Employment` and attribute set `{company, companyCik, positions}`. We denote the set of all entity nodes by $\mathbf{U}$.

Definition 9 (Super Node). A super node $\mathbf{S}$ is an object with a name $\mathbf{S}.name$, a flag $\mathbf{S}.isIndex$ and a set of entity nodes $\mathbf{S}.nodes$. The flag is true iff the corresponding entity is an index, and $\mathbf{S}.nodes = \{\mathbf{N} | \mathbf{N} \in \mathbf{U} \wedge \mathbf{N}.name = \mathbf{S}.name\}$

Super nodes are important to capture the fact that several integration rules may use different indexes to populate the same entity. For example, m1 and m1′ (see Section 5.5.2) both populate Person but with different employment indexes. If we denote the super node for Person by $\mathbf{S}_{\mathsf{Person}}$, then we have $\mathbf{S}_{\mathsf{Person}}.nodes = \{\mathbf{N}_{\mathsf{m1}}, \mathbf{N}_{\mathsf{m1'}}\}$. We denote the set of super nodes derived from H by $\mathbf{V}$. Given a super node $\mathbf{S}$, we use $\mathbf{S}_c$ to refer to the chase result corresponding to the entity with $\mathbf{S}.name$.

Definition 10 (Reference Edge). A reference edge $\mathbf{E}$ is a directed edge pointing from a super node $\mathbf{S}$ to an attribute A in $attrs$ of some entity node $\mathbf{N}$, and annotated with a function name. We denote the super node by $\mathbf{E}.from$, the attribute by $\mathbf{E}.to$ and the function by $\mathbf{E}.func$.

Reference edges reflect how the entities refer to each other. For instance, if we denote the super node of Employment by $\mathbf{S}_{\mathsf{Employment}}$, m1 implies a reference edge $\mathbf{E_1}$, pointing from $\mathbf{S}_{\mathsf{Employment}}$ to $\mathbf{N}_{\mathsf{m1}}.attrs.emp$. Since no UDF is applied here, $\mathbf{E_1}.func = null$. We denote the set of all edges by $\mathbf{W}$. Given an edge $\mathbf{E}$ and a super node $\mathbf{S}$, we write $\mathbf{E} \rightarrow \mathbf{S}$, if $\exists \mathbf{N} \in \mathbf{S}.node$, s.t. $\mathbf{E}.to \in \mathbf{N}.attrs$.

Definition 11 (Reference Summary). A reference summary $\mathbf{K}$ is a triple $(\mathbf{U}, \mathbf{V}, \mathbf{W})$.

Figure 5.2 depicts an example reference summary corresponding to our running example, where structures with solid line represent the de-reference logic implied by m1-m4, and the ones with dashed line come from additional rules (m1′,

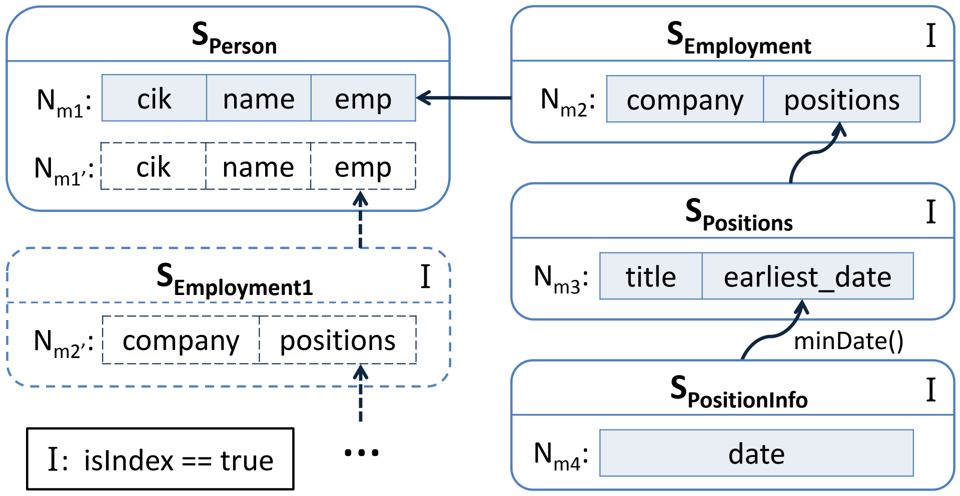


Figure 5.2: An Example Reference Summary

m2′, etc.). Note that in S_{Person}, we have two entity nodes, N_{m1} and $N_{m1'}$. While the emp attribute in N_{m1} is pointed by $S_{Employment}$, the emp in $N_{m1'}$ is pointed by $S_{Employment1}$. This reflects the fact that the same field emp in Person is populated by different indexes, and indicates, during de-referencing, how to inline the correct index to construct the fully nested output.

5.7.2 De-reference With Reference Summary

Given a reference summary, we can compile it to a sequence of optimized queries by first traversing its super nodes, and then de-referencing the entity represented by each node. In principle, the de-reference result is order-independent because the chase results are de-correlated from each other. However, we note that a top-down manner can be computationally expensive (as the recursive implementation), since the deeper in the summary it goes, the deeper in the entity structure where de-reference occurs. Thus we choose a bottom-up traverse order.

Intuitively, the generation of a traverse order on a reference summary graph is similar to a topological sort on a DAG. Thus here we present it by reusing a topological sort. To do this, we first construct a new edge set $\mathbf{W_T} \leftarrow \{(\mathbf{S_1}, \mathbf{S_2}) | \mathbf{S_1}, \mathbf{S_2} \in \mathbf{V} \wedge \exists \mathbf{E} \in \mathbf{W} s.t. \mathbf{E}.from = \mathbf{S_1} \wedge \mathbf{E} \rightarrow \mathbf{S_2}\}$. We can then employ any standard topological sort algorithm to generate an order of super nodes following the directions in the newly defined edge set. We denote this process by: $\mathbf{L} \leftarrow \mathrm{TraverseOrder}(\mathbf{K}) \triangleq \mathrm{TopologicalSort}(\mathbf{V}, \mathbf{W_T})$.

We now process the super node $\mathbf{S}$ from $\mathbf{L}$ in order. For each $\mathbf{S}$, we invoke the procedure in Algorithm 8.

