Managing Updates and Transformations in Data Sharing Systems

by

Arni Mar Thrastarson

B.Sc. Computer Science, The University of Iceland, 2012
B.Sc. Software Engineering, The University of Iceland, 2012

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in

The Faculty of Graduate and Postdoctoral Studies

(Computer Science)

THE UNIVERSITY OF BRITISH COLUMBIA

(Vancouver)

October 2014

© Arni Mar Thrastarson 2014

Abstract

Dealing with dirty data is an expensive and time consuming task. Estimates suggest that up to 80% of the total cost of large data projects is spent on data cleaning alone. This work is often done manually by domain experts in data applications, working with data copies and limited database access. We propose a new system of update propagation to manage data cleaning transformations in such data sharing scenarios. By spreading the changes made by one user to all users working with the same data, we hope to reduce repeated manual labour and improve overall data quality. We describe a modular system design, drawing from different research areas of data management, and highlight system requirements and challenges for implementation. Our goal is not to achieve full synchronization, but to propagate updates that individual users consider valuable to their operation.

Preface

This dissertation is original, unpublished, independent work by the author, Arni Mar Thrastarson.

Table of Contents

Abstract ii

Preface iii

Table of Contents iv

List of Tables vii

List of Figures viii

Acknowledgements ix

1 Introduction 1

2 Case Studies 6

2.1 Scenario 1: Changes in Application Data 7

2.2 Scenario 2: Changes in Source Data 9

2.3 Scenario 3: Data Integration with Limited Storage 12

2.4 Scenario 4: Unknown Transformations 14

3 System Overview 16

3.1 General Overview 16

3.2 System Components 17

3.2.1 Data Sources 17

3.2.2 Data Update Store 19
3.2.3 User Data Definitions 19
3.2.4 Data Files 20
3.2.5 Data Integration 20
3.2.6 Update Consolidation 22
3.2.7 User Mappings 22
3.2.8 Standard ETL Transformations 23
3.2.9 Data Cleaning Changes 23
3.2.10 Transformation Phase 24
3.2.11 User Application 25
3.2.12 Transformation Scripts 25
3.2.13 Transformation Inference Engine 26
3.2.14 Update Propagation to Sources 27

4 Analyzing Case Studies 31
4.1 Scenario 1 31
4.1.1 Retrieving and Transforming the Data 31
4.1.2 Propagating Updates to the Source 37
4.2 Scenario 2 39
4.3 Scenario 3 45
4.4 Scenario 4 48

5 Related Work 49
5.1 Collaborative Data Sharing Systems 49
5.2 Data Transformation Systems 51
5.3 Data Provenance 53
5.4 Data Exchange 55

5.5 Probabilistic Databases 58
5.6 Data Cleaning 59
5.7 Extraction, Transformation, and Loading 60
5.8 View Maintenance 62

6 Conclusions and Future Work 64

Bibliography 70

List of Tables

4.1 Sample data source. 32
4.2 Sample unprocessed application data. 33
4.3 Sample processed application data. 35
4.4 Sample data source after making updates. 40
4.5 Sample update buffer. 42
4.6 Updates presented to application user. 43

List of Figures

3.1 The general system architecture. 18

4.1 Scenario 1: Retrieving and transforming data from a source. . 34

4.2 Scenario 1: Propagating updates to the source. 38

4.3 Scenario 2: Propagating new data and changes from the data source. 41

4.4 The data integration scenario. 47

Acknowledgements


```
Let no man glory in the greatness of his mind,
but rather keep watch o'er his wits.
Cautious and silent let him enter a dwelling;
to the heedful comes seldom harm,
for none can find a more faithful friend
than the wealth of mother wit.

He knows alone who has wandered wide,
and far has fared on the way,
what manner of mind a man doth own
who is wise of head and heart.

Wise in measure let each man be;
but let him not wax too wise;
for never the happiest of men is he
who knows much of many things.

(Snorri Sturluson - 13th century Icelandic poet)
```

I am continually grateful for the kind guidance provided by my excellent supervisor, Dr. Rachel Pottinger. I thank her for making all the necessary pushes and shoves. My gratitude extends to Dr. Ed Knorr for taking the time to read this document. I continue to be amazed by the loving and unwavering support of my parents, both of whom still have little idea of what I actually do. A last shoutout goes out to my good friends Hafsteinn, Gisli Baldur and Siggi Logi, for their encouragement and fun times throughout this process.