Algorithm 8 De-reference Entity Node

```
1:  procedure De-reference($\mathbf{S}$)
2:      $P \leftarrow$ empty entity                                   ▷ Initialize
3:      for $\mathbf{N} \in \mathbf{S}.nodes$ do                         ▷ Iterate over entity nodes
4:          $Q \leftarrow \mathbf{S}_c$                                  ▷ Retrieve the chase result
5:          if $\mathbf{S}.isIndex$ then                              ▷ If index, flatten it
6:              $Q \leftarrow$Unnest($Q$,value)
7:          end if
8:          for $\mathbf{E} \in \{\mathbf{E} | \mathbf{E} \in \mathbf{W} \wedge \mathbf{E}.to \in \mathbf{N}.attrs\}$ do
9:              $\mathbf{T} \leftarrow \mathbf{E}.from$
10:             $A \leftarrow \mathbf{E}.to$
11:             $R = \mathbf{T}_c$                                   ▷ Get the referenced entity
12:             $Q \leftarrow Q \bowtie R | Q.value.A.key = R.key$, $\mathbf{E}.func$   ▷ Join
13:         end for
14:         if $\mathbf{S}.isIndex$ then                              ▷ If index, re-group it
15:             $Q \leftarrow$Nest($Q$,key)
16:         end if
17:         $P \leftarrow P \sqcup Q$                                  ▷ C-Union results
18:     end for
19:     $\mathbf{S}_c \leftarrow P$                                      ▷ Update the chase result
20: end procedure
```

Given a super node $\mathbf{S}$, Algorithm 8 iterates over the entity nodes under $\mathbf{S}$ (line 3). For each entity node $\mathbf{N}$ and each reference inside $\mathbf{N}$ (line 8), we retrieve the referenced chase result (already de-referenced according to traversing order) and join it into the referring attribute (line 12). The function, if presented, will be applied during the join. Since join only works on flat structure, if $\mathbf{S}$ is an index, we unnest it before the join and nest it after the join (line 6, 15). Finally, we c-union the de-reference results from each entity node (line 17) and update the chase result (line 19).

5.7.3 Maintain Reference Summary

In an incremental scenario, as the de-reference logic is being enriched, the summary is also subject to evolution. Instead of computing the new summary from scratch, it is preferable to incrementally maintain the old summary. Formally, if we denote the incremental part of the summary by $\mathbf{\Delta K}$ and the new summary by $\mathbf{K}'$, our goal is to find a way to merge $\mathbf{K}$ with $\mathbf{\Delta K}$ to produce $\mathbf{K}'$.

Example 12 (Merge Reference Summaries). Suppose in a following incremental stage, we absorb a new data set which includes information about the company boards the person has been on. Thus we have a new index Board, together with a new rule m7, which is identical to m1 except for that it adds a new attribute board to Person and populates it with an index look-up in Board with cik.

We first generate $\mathbf{\Delta K}$, which contains the corresponding super nodes and entity nodes for Board and Person, together with new reference edges indicated by m7. We then merge $\mathbf{\Delta K}$ onto $\mathbf{K}$ by traversing $\mathbf{\Delta K}$ using the order described before. For Board, since it is new, the corresponding super node and entity node, together with relevant edges are directly copied from $\mathbf{\Delta K}$ to $\mathbf{K}$. For Person, however, because it exists before, we have to merge the old super node $\mathbf{S}_{\mathsf{Person}}$ with the new one $\mathbf{S}'_{\mathsf{Person}}$. We start by merging the entity nodes. Since the new entity node $\mathbf{N}_{\mathsf{m7}}$ from $\mathbf{S}'_{\mathsf{Person}}$ contains a new attribute board, we have to *expand* the old entity node $\mathbf{N}_{\mathsf{m1}}$ accordingly. After this, we merge the two nodes with relative edges updated.

We note here, not every new entity node can be merged with an old one. For instance, if H contains m1′, $\mathbf{N}_{\mathsf{m1'}}$ cannot be merged with $\mathbf{N}_{\mathsf{m7}}$, since m1′ populates emp with a different index (Employment1). In contrast, $\mathbf{N}_{\mathsf{m1}}$ can be merged with $\mathbf{N}_{\mathsf{m7}}$, since the attributes and referenced indexes from both nodes are identical except for the new board. Formally, we say $\mathbf{N}_{\mathsf{m1}}$ is contained in $\mathbf{N}_{\mathsf{m7}}$ given $\mathbf{W}$ and $\mathbf{\Delta W}$, or $\mathbf{N}_{\mathsf{m1}} \preceq \mathbf{N}_{\mathsf{m7}}|\mathbf{W}, \mathbf{\Delta W}$. Given this notion of containment, we describe our merge in Algorithm 9.

Algorithm 9 Merge Reference Summary

```
procedure MergeSummary($\mathbf{K}$,$\mathbf{\Delta K}$)
    $\mathbf{L} \leftarrow$ TraverseOrder($\mathbf{\Delta K}$)                          ▷ Sort new nodes
    while $\neg\mathbf{L}.isEmpty()$ do
        $\mathbf{S}' \leftarrow \mathbf{L}.head()$
        $\mathbf{S} \leftarrow \mathbf{S}|\mathbf{S} \in \mathbf{V} \wedge \mathbf{S}.name = \mathbf{S}'.name$          ▷ Find match
        if $\mathbf{S} = null$ then                                  ▷ Add super node
            $\mathbf{V} \leftarrow \mathbf{V} \cup \{\mathbf{S}'\}$, $\mathbf{U} \leftarrow \mathbf{U} \cup \{\mathbf{S}'.nodes\}$
            $\mathbf{W} \leftarrow \mathbf{W} \cup \{\mathbf{E}|\mathbf{E} \in \mathbf{\Delta W} \wedge \mathbf{E} \rightarrow \mathbf{S}'\}$
        else                                                 ▷ Merge super node
            $\mathbf{R} \leftarrow \{\mathbf{E}|\mathbf{E} \in \mathbf{W} \wedge \mathbf{E}.from = \mathbf{S}\}$
            for $\mathbf{E} \in \mathbf{R}$ do                               ▷ Update outgoing edges
                $\mathbf{E}.from \leftarrow \mathbf{S}'$
            end for
            for $\mathbf{N} \in \mathbf{S}.nodes$ do                          ▷ Update entity nodes
                $\mathbf{N}' \leftarrow \mathbf{N}'|\mathbf{N}' \in \mathbf{S}'.nodes \wedge \mathbf{N} \preceq \mathbf{N}'|\mathbf{W}, \mathbf{\Delta W}$
                if $\mathbf{N}' = null$ then                         ▷ Expand old entity node
                    $\mathbf{S}.nodes \leftarrow \mathbf{S}.nodes \backslash \{\mathbf{N}\}$
                    $\mathbf{N} \leftarrow$ Expand($\mathbf{N}$)
                    $\mathbf{S}'.nodes \leftarrow \mathbf{S}'.nodes \cup \{\mathbf{N}\}$
                else                                         ▷ Merge entity nodes
                    $\mathbf{R} \leftarrow \{\mathbf{E}|\mathbf{E} \in \mathbf{W} \wedge \mathbf{E}.to \in \mathbf{N}.attrs\}$
                    for $\mathbf{E} \in \mathbf{R}$ do
                        $A \leftarrow \mathbf{E}.to$, $\mathbf{E}.to \leftarrow \mathbf{N}'.attrs.A$
                    end for
                    $\mathbf{U} \leftarrow \mathbf{U} \backslash \{\mathbf{N}\}$, $\mathbf{U} \leftarrow \mathbf{U} \cup \{\mathbf{N}'\}$
                end if
            end for
            $\mathbf{V} \leftarrow \mathbf{V} \backslash \{\mathbf{S}\}$, $\mathbf{V} \leftarrow \mathbf{V} \cup \{\mathbf{S}'\}$                       ▷ Move node
            $\mathbf{W} \leftarrow \mathbf{W} \cup \{\mathbf{E}|\mathbf{E} \in \mathbf{\Delta W} \wedge \mathbf{E} \rightarrow \mathbf{S}'\}$           ▷ Add edges
        end if
    end while
end procedure
```

We first retrieve the super nodes from $\mathbf{\Delta K}$ according to the order described before (line 2). For each node $\mathbf{S}'$, we check if there exists a node $\mathbf{S}$ in $\mathbf{K}$ representing the same entity (line 5). If no, we add this super node, its member entity nodes together with its relevant edges to $\mathbf{K}$ (line 7-8). Otherwise, we update the structure as follows.