Chapter 1

Introduction

As reliance on data increases in modern businesses, the need for appropriate and efficient data management solutions keeps growing. The ability to collect data at large scales, and a lowering cost of storage, are coupled with a growing awareness of the need to incorporate data as an essential part in management decisions.

"Garbage in, garbage out," is a well known mantra about computer systems that applies in equal measure to data dependent decision support systems, whose relevance will in large part be dictated by the quality of the input data. When our business decisions rely on data in such a multidimensional way–past, present, and future–distorted analysis of dirty data can even eliminate the potential benefits of a data-driven approach [30].

Errors in data should be considered an inherent part of data management systems, and they can be introduced at various different phases of data collection efforts. Measurement errors due to miscalibrated sensors in scientific settings are a common source of errors, as well as human input errors when data has to be manually entered. Other common sources include distillation errors, where an analyst transforms or aggregates data in some incorrect way, and data integration errors when data from more than one source is merged, causing inconsistencies in the final outcome. Hellerstein [30] has described the "lifetime" of data as an iterative process, potentially

lasting over large spans of time and space, that includes such tasks as collecting, transforming, storing, cleaning, and analyzing data. The complexity of scenarios where data quality can be compromised, often involving multiple organizations and human actors, foreshadows a difficult problem in practice.

Advances in data integration alone are deserving of a special mention in this regard. Many applications are now reliant on their ability to query data that resides in different heterogeneous sources. In addition, common scenarios in industry also require data from different sources to be merged and consolidated in company databases. Expecting data collection and storage to adhere to a designed initial schema, with little evolution over time is unrealistic in modern enterprises, and this is liable to cause thorny data quality problems in applications that rely on derived or integrated versions of data. Often, the end result is the introduction of more noise to data than in traditional single source systems [28].

We have seen how dirty data is a prevalent issue in most data management applications. Dealing with these concerns, by first discovering and then correcting various quality issues, is an incredibly costly measure. Estimates suggest that up to 80% of the total development time and cost of large data projects is spent solely on data cleaning [21]. That is a huge proportion and one that can't be ignored, especially considering the growing dependence on data solutions across the spectrum. Data cleaning is now the biggest bottleneck in the data analysis pipeline, and due to the cost sensitivity of many projects, it is obvious that even slight improvements on state-of-the-art methods can be of significant financial benefit. Because of this, research interest on data cleaning has proliferated in recent years, followed by an increased offering of dedicated enterprise software.

Although methods for automatic anomaly detection exist, and are con-

tinually being improved, human judgment is still required to judge the result. The manual labour currently needed for data cleaning is therefore quite extensive. Not only does the effort require numerous man-hours, but in practice the people who are best suited to make judgment calls about data quality, are in many cases either experts in the domain of the data, or highly trained analysts. The time and skill level needed to clean data manually will quickly incur large costs on a data project.

A motiviating assumption we make, is that some of the most valuable data cleaning is performed relatively late in the data analysis pipeline. This is when a domain expert starts working with data, already collected and stored in a data source, and must first examine it to correct errors. Note that in this case, the expert is not the same as the database administrator, who might not be as familiar with the domain, and whose main task is to maintain the data rather than use it. The users that actually do so, are likely to only work with particular subsets of the data at a time. Moreover, in a common scenario, this subset is a data copy that the analyst receives, and manipulates locally on his machine. In that case, all data cleaning efforts would be lost to the data source if the local data is simply discarded or ignored after use.

Given the cost and value of data cleaning operations, there is a large motivation to lessen the need for manual inspection of large data sets. To meet some of these challenges, the business intelligence community has recently called for new systems, with a specific control loop that supports write-back to data sources. Such a system could then be used to both correct and annotate the data at its source [48]. An important step in this direction would be an effort to limit wasted data cleaning efforts in data sharing systems where more than one user works on subsets or copies of the same data. This

is the ultimate goal of the system we describe in this thesis. By constructing a framework to facilitate the sharing of data cleaning efforts between different users, we foresee a system of reusable updates, that reduces repeated manual labour and promotes up-to-date data of higher overall quality.

A flurry of new ideas has been introduced in data management research in recent years. Since data management is a big business these ideas are often implemented in prototypes, followed by applicable enterprise solutions. The operation of a typical relational database management system has already been extensively studied, so the focus in data management research has recently shifted to larger systems of multiple data sources, issues with data integration, systems for specialized applications, and data analytics. A common theme has also been effort to combine important ideas into composite systems that serve a broader purpose. An example would be the Trio system [56] that integrates probabilistic databases and provenance ideas. We think this trend will continue and eventually converge in a new general purpose data management system. Exploring new systems that integrate different ideas and try to solve specific problems, is an important step in gauging their usefulness and feasibility as parts of a larger general purpose system.