We merge $\mathbf{S}$ and $\mathbf{S}'$ by moving relevant structure from $\mathbf{S}$ to $\mathbf{S}'$. For inter-node structure, for each edge pointing from $\mathbf{S}$, we change their source to $\mathbf{S}'$ (line 10-12). For any edge that points to some entity node in $\mathbf{S}'$, we add them to $\mathbf{K}$ (line 29). For intra-node structure, for each entity node $\mathbf{N}$ in $\mathbf{S}$ we check if there is any entity node $\mathbf{N}'$ in $\mathbf{S}'$ that contains $\mathbf{N}$ (line 15). If no, we expand $\mathbf{N}$ and move it to $\mathbf{S}'$ (line 17-19). Otherwise, we re-link relative edges so that edges originally pointing to $\mathbf{N}$ now point to $\mathbf{N}'$ (line 21-24). Finally, we add the updated $\mathbf{S}'$ and related structure to $\mathbf{K}$ (line 28-29). We proceed the rest nodes in the same way.

5.8 Evaluation

We now describe how we evaluated our chase and de-reference approach using the complete SEC financial integration scenario. Specifically, we compared our chase and de-reference approach to the non-incremental approach in two scenarios. First, a “view adaptation” scenario where new types of data arrive and the integration logic changes. Then, a “view update” scenario where only new increments of the existing input entities arrive and need to be integrated into the existing target entities. All experiments were run on an IBM System x3550 with 2 CPUs (4 cores each) and 32 GB main memory. In all cases, the rules were expressed in HIL and compiled down into Jaql [19] scripts.

5.8.1 Integration in Incremental Stages

In this experiment, we divide the incremental integration into five stages (Table 5.1). The first stage is the initial mapping from Person into IRP. The specification of this stage largely follows our description in Section 5.2, but includes additional rules and entities to produce an extra board field in Person, as well as rules to handle its provenance and temporal aggregation. The input IRP consists of $297,358$ records, extracted from all documents with insider transactions of executives in the finance industry from 2005 to 2010. This input produced $32,816$ Person records.

In the subsequent stages, we incrementally add new data sources and new rules to incorporate the sources into Person. Since entity resolution is not our focus, we assume all links resolving the new sources to Person are already computed. For stage two, JobChange data was incorporated using 5 new rules that updated both Employment and Board. In stage three, a Committee data source was added using 3 more rules that updated 2 existing entities and added a new Committees entity containing information about the committees the person has been involved on each company. Stage four enriched Person with biographies from Bio. The biography information is split for officers and non-officers, and fused into the bios field in Board and Employment respectively. Finally, in stage five, we improved Person with an additional source Signatures, which are records extracted from a special signature section of a certain type of input documents, and provides additional information about key people and their employment. A summary of these integration stages is shown in Table 5.1.

Stage	Data Source	# Records	# Attrs	# Rules	# Updated Entities	# New Entities
1	IRP	$297,358$	9	6	0	0
2	JobChange	$1,077$	7	5	5	0
3	Committee	$63,297$	10	3	2	1
4	Bios	$23,195$	9	5	3	2
5	Signatures	$319,154$	11	5	5	0

Table 5.1: Summary of SEC Integration Stage

For each integration stage, we compared the execution time for three integration approaches (Figure 5.3). First, we measured the total execution time for the **baseline** integration process (the left-most bar for each stage in Figure 5.3), which compiled the whole integration logic into executable scripts and ran everything from scratch (see P_H in Section 5.3). Second, we evaluated our non-incremental chase and de-reference approach. That is, we chased all the input with the whole integration logic upon the current stage, and then de-referenced the chase results (Equation 5.7). We measured the execution time for both the chase and de-reference procedures (the second bar for each stage in Figure 5.3) . Finally, we used our incremental chase on the new data, merged the results with previous chase results, and then de-referenced the results (Equation 5.11). We measured the total time for chase and merge, and the total time for de-reference. (This is represented by the last bar in Figure 5.3. Notice that, this bar is missing in stage 1 since there is no previous data to merge.)

We observed a significant reduction in total time with our incremental chase and de-reference approach, compared with the baseline approach and the non-incremental chase and de-reference approach. Specifically, the total execution time for the incremental chase and de-reference approach is only $\mathbf{23\% - 36\%}$

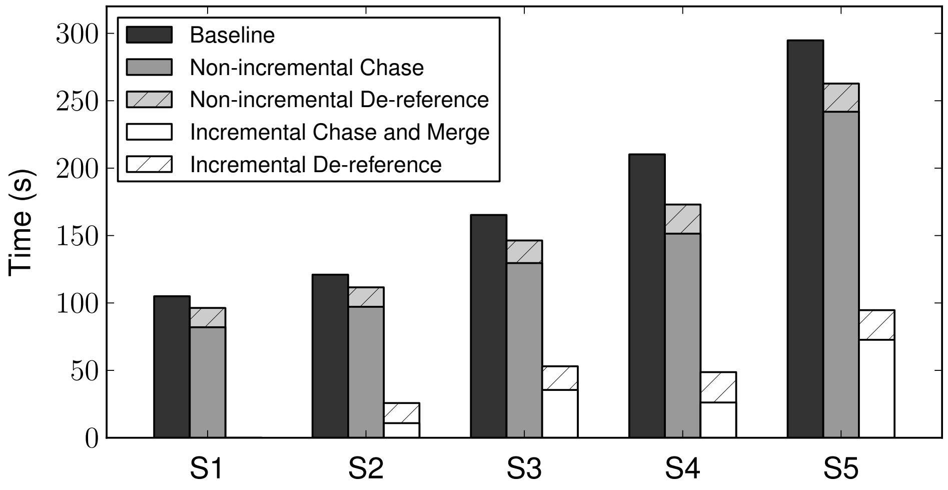


Figure 5.3: Integration Times in SEC Integration Stages

of the total time for the baseline approach. The difference is due to the fact that our incremental approach only computes the chase for the new data and merges the results to the existing data. This can be clearly seen if we compare the bars of the non-incremental approach with those of the incremental approach. Notice that most of the time in the non-incremental approach is spent doing the chase. In fact, the de-reference stage time is same for both the non-incremental and incremental cases. Both need to put together the same chase results into the nested structure, the main difference here being that in the incremental case, a large portion of those chase results were computed in the previous stage.