Motivated by the growth in data-driven business solutions, the importance of data quality for their effectiveness, and the large cost and time-consuming work of ensuring that quality, we propose a new system of update propagation for common data sharing scenarios. In doing so, our goal is to utilize recent advances of important new research topics, in the spirit of the current drive to integrate them into new practical solutions.

In this thesis, we describe a new framework for update propagation in collaborative data sharing scenarios. These are scenarios where analysts

and applications use data from any number of sources, either working with a locally stored copy, or by fetching data directly at run-time, in a typical data integration fashion. Updates are taken to be all changes made to the data by any user, and by their propagation the intention is to notify all relevant users of their existence, spreading the news, so they can conveniently choose which changes to apply to their own data copies. The goal is not total synchronization between users, but a heightened awareness of system-wide actions, to reduce manual labour and qualitatively improve overall data quality.

The key contributions of this thesis are the modular system design, and the identification of necessary system requirements and functions. We present a distributed system architecture, composed of numerous modules that draw from different topics in data management. Among the functions performed by different modules are the storage of all schema- and data mappings that describe how different users' data is related, the logging of various actions and transformations, as well as the identification and propagation of relevant updates from one user to another. Together, the modules ensure that both application users and data sources can seamlessly use shared data, while progressively supporting individual data cleaning transformations and update propagations to improve overall quality in the system.

The remainder of the thesis is organized as follows. In Chapter 2, we discuss four motivating case studies, each highlighting specific system requirements. Chapter 3 gives an overview of the system architecture, before describing each module separately. Chapter 4 shows how our system operates in practice when faced with each of our previous use cases. An overview of related work is found in Chapter 5, and we conclude this thesis in Chapter 6, with a discussion on future work and important challenges.

Chapter 2

Case Studies

The ultimate goal of our system, is for it to be a framework for update propagation in applications with shared data. These assume any number of original data sources and an arbitrary number of users, whose applications take data copies from at least one of the sources as input. The system we seek to describe must be robust enough to solve the known problems accurately, yet general enough to cover a broad field of common use cases.

In order to discover the system requirements, and to motivate various design choices, we investigate four key scenarios. The scenarios vary in their specific goals and assumptions, but the main motivation of the system is to ensure update propagation, while minimizing manual repetition and human errors in data manipulation. An adequate solution is one where all relevant system users are made aware of global data corresponding to their own, in some timely manner, and are given a choice to either accept or deny said changes. That is not to say that we want a system of full synchronization among users. By propagating the transformations made by other users of the same data, we want the decision to lie with each user to accept only updates they deem to be relevant and valuable to their data copy.

2.1 Scenario 1: Changes in Application Data

A common use case is where a user checks out data from a database to use in his own application. It is a fair assumption in many instances that a third party application owner does not have write privileges to the data sources in question. Furthermore, he should not be required to even have the technical know-how to query a database correctly. Instead we will assume that the application owner will request a particular set of data, through some interface, or be directly provided one by an administrator of the data source.

An example of this scenario is the one of a business analyst who is tasked with writing a report on a company's recent performance. We can assume that our analyst is an expert in the domain of the company's business, and by extension the domain of its data. She is not necessarily a database administrator, and is not tasked with the primary storage of the company's data. To perform her analysis our user needs some specific data, for example sales figures for the last quarter, and in this case is given a spreadsheet containing a copy of that data. The particular method of data exchange could be by means of any standard file format for structured data. The important thing is that the user receives her own copy of the requested data for local storage.

As with all data, it is safe to assume that it may contain errors. The first task of our analyst is therefore to carefully examine the data for any inconsistencies that need fixing. We contend that few are better suited for this arduous and time consuming task than the domain experts, tasked with extracting value from the data. This process of data cleaning is interleaved with one of data wrangling, that is systematically changing the structure of

data, to fit some purpose or input format for a later application.

We believe that any number of changes made by a user in this data transformation stage could be of great value to the administrator of the original data source, as well as future users of the data. A system that facilitates the exchange between interested parties, of changes made to data could potentially do a lot of good. The act of reporting news about updates from other parts of the system back to the source would save future users from having to spend time on repeating the work.