5.8.2 Integration with Incremental Data

In this section we study the behavior of our incremental integration approach when the integration logic is fixed and only increments of new data for the existing entities arrive. In this study we only used the six rules that populate Person from IRP. To simulate the arrival of incremental data, we partitioned the IRP records into two equal parts. We use the first half of IRP as the initial input from which we computed an initial version of Person. The second half of IRP records are subdivided into ten equal pieces, each of which is used as a new increment of IRP data that must be merged to the existing Person. We simulated the arrival of new data of increasingly bigger increments as follows. First we measures the time to merge one of the small segments into the Person computed with 50% of the data (i.e., the overall data is 55% of the original IRP). We then measured the time to merge two of the small segments into the same Person (i.e., overall 60% of the original data). We repeated this with the rest of the smaller increments until our last run in which the increment of data is 50% of IRP. We measured the same set of times as before, except that we further separated the chase and merge times for the incremental approach. The results are shown in Figure 5.4, with the x-axis indicating the overall amount of IRP integrated at each incremental step.

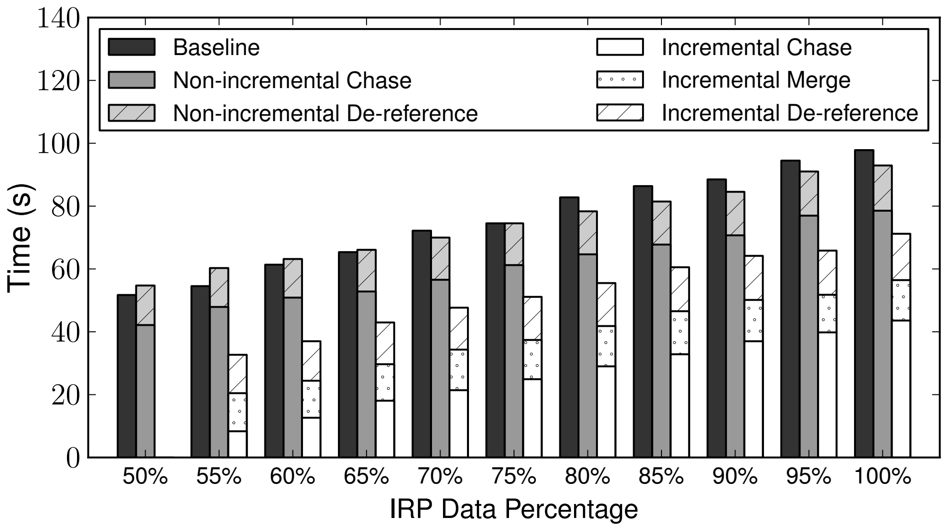


Figure 5.4: Integration Times in the Basic SEC Integration Stage, with Incremental Input Data

Our incremental chase and de-reference approach significantly reduces the overall integration time (**32**% on average). Furthermore, the baseline time and all chase times increase steadily as the integrated data size increases, while the de-reference times and the merge time remains virtually constant. This indicates that the de-reference and merge processes are relatively data-insensitive. For the incremental case, at first, the overhead of merge and de-reference is relatively large (74% of the total integration time). But as the chase time increases, this overhead becomes less significant (38% of total time at the end).

We also observe that the incremental chase time is proportional to the incremental input size. Notice that the incremental chase time for the last run is virtually identical to the chase time of the first run. Both these chase operations used different halves of the IRP data.

5.9 Conclusions and Future Work

In this chapter, we studied the incremental information integration problem towards user-friendly information integration. We started by formalizing the declarative incremental integration problem and proposed a chase and de-reference mechanism for a non-incremental declarative integration. We then extended this mechanism to the incremental scenario, by maintaining and reusing previous chase results. We further optimized the de-reference procedure by proposing a *reference summary* graph. We also developed novel algorithms for deriving and maintaining these reference summaries.

Our chase and de-reference approach is not the only possible strategy for implementing incremental information integration flow. In our approach, much of the incremental processing is done in the chase step and de-reference operates on the previous chase results and the new ones. We are currently working on an alternative algorithm that reuses part of the existing nested target structure directly during the de-reference phase. Through out this chapter we have assumed that the underlying file system has no update semantics (e.g., HDFS). To create the target entities (e.g., Person), de-reference currently materializes the entire target entity. There are file systems that do support updates and, furthermore, support indexing of data (e.g., HBASE). A goal of this project is to compile HIL programs into such systems without the need to change the program logic. In such file systems, the chase phase will update indexes in-place and the de-reference strategy could decide to update only the affected parts of the target entities.

CHAPTER VI

Related Work

6.1 Database Usability

The general idea of making databases systems more usable was introduced by Jagadish et al. [54]. This keynote paper studied the reasons for poor accessibility of state-of-the-art database systems and proposed a presentation data model based on direct manipulation with a schema later approach.

6.2 Personal Information Management

Personal information management, or PIM, is intended to support individual users to order their daily lives through the acquisition, organization, maintenance, retrieval and sharing of information [58, 96]. The phrase was first used in the 1980s [62] studying psychological issues involved in the automation of information management. However, the spectrum of PIM spans from cognitive psychology and science, human-computer interaction to data, information and knowledge management. While PIM has drawn increasing attention in the cognitive science and HCI community, to the best of our knowledge, the database field has limited contribution to this general problem.

An important assumption made by PIM is, nowadays personal information is often scattered and isolated in separate applications and devices, with various formats not directly usable. This raises the request to integrate information from various distributed collections with suitable mapping and filtering applied. X. Dong et al. [34] built a prototype PIM system from a data management point of view by creating a mediated logical view of all personal information. However, they focus on the information search and the schema and mappings are fixed. We believe the end-user should have complete freedom to design and evolve both the integrated view and the underlying mappings. As a result, we investigate user-friendly approaches for the user to design the integration, mapping and filtering by themselves.

6.3 End-User Programming

In the HCI community, there has been a great deal of past work in end-user programming, or EUP [57, 100]. Among them, Cypher [30] and Lieberman [66] describe a number of programming-by-demonstration and programming-by-example systems. In general, this dissertation shares the same spirit with EUP as enabling the end-user to implicitly program a data management script with a user-friendly interface. However, EUP focuses on an easy derivation of the scripts which can be repeated used in future applications. On the other sides, this dissertation struggles on the generation of a democratic data management scheme for the end-user. Rather than trying to enable end-users to easily derive a reusable data manipulation script, we are more devoted to allowing the end-users to freely restructure, reposition, integrate and filter their own data, so that they can understand the essence of the data they have and make maximum use out of it.

6.4 Schema Design

Database schema design has been studied extensively [16, 45, 21]. There is a great deal of work on defining a good schema, both from the perspective of capturing real-life requirements (e.g., normalization) and supporting efficient queries. However, such schema design has typically been considered a heavyweight, one-time operation, which is done by a technically sophisticated database expert based on careful requirements analysis and planning. When considering schema design for end-users, the challenge has shifted to enabling non-technical people to give birth to a reasonably defined database schema [54].

6.5 Schema Evolution

Schema evolution is a well studied area. There is a great amount of earlier work on schema evolution in object-oriented databases [13, 64]. Recently, there arises an interest on schema evolution in web information systems [27, 76, 28, 29], with a focus on systemizing schema changes, optimizing query rewriting and automating database migration. The idea of proposing a language of schema modification operators to concisely express complex schema changes inspires the algebra design in CRIUS, where the end-user may leverage a set of simple yet expressive schema evolution operators to accomplish complicated evolution tasks using direct manipulation.