The first fundamental task of our system is to establish and maintain a history of transformations made by the user. The most intuitive way to do this would be to maintain a log of all actions made, in a transformation script. The design choice for how such a log should be transcribed is important. A log of tuple-level updates, similar to a database log, is not suitable since it does not capture the transformations at a high enough level. Since the user is expected to be working in a spreadsheet-like environment, it would also be less intuitive, not to mention verbose, for the transaction log to be at the transaction level of a relational database.

Prior research on data transformation systems [38] has defined a declarative transformation language, complete with eight classes of transforms, that is specifically designed to deal with common data cleaning tasks and data wrangling operations. This set of operations is provably sufficient to cover all one-to-one and one-to-many transformations that structured data could possibly undergo [44, 52]. A log generated in this language would solve the the task of keeping track of user transformation history.

A major hurdle on the way to effective transformation logging, is to force the user to stick to using only the meaningful transformations defined by our transformation language. As it is, most common spreadsheet software

allows users to perform ad-hoc changes to data that might be difficult to interpret as transformations on rows and columns. This problem will be further discussed below.

Since both source and application owners are allowed to arbitrarily make changes not only to the data, but to schemas as well, our system must be capable of mapping data between the application and source schemas before propagating it back. This requires a schema mapping module, common in previous work on data exchange and materialized views. Such a module would have to be incrementally updated whenever either side makes changes to its schema.

2.2 Scenario 2: Changes in Source Data

Aspiring to improve data quality at the sources by propagating application changes back to them, is a worthy objective, but on its own probably not worth the extra effort from the application user's perspective. His data is now clean and transformed to his liking so the added overhead of logging actions and sending back updates could be perceived as excessive work. To add to his benefit we introduce the source-centric scenario.

Data is not only cleaned and changed at the application end of the system. Both operations will also be performed every now and then by the administrator of the original data source. This would result in value updates and the occasional schema update. Insertions of new data, or deletions of old data, would be even more common at the source, especially if the source is a transactional database. Whatever changes are made to the source data, they are likely to affect in some way any application working with an older copy of the now changed data.

In this scenario we assume that a user is already working with his own copy of the source data before the original data is updated. To illustrate, we can imagine an academic working on a model that takes as input development indicators about the state of affairs in African countries. The indicators are maintained by the World bank and made available as raw data files on the institution's public website. As new data becomes available, on a semi-regular basis, the indicators are updated and new files are published. We must now solve the problem of how our system will propagate these changes to the academic, essentially offering him to refresh his current copy.

To meet the challenges of this use case our system must be capable of:

- Recording changes made at the source.
- Remembering what query that originally generated the user's data.
- Knowing when the user checked out his copy, or when he last received updates from the source.
- Mapping data between the source and user schemas.
- Updating schema mappings in a timely fashion if either source or user schemas change.
- Applying applicable mappings the user has defined on new data.

In many ways we can model the relationship of the original source and the user's data copy after the relationship between a database schema and a materialized view defined on the schema, the difference being that we do not require the user's data to be a queryable database table. The task of propagating updates from the source to the user is now akin to the problem

of view maintenance in data warehouses [51]. A key difference between our scenario and materialized views is the fact that we do not require the user's data to be fully synchronized with the source after a data refresh.

The main reason view maintenance algorithms come into play here is because a user's data copy from a source is essentially the result set of some query the user evaluates at a given time. If the query is complex, for example involving aggregation or particular kinds of projections, this requires more sophisticated techniques for synchronizing the materialized view [26]. For example, a simple projection can produce duplicate rows that have to be accounted for when making insertions and deletions in the view. To support our desired functionality for this scenario, our system will likely maintain a store of metadata on the user side to keep track of row-level statistics. It is possible that data provenance techniques could be enhanced to contribute to this module. Further research is needed on whether we must constrict users to some subset of queries for our system to work as proposed.

Like before, we assume that a user would like to perform data cleaning operations and transformations on the new data coming from the source. This is mostly handled under Scenario 1 in the above discussion. In addition, we would like our system to remember previous mappings that the user has made and deemed feasible, so that they can be applied on the fly to newly inserted data through an update propagation from the source. This would potentially save the user the time otherwise spent manually re-applying past changes. Not every transformation is suitable as a stored mapping. For example, errors that are corrected during data cleaning are often specific to a single value, and do not generally apply to other data rows. Our system will likely require user input on what mappings should be applied to all incoming data.

2.3 Scenario 3: Data Integration with Limited Storage

Many data applications are designed not to store data locally, but rather to retrieve it over a network only when needed. In one classical data integration situation, data is stored remotely in any number of sources, and queries made over a universal schema return data to the user [28, 29]. Another situation that is becoming popular is when web applications make their data available to other developers through an application programming interface.

The assumption in this scenario is that after the data has been acquired from its remote origin it gets fed directly to an application, and is never stored on the user side. This can be for a lack of local storage but in most cases the reason is the user's need for the data to be as fresh as possible. By never storing data locally the user is also free of all view maintenance concerns that were discussed in Scenario 2 above.

To motivate this scenario, we begin with an example where a group of researchers has been looking at data about UBC's campus. Among other things, they have looked at an application by a local company (Cloverpoint) that is an interactive visualization of the UBC campus. This application is enhanced with data from many different sources, one of which is the open data portal of the city of Vancouver. One of multiple public datasets made available by the city authorities is called "city trees", with information about every single tree within the city limits, including its species, age and geographical location.[1] When loaded into the UBC campus, visualization errors became evident in the data. For example, duplicates (trees on top of trees) were discovered in some locations, due to rough estimations when the

[1] See `data.vancouver.ca/datacatalogue/streetTrees.htm`.

data was recorded. The key problem here is that since the data is not kept in local storage, even after errors have been identified, there is no easy way to fix them.

To meet the challenges this scenario has outlined, our system has to:

- Define and store transformations locally for individual rows of data.
- Apply stored transformations to the correct data at runtime.
- React correctly to new updates made at the source with respect to application transformations.
- Propagate suggested transformations back to the source.

We can assume that the application user's objectives here, namely storing transformations and adding them to new data where they apply, are identical to the ones outlined in Scenario 1 above. The only difference is the lack of local storage. The user is still interested in cleaning and transforming his data, but since he simply can't manipulate it, we need to adjust our system architecture to ensure persistent user transformations. Like before, we are interested in capturing transformation logs when our user manipulates the data. The act of logging should now result in an executable script that gets applied to data every time it is loaded into our application.

A not so subtle difference between the saved transformations here and the case where local storage is assumed, is that we are now interested in applying all changes the user has made, and not just some general transformations applicable to new data. Because of that, the added complexity of including and mapping all changes in data values to the correct rows must be carefully considered in the system design. However, this does not eliminate the need

to keep track of general transformation schemes so they can be applied to new incoming data.

Besides a different setup and the application user's new interface with his data, the goal of our system ultimately remains the same. We want it to propagate updates between collaborating agents (the user application and its data source) while at the same time maintaining all changes between application sessions. We still expect our system to notify the data source of changes made by the user and vice versa. The operation of those components should be fairly similar to how they are described above, with the possible addition of a layer to check if original data anomalies still persist. Such a layer could help prune the transformation script before data is loaded into the application.

2.4 Scenario 4: Unknown Transformations

A crucial component of our proposed system is the transformation log of all changes made by the user. Without knowing exactly what those changes are, any intentions of propagating updates back to the source are moot. The task of forcing the user to make meaningful transformations and capture them for later use in a transformation script has been closely studied [38, 39, 52]. For such solutions to work new tools must be developed, and people who work with that data must use those tools. The industry is already pushing in the right direction[2], but until new solutions gain widespread traction we must still assume that most data wrangling will be done using common spreadsheet software or ad hoc scripts.

Restricting our work to the wonderful world where everybody uses the

[2]See `www.trifacta.com/product/technology/` for one example.

best tools to make life easier for our system would be great, but since that world is not here yet we are forced to eventually address the situation on the ground. Perfect transformation logs are unlikely to be captured in practice. For the sake of generality we must therefore consider our options for the scenario where all information on user changes is absent. In this case all we have to work with are copies of the data before and after possible changes were made.

The question at hand is this: Given relational data from a source database and a modified cleaned copy of the same data, is it possible to automatically infer the script of transformations that modified the source copy with no additional information available? The answer is *we don't know*. To present a definitive solution to this problem would require substantial extra work that is out of the scope of this study. It is, however, a problem worth mentioning. It serves as an important direction for future research and should our system be implemented, its solution would be greatly beneficial.