6.6 Nested Relations

Nested relations introduce a hierarchical semantics and suggest a natural presentation to integrate data and schema in a way that is easily perceivable to end-users. The theoretical foundation of CRIUS is built on top of nested relations and their normal forms, which have been studied extensively [87, 80, 75, 26, 86, 82, 87, 97, 38, 88]. Although state-of-the-art database systems have very limited native support for hierarchical data storage, [31] has suggested a possibility for expressing nested structures with flat relations by treating the mapping as a simulation procedure.

6.7 Direct Manipulation

While the database community have been focusing on advanced system capability for a long time, our neighbors in the human computer interaction community have examined how to bridge the gap between user capabilities and computer functionalities. In particular, they have developed direct manipulation, which is considered to be one of the most popular user-friendly paradigms [52, 91].

Direct manipulation techniques are employed thought the systems in this thesis. Each operator in the span table algebra introduced in CRIUS is designed as a direct manipulation using mouse point-and-clicks or drag-and-drops. Mweaver implements a direct manipulation interface in which current mapping generation status is displayed immediately after each user data entry.

6.8 Graph Specification on RDBMS

Polaris [92] features an interface for exploration of multi-dimensional databases, by extending the Pivot Table interface to directly generate a rich and expressive set of graphical displays on top of relational databases. A succinct visual specification is developed for describing such table-based graphical displays and interpreted as a sequence of relational database operations.

This inspires our work of abstracting a hierarchical data presentation and proposing an user-friendly algebra on top of relational databases. However, while their work focuses on displaying statistical data analysis in a flexible way on top of a static database, our work enables schema evolution in an intuitive manner as the underlying database is dynamically modified.

6.9 Schema Matching and Mapping

Research into schema matching and mapping make up an enormous body of work, as described in a recent text [17]. State-of-the-art schema matching approaches can be roughly classified into three categories. *Schema-based* techniques perform matching by examining metadata, such as in Clio [84] and Similarity Flooding [72]. *Instance-based* approaches determine the similarity between schema elements from the similarity of the characteristics of their instances [68, 60]. Many systems utilize a combination of these two techniques, such as LSD [33], Cupid [69], COMA [32]. *Usage-based* methods improve matching quality by exploiting usage information, such as query logs [36] and search clicklogs [78].

6.10 Schema Mapping Design using Examples

Data examples have been an important part of the schema mapping literature. Alexe, et al. recently developed Eirene, a system for interactive design and refinement of schema mappings using data examples, by GLAV fitting generation [10]. Eirene offers abundant flexibility in that it derives the mappings as long as the source and the target schema, as well as a few paired examples under both schemas are provided. However, consequently, the user has to understand both schemas in order to fill in valid data examples and explicitly specify join paths by linking related tables using data with the same value. This may result in some user burden, especially in the presence of a complex source schema and long join paths in the mapping. In contrast, MWeaver assumes the existence of a complete source database instance, to which the user-input samples belong. As a result, the user does not need to know the source schema or to specify the join paths, because the system can use the source instance as a knowledge base to automatically derive the mappings.

6.11 Debugging Schema Mappings using Examples

Yan, et al. attempted to choose tuples that best exemplify a mapping [101]. Alexe, et al. systematically investigated the capabilities and limitations of data examples in explaining and understanding schema mappings, especially in using universal examples to characterize mappings defined by s-t tgds [9]. However, these are done in an “explanatory” phase after the mapping has been generated.

SPIDER [8] and MUSE [7] are designed to refine a partially-correct mapping generated by a more traditional tool. They first generate candidate mappings using a match-driven mapping tool, and then ask the user to debug them by examining user-proposed examples. In contrast, MWeaver asks users to simply enter data items, and to trust that the system will find the correct mapping.

6.12 Automatic Schema Matching

Drumm et al. designed QuickMig for automatic schema matching for data migration [35]. It asks a user to manually create target instances in the source and then apples standard instance-based matching algorithms on these sample instances to determine the matching. However, these sample instances are only used to generate schema element correspondences rather than mapping structure.

6.13 Interactive Information Integration

Recently, there has been a trend toward leveraging user feedback to improve the quality of an information integration task. Talukdar et al. [95, 94] developed system Q to assist the user in creating integration queries. In system Q, integration is defined as a union of queries weighted by relevance. The system shows the query result to the user, who in turn provides feedback to the system by judging whether a result tuple is relevant. In MWeaver, the system notifies the user about the current mapping generation status, and the user provides feedback in the form of additional sample instances.

Recent work [53] proposes a smart copy and paste (SCP) model and architecture for seamlessly combining design-time and run-time aspects of data integration. The desired schema mappings are generalized from user actions of copies and pastes with continuous interactions between the user and the system. The idea of such design-time and run-time combination is reflected in both CRIUS and MWeaver, where the user experiments schema evolution or schema mapping in a step by step manner with instant system feedback. While SCP deals with data integration among multiple datasets, CRIUS is focusing on integrating schema update and data manipulation inside a single database. MWeaver enables schema mapping deduction in a more generalized setting where the explicit source-to-target correspondence implied by the copy-and-paste is not available in a sample-driven context.

6.14 Query by Example

Query-by-Example (QBE) [103] is a well-known work that employs example data to assist in query generation. The user constructs a query with QBE by providing example tuples under both the database schema and the result view. Examples with same value suggest how the relations are joined and which attributes are projected. While the user can supply fake data in QBE, the input to MWeaver must be samples from the database instance. As a result, the user has to manually specify join paths by simulated IDs in QBE, while MWeaver is able to automatically derive the join paths from sample values.

6.15 Database Keyword Search

MWeaver has a strong relationship to database keyword search techniques, which have been extensively studied in the literature [20, 5, 50]. However, database keyword search focuses on querying tuples that may be related to the keywords; in contrast, MWeaver focuses on determining the exact mapping that produces a target database containing the samples.

6.16 HIL: A High-Level Integration Language

The incremental integration framework introduced in this dissertation build-s upon the HIL language [48] for declarative entity integration. While the major difference in this dissertation is in providing incremental evaluation capabilities, another difference is that we establish a stand-alone two-phase semantics for evaluation of integration rules that is based on chase and de-reference. We also give a novel and efficient algorithm for de-referencing of nested data that is based on generating queries with join operations.

6.17 Pointer Navigation

De-referencing is related, to some extent, to the notion of pointer navigation in object-oriented databases, where various papers (e.g., [90]) studied the benefits of replacing pointer-based joins into value-based joins. However, our de-referencing applies to an entire data set (the target), with arbitrarily many levels of nesting and unbounded number of references. As a consequence, our algorithm needs to use an auxiliary data structure to keep track of all the possible reference types that may appear in a record at a given level in the hierarchy.

6.18 Incremental View Maintenance and View Adaptation

Incremental view maintenance [25, 77] deals with methods for efficiently updating materialized views when the source data is updated. View adaptation [44, 74] is a variant of view maintenance that investigates methods of keeping the data in a materialized view up-to-date in response to changes in the view definition itself. First, we note that incremental integration method includes both types of techniques as special cases, since our incremental evaluation s-trategy applies to changes in the source data alone, the integration rules alone, and in both source data and integration rules, which is our main use case and corresponds to adding new data sources into the integration process. At the same time, we note below a few other specific differences with respect to both incremental view maintenance and view adaptation.

Incremental view maintenance for data integration is the focus in [43], which gives self-maintainability conditions for outer-join views, namely conditions under which an outer-join view can be re-materialized by using the view itself and the changes only to the source databases. The paper restricts itself to a single target relation, while the changes to the database are single-tuple inserts, updates or deletes. In contrast, our target schema consists of an arbitrarily complex set of entities with inter-dependencies, and the changes to the source data are given as a new increment of arbitrary size. As for view adaptation, the work in [44] considers how to reevaluate a SQL query in response to incremental changes to the query, such as adding or removing a filter, adding a join with a relation, changing a grouping condition, etc. The view adaptation algorithm makes essential use of the fact that the previous version of the query is known. In contrast, we allow for new integration logic to be added that is completely independent of the previous rules. In turn, this requires the development of new types of incremental techniques.

CHAPTER VII

Conclusions

This dissertation studies various approaches towards enabling end-users to easily structure, integrate, filter and maintain their every-day data. To help the user structure their data, this thesis proposes a Span Table interface and algebra for the user to easily create and modify the data schema in an intuitive way. Since integrity constraints become flexible and vulnerable in such an environment with casual schema design, the thesis also studies how to guide user data-entry to ensure a healthy set of integrity constraints. The thesis folds these ideas into a system called CRIUS, and demonstrates its usability by both performance experiments and user studies.

To facilitate the user to integrate information to their own data repository, this dissertation proposes a sample-driven approach to automatically derive schema mapping from user-input sample target data. This frees the end-users from the burden of understanding complex source schema and the burden of specifying intricate mapping structure placed by state-of-the-art schema mapping tools. This thesis presents a prototype system MWeaver, which implements the idea on top of an efficient mapping weaving algorithm. It is shown via user study that the system is much more user-friendly than the state-of-the-art mapping tools. This thesis also demonstrates that the weaving algorithm is very efficient via performance experiments.

In order to enable the end-users to easily filter their data, this dissertation proposes an approach to automatically derive the selection condition by user-input examples that the user wants to select. Since these examples may not determine the desired condition, this thesis also introduces an expressive representation of the selection condition together with an algebra of direct manipulation operators for the user to refine the initial selection condition derived from examples. The dissertation shows that this approach is able to derive high-quality initial conditions from just a few examples via a simulated user s-tudy, and our algorithm for deriving such initial conditions is very efficient and scalable.

Finally, this dissertation studies the problem of reducing user burden in the face of incremental information integration. When new data comes in existing data sources, or new data with new schema are incorporated, instead of asking the user to re-perform the integration task from scratch, this thesis proposes a new approach which incrementally updates the previous integration result using only the new data and new mapping logic. This thesis also develop-s innovative algorithms to implement such incremental integration approach. According to evaluation on real-world data, this thesis shows significant performance benefits of the incremental integration approach.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Altova mapforce. http://www.altova.com/mapforce.html.

[2] Microsoft biztalk server. http://www.microsoft.com/biztalk/en/us/.

[3] Stylus studio. http://www.stylusstudio.com/.

[4] Serge Abiteboul and Nicole Bidoit. Non First Normal Form Relations: An Algebra Allowing Data Restructuring. *JCSS*, 33(3):361–393, 1986.

[5] S. Agrawal, S. Chaudhuri, and G. Das. DBXplorer: A system for keyword-based search over relational databases. In *ICDE*, page 5, 2002.

[6] AV Aho, C. Beeri, and JD Ullman. The theory of joins in relational databases. *TODS*, 4(3):297–314, 1979.

[7] B. Alexe, L. Chiticariu, R.J. Miller, and W.C. Tan. Muse: Mapping understanding and design by example. In *ICDE*, pages 10–19, 2008.

[8] B. Alexe, L. Chiticariu, and W.C. Tan. SPIDER: a schema mapPIng DEbuggeR. In *VLDB*, pages 1179–1182, 2006.

[9] B. Alexe, P.G. Kolaitis, and W.C. Tan. Characterizing schema mappings via data examples. In *SIGMOD*, pages 261–272, 2010.

[10] B. Alexe, B. ten Cate, P.G. Kolaitis, and W.C. Tan. Designing and refining schema mappings via data examples. In *SIGMOD*, page 133, 2011.

[11] Bogdan Alexe, Wang-Chiew Tan, and Yannis Velegrakis. Stbenchmark: towards a benchmark for mapping systems. *Proc. VLDB Endow.*, 1:230–244, August 2008.

[12] M. Arenas and L. Libkin. A normal form for xml documents. *TODS*, 29(1):195–232, 2004.

[13] Jay Banerjee, Won Kim, Hyoung-Joo Kim, and Henry F. Korth. Semantics and implementation of schema evolution in object-oriented databases. In *Proceedings of the 1987 ACM SIGMOD international conference on Management of data*, SIGMOD '87, pages 311–322, New York, NY, USA, 1987. ACM.

[14] Pablo Barceló. Logical foundations of relational data exchange. *SIGMOD Rec.*, 38:49–58, June 2009.

[15] H. Bast and I. Weber. Type less, find more: fast autocompletion search with a succinct index. In *SIGIR*, pages 364–371, 2006.

[16] C. Batini, M. Lenzerini, and S. B. Navathe. A comparative analysis of methodologies for database schema integration. *ACM Comput. Surv.*, 18(4):323–364, December 1986.

[17] Z. Bellahense, A. Bonifati, and E. Rahm, editors. *Schema Matching and Mapping*. Springer, 2011.

[18] P.A. Bernstein, S. Melnik, and J.E. Churchill. Incremental schema matching. In *VLDB*, pages 1167–1170, 2006.

[19] K Beyer, Vuk Ercegovac, Rainer Gemulla, Andrey Balmin, Mohamed Eltabakh, Carl-Christian Kanne, Fatma Ozcan, and Eugene J Shekita. Jaql: A Scripting Language for Large Scale Semistructured Data Analysis. *PVLDB*, 4(12), 2011.

[20] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, S. Sudarshan, and IIT Bombay. Keyword searching and browsing in databases using BANKS. In *ICDE*, page 431, 2002.

[21] Joachim Biskup. Achievements of relational database schema design theory revisited. In *Semantics in Databases*, pages 29–54, 1998.

[22] Peter Buneman, Sanjeev Khanna, Keishi Tajima, and Wang Chiew Tan. Archiving Scientific Data. In *SIGMOD*, pages 1–12, 2002.

[23] D. Burdick, M. A. Hernández, H. Ho, G. Koutrika, R. Krishnamurthy, L. Popa, I. R. Stanoi, S. Vaithyanathan, and S. Das. Extracting, Linking and Integrating Data from Public Sources: A Financial Case Study. *IEEE Data Eng. Bull.*, 34(3):60–67, 2011.