There are at least two possible ways to tackle this problem: the smart way and the dirty way. The former has a focus on data mining, and the latter requires more draconian comparisons of the data sets. Recently there has been a lot of interest in employing data mining techniques to aid data cleaning systems [27, 35, 38, 39, 52]. Some techniques proposed include various similarity metrics or clustering methods. We have reason to believe that these can be reverse engineered or used to focus the efforts of automatic change detection. Another approach would be to simply compute the difference between the two data instances and try to infer a transformation script that way. Parts of this method would likely be somewhat similar to the *Diff* operator from the Model Management literature [7].

Chapter 3

System Overview

Guided by the case studies presented in the previous chapter, we aspire to lay out in broad strokes the key components of a general update propagation system. We start with a discussion, in Section 3.1, on the general overview of the system and highlight its two types of users. We continue in Section 3.2, and begin describing the various system modules, their different functions, and the challenges that might arise in implementation.

3.1 General Overview

We propose a system to manage updates and transformations in data sharing scenarios, the outline of which can be seen in Figure 3.1. A more detailed discussion on each module and its purpose follows below; but for now, let's direct our focus to the big picture operation of the system.

We assume that the operation of the system is split between two distinct entities: data sources and data applications. We further assume that an instance of our system contains any number of data sources and any number of applications—one of each in the simplest case. The examples we will consider are all limited to a single application, but increasing the kinds of applications should follow trivially from our discussion. We expect the input data to the applications to come either from conventional databases

or a data integration system, as discussed in Scenarios 1 and 3 respectively in Chapter 2. In the same vein, a data application is taken to be anything that takes structured data as input. This would also cover a simple data analysis, where the application could consist of various statistical functions and visualizations. In Figure 3.1, the split between the two interacting system entities is marked by a dotted vertical line.

We expect two types of primary users to interact with our system: a database administrator for each data source and an application user. The difference is important because we assume a different background and skill set for each type. The database administrator is assumed to be an expert in data management and storage, but not necessarily familiar with the semantics of the data. The application user however, is assumed to be a domain expert for the particular data his application is using. We argue that this expertise makes the application user better suited to judge many data anomalies that are expected to exist in the data. The person we refer to as the application user is an individual who is responsible for managing all data after it is received from the data sources and before it enters the application. For our purposes, we won't concern ourselves with any secondary consumers of the data application.

3.2 System Components

3.2.1 Data Sources

Any number of data sources should be able to tie to the system and serve as the main repository of its data. Such a repository should store data in a structured relational format for two main reasons. We need the data to be in a rigid format so that transformations down the line are meaningful