[24] M.J. Cafarella, A. Halevy, and N. Khoussainova. Data integration for the relational web. *VLDB*, 2(1):1090–1101, 2009.

[25] Stefano Ceri and Jennifer Widom. Deriving Production Rules for Incremental View Maintenance. In *VLDB*, 1991.

[26] Latha S. Colby. A recursive algebra and query optimization for nested relations. In *SIGMOD*, pages 567–582, 1989.

[27] C. Curino, H. Moon, and C. Zaniolo. Managing the history of metadata in support for db archiving and schema evolution. *Advances in Conceptual Modeling–Challenges and Opportunities*, pages 78–88, 2008.

[28] C.A. Curino, L. Tanca, H.J. Moon, and C. Zaniolo. Schema evolution in wikipedia: toward a web information system benchmark. In *In International Conference on Enterprise Information Systems (ICEIS*. Citeseer, 2008.

[29] Carlo A. Curino, Hyun J. Moon, and Carlo Zaniolo. Graceful database schema evolution: the prism workbench. *Proc. VLDB Endow.*, 1(1):761–772, August 2008.

[30] Allen Cypher, Daniel C. Halbert, David Kurlander, Henry Lieberman, David Maulsby, Brad A. Myers, and Alan Turransky, editors. *Watch what I do: programming by demonstration*. MIT Press, Cambridge, MA, USA, 1993.

[31] Jan Van den Bussche. Simulation of the nested relational algebra by the flat relational algebra, with an application to the complexity of evaluating powerset algebra expressions. *Theoretical Computer Science*, 254(1-2):363–377, 2001.

[32] H.H. Do and E. Rahm. COMA: a system for flexible combination of schema matching approaches. In *VLDB*, pages 610–621, 2002.

[33] A.H. Doan, P. Domingos, and A.Y. Halevy. Reconciling schemas of disparate data sources: A machine-learning approach. In *SIGMOD*, pages 509–520, 2001.

[34] Xin Luna Dong and Alon Halevy. A platform for personal information management and integration. In *Proceedings of VLDB 2005 PhD Workshop*, page 26. Citeseer, 2005.

[35] C. Drumm, M. Schmitt, H.H. Do, and E. Rahm. Quickmig: automatic schema matching for data migration projects. In *CIKM*, pages 107–116, 2007.

[36] H. Elmeleegy, M. Ouzzani, and A. Elmagarmid. Usage-based schema matching. In *ICDE*, pages 20–29, 2008.

[37] J. Euzenat and P. Shvaiko. *Ontology matching*. Springer-Verlag New York Inc, 2007.

[38] R. Fagin. Multivalued dependencies and a new normal form for relational databases. *TODS*, 2(3):262–278, 1977.

[39] Ronald Fagin, Phokion G. Kolaitis, Lucian Popa, and Wang Chiew Tan. Composing Schema Mappings: Second-Order Dependencies to the Rescue. *TODS*, 30(4):994–1055, 2005.

[40] PC Fischer, LV Saxton, SJ Thomas, and D. Van Gucht. Interactions between dependencies and nested relational structures. *Journal of Computer and System Sciences*, 31(3):343–354, 1985.

[41] D. Florescu and D. Kossmann. Storing and querying xml data using an rdmbs. *IEEE Data Engineering Bulletin*, 22(3):27–34, 1999.

[42] G. Graefe. Efficient columnar storage in b-trees. *SIGMOD*, 2007.

[43] Ashish Gupta, H. V. Jagadish, and Inderpal Singh Mumick. Data Integration using Self-Maintainable Views. In *EDBT*, 1996.

[44] Ashish Gupta, Inderpal Singh Mumick, Jun Rao, and Kenneth A. Ross. Adapting Materialized Views after Redefinitions: Techniques and a Performance Study. *Inf. Syst.*, 26(5):323–362, 2001.

[45] Terry Halpin. *Conceptual schema and relational database design (2nd ed.)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[46] C.S. Hara and S.B. Davidson. Reasoning about nested functional dependencies. In *POD-S*, pages 91–100, 1999.

[47] Jeffrey Heer, Maneesh Agrawala, and Wesley Willett. Generalized selection via interactive query relaxation. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 959–968. ACM, 2008.

[48] M. Hernández, G. Koutrika, R. Krishnamurthy, L. Popa, and R. Wisnesky. HIL: A High-Level Scripting Language for Entity Integration. In *EDBT*, 2013.

[49] M.A. Hernández, R.J. Miller, and L.M. Haas. Clio: A semi-automatic tool for schema mapping. In *SIGMOD*, page 607, 2001.

[50] V. Hristidis and Y. Papakonstantinou. DISCOVER: Keyword search in relational databases. In *VLDB*, page 681, 2002.

[51] Y. Huhtala, J. Karkkainen, P. Porkka, and H. Toivonen. Tane: An efficient algorithm for discovering functional and approximate dependencies. *The Computer Journal*, 42(2):100, 1999.

[52] Edwin L. Hutchins, James D. Hollan, and Donald A. Norman. Direct manipulation interfaces. *Hum.-Comput. Interact.*, 1(4):311–338, December 1985.

[53] Zachary G. Ives, Craig A. Knoblock, Steven Minton, Marie Jacob, Partha Pratim Talukdar, Rattapoom Tuchinda, JoseLuis Ambite, Maria Muslea, and Cenk Gazen. Interactive data integration through smart copy and paste. In *CIDR*, 2009.

[54] H. V. Jagadish, Adriane Chapman, Aaron Elkiss, Magesh Jayapandian, Yunyao Li, Arnab Nandi, and Cong Yu. Making database systems usable. In *SIGMOD*, pages 13–24, 2007.

[55] M. Jakobsson. Autocompletion in full text transaction entry: a method for humanized input. *SIGCHI*, 17(4):327–332, 1986.

[56] M. Jayapandian, A. Chapman, V.G. Tarcea, C. Yu, A. Elkiss, A. Ianni, B. Liu, A. Nandi, C. Santos, P. Andrews, et al. Michigan molecular interactions (mimi): putting the jigsaw puzzle together. *Nucleic acids research*, 35:D566, 2007.

[57] Capers Jones. End user programming. *Computer*, 28(9):68–70, 1995.

[58] William Jones. Personal information management. *Annual review of information science and technology*, 41(1):453–504, 2007.

[59] V. Kacholia, S. Pandit, S. Chakrabarti, S. Sudarshan, R. Desai, and H. Karambelkar. Bidirectional expansion for keyword search on graph databases. In *VLDB*, pages 505–516, 2005.

[60] J. Kang and J.F. Naughton. On schema matching with opaque column names and data values. In *SIGMOD*, pages 205–216, 2003.

[61] P.G. Kolaitis. Schema mappings, data exchange, and metadata management. In *PODS*, pages 61–75, 2005.

[62] Mark W Lansdale. The psychology of personal information management. *Applied Ergonomics*, 19(1):55–66, 1988.

[63] M. Lenzerini. Data integration: A theoretical perspective. In *PODS*, pages 233–246, 2002.

[64] Barbara Staudt Lerner and A. Nico Habermann. Beyond schema evolution to database reorganization. *SIGPLAN Not.*, 25(10):67–76, September 1990.

[65] W.S. Li and C. Clifton. SEMINT: A tool for identifying attribute correspondences in heterogeneous databases using neural networks. *Data and Knowledge Engineering*, 33(1):49–84, 2000.

[66] Henry Lieberman. *Your wish is my command: Programming by example.* Morgan Kaufmann, 2001.

[67] S. Lopes, P. Jean-Marc, and L. Lakhal. Efficient discovery of functional dependencies and armstrong relations. *EDBT*, pages 350–364, 2000.

[68] J. Madhavan, P.A. Bernstein, A.H. Doan, and A. Halevy. Corpus-based schema matching. In *ICDE*, pages 57–68, 2005.

[69] J. Madhavan, P.A. Bernstein, and E. Rahm. Generic schema matching with cupid. In *VLDB*, pages 49–58, 2001.

[70] D. Maier, A.O. Mendelzon, and Y. Sagiv. Testing implications of data dependencies. *TODS*, 4(4):469, 1979.

[71] Heikki Mannila and Kari-Jouko Räihä. Dependency inference. In *Proceedings of the 13th International Conference on Very Large Data Bases*, pages 155–158. Morgan Kaufmann Publishers Inc., 1987.

[72] S. Melnik, H. Garcia-Molina, and E. Rahm. Similarity flooding: A versatile graph matching algorithm and its application to schema matching. In *ICDE*, pages 117–128, 2002.

[73] R.J. Miller, L.M. Haas, and M.A. Hernéandez. Schema Mapping as Query Discovery. pages 77–88, 2000.

[74] Mukesh K. Mohania and Guozhu Dong. Algorithms for Adapting Materialised Views in Data Warehouses. In *CODAS*, pages 309–316, 1996.

[75] Wai Yin Mok, Yiu-Kai Ng, and David W Embley. A normal form for precisely characterizing redundancy in nested relations. *TODS*, 21(1):77–106, 1996.

[76] Hyun J. Moon, Carlo A. Curino, Alin Deutsch, Chien-Yi Hou, and Carlo Zaniolo. Managing and querying transaction-time databases under schema evolution. *Proc. VLDB Endow.*, 1(1):882–895, August 2008.

[77] Inderpal Singh Mumick, Dallan Quass, and Barinderpal Singh Mumick. Maintenance of Data Cubes and Summary Tables in a Warehouse. In *SIGMOD*, pages 100–111, 1997.

[78] A. Nandi and P.A. Bernstein. HAMSTER: using search clicklogs for schema and taxonomy matching. *VLDB*, 2(1):181–192, 2009.

[79] N. Novelli and R. Cicchetti. Functional and embedded dependency inference: a data mining point of view. *Information Systems*, 26(7):477–506, 2001.

[80] Z Meral Ozsoyoglu and Li-Yan Yuan. A new normal form for nested relations. *TODS*, 12(1):111–136, 1987.

[81] Efstratios Papadomanolakis and Anastassia Ailamaki. Autopart: Automating schema design for large scientific databases using data partitioning. In *SSDBM*, pages 383–392, 2004.

[82] J. Paredaens and D. Van Gucht. Converting nested algebra expressions into flat algebra expressions. *TODS*, 17(1):65–93, 1992.

[83] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[84] L. Popa, Y. Velegrakis, M.A. Hernández, R.J. Miller, and R. Fagin. Translating web data. In *VLDB*, pages 598–609, 2002.

[85] E. Rahm and P.A. Bernstein. A survey of approaches to automatic schema matching. *VLDB*, 10(4):334–350, 2001.

[86] Balaji Rathakrishnan and Junguk L. Kim. An extended recursive algebra for nested relations and itsoptimization. In *COMPSAC*, page 145, 1993.

[87] Mark A. Roth, Herry F. Korth, and Abraham Silberschatz. Extended algebra and calculus for nested relational databases. *TODS*, 13(4):389–417, 1988.

[88] HJ Schek and MH Scholl. The relational model with relation-valued attributes. *Information Systems*, 11(2):137–147, 1986.

[89] Jayavel Shanmugasundaram, Kristin Tufte, Chun Zhang, Gang He, David J. DeWitt, and Jeffrey F. Naughton. Relational databases for querying xml documents: Limitations and opportunities. In *VLDB*, page 314, 1999.

[90] Eugene J. Shekita and Michael J. Carey. A Performance Evaluation of Pointer-Based Joins. In *SIGMOD*, pages 300–311, 1990.

[91] Ben Shneiderman. Direct manipulation: A step beyond programming languages (abstract only). In *Proceedings of the joint conference on Easier and more productive use of computer systems. (Part - II): Human interface and the user interface - Volume 1981*, CHI ’81, pages 143–, New York, NY, USA, 1981. ACM.

[92] Chris Stolte, Diane Tang, and Pat Hanrahan. Polaris: A system for query,analysis and visualization of multi-dimensional relational databases. *IEEE Trans. Vis. Comput. Graphics*, 8(1):52–65, 2002.

[93] M. Stonebraker, D.J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O’Neil, et al. C-store: a column-oriented dbms. In *VLDB*, pages 553–564, 2005.

[94] P.P. Talukdar, Z.G. Ives, and F. Pereira. Automatically incorporating new sources in keyword search-based data integration. In *SIGMOD*, pages 387–398, 2010.

[95] P.P. Talukdar, M. Jacob, M.S. Mehmood, K. Crammer, Z.G. Ives, F. Pereira, and S. Guha. Learning to create data-integrating queries. *VLDB*, 1(1):785–796, 2008.

[96] Jaime Teevan, William Jones, and Benjamin B Bederson. Personal information management. *Communications of the ACM*, 49(1):40–43, 2006.

[97] D. Van Gucht and P.C. Fischer. Multilevel nested relational structures. *Journal of Computer and System Sciences*, 36(1):77–105, 1988.

[98] M.W. Vincent, J. Liu, and C. Liu. Strong functional dependencies and their application to normal forms in xml. *TODS*, 29(3):445, 2004.

[99] Steven A Wolfman, Tessa Lau, Pedro Domingos, and Daniel S Weld. Mixed initiative interfaces for learning tasks: Smartedit talks back. In *Proceedings of the 6th international conference on Intelligent user interfaces*, pages 167–174. ACM, 2001.

[100] Jeffrey Wong and Jason I. Hong. Making mashups with marmite: towards end-user programming for the web. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI ’07, pages 1435–1444, New York, NY, USA, 2007. ACM.

[101] L.L. Yan, R.J. Miller, L.M. Haas, and R. Fagin. Data-driven understanding and refinement of schema mappings. In *SIGMOD*, page 485, 2001.

[102] C. Yu and HV Jagadish. Efficient discovery of xml data redundancies. In *VLDB*, pages 103–114, 2006.

[103] M.M. Zloof. Query by example. In *Proceedings of the May 19-22, 1975, national computer conference and exposition*, pages 431–438, 1975.