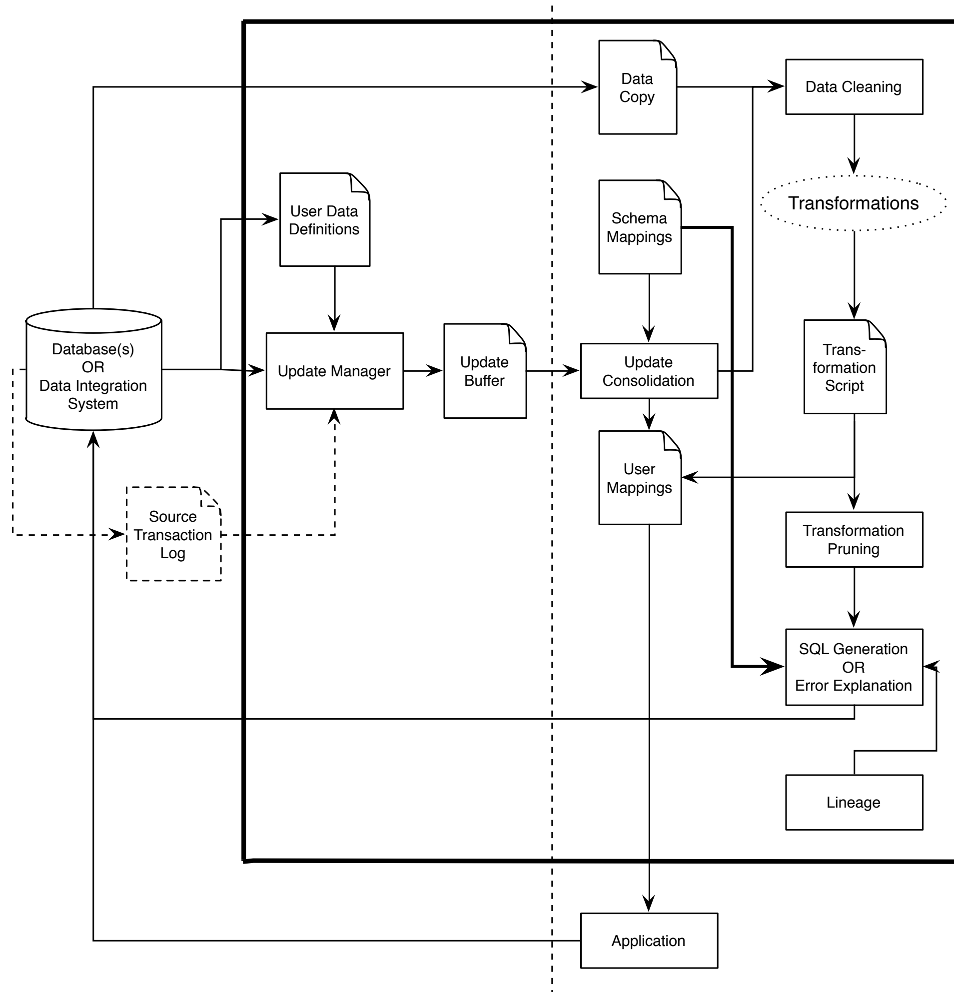

Figure 3.1: The general architecture design of the proposed system. Modules to the left of the dotted vertical line represent modules that interact with the input data sources, while modules to the right of the line interact with the data application.

and well behaved. We also require the data sources to be queryable, since defining and keeping track of data copies will be far easier that way. Most of our use cases expect data sources to be traditional transactional databases, requiring our system to expect and handle regular insertions and deletions of data rows.

3.2.2 Data Update Store

Since update propagation between sources and applications is not expected to be continuous, we must include a module to act as a buffer for updates, made at the source, that have not yet been applied to a particular user's data copy. We call this component the Data Update Store and maintain it at every data source, one section for each checked out data copy. This component should be similar to the transformation log on the user side, but smart enough to only store transformations that affect the specific data that resides with the user. If the user does not store his data locally, the data update store might not be necessary. In that case, we would still argue for its importance, because notifying a user of changes made at the source so he can evaluate them for his purpose would likely be desired.

3.2.3 User Data Definitions

Given a relational database D as a data source, a user data definition is a query over D that defines the specific data copy checked out by a user application. A user data definition is similar to the query definitions of logical views, and at the time of checkout the result of it, sometimes called a report, defines the exact instance of data the corresponding user application receives. In order to know what data updates apply each time, our system

must maintain a table of definitions at every data source for any user applications that it serves. Data files produced by the source for distribution should also be tied to their general query definition.

3.2.4 Data Files

It is unlikely that data applications will in all scenarios have direct access to their working data at the sources. In a similar vein, users can't be expected to always have the capabilities or the schema understanding to query a database directly for what they want. In those cases, and most cases where certain data is being made publicly available, a database administrator will compile a data file in a standard format. The file could be a spreadsheet, csv file, or any popular format known to ease data exchange between parties. Our system design must gracefully handle the intermediate step of data files in its use cases.

3.2.5 Data Integration

Querying multiple heterogeneous databases for data is an increasingly common task, even in otherwise simple applications. If the user is only presented with the query result, and the data still resides in its original places this is called data integration. This problem has been heavily researched in the last decades [28, 46] and remains of some interest. Since we want to allow data input to our system to come from an integration of such databases, we must consider the standard interfaces available to external systems in those scenarios.

A common approach is to describe data sources as view expressions under a mediated universal schema. The user queries the universal view as a

single logical data source, and the rest is taken care of. For this to work, schema mappings, sometimes called source descriptors, must be generated and maintained by some means. The design of our system should support a common module for data integration.

Schema Mappings

The relationship between data sources and their universal view in data integration systems is described with a mechanism called schema mappings [47]. They are a collection of formulas in some logic, usually a convenient subset of first-order logic, that express all constraints between a source schema and a related target schema. If a user application needs to change the structure of its data copy, such schema level differences must be recorded with schema mappings and stored on the user side.

In the case where a user application stores the full data copy locally, schema mappings can be used to map directly between a source schema, possibly a logical schema representing many sources, and the application schema. For this purpose, two distinct classes of mappings are common: the so-called tuple-generating dependencies on one hand, and equality dependencies on the other [24].

Data Exchange

Data exchange is one of the oldest fundamental problems in database research. Given an instance of source data and a schema mapping between source and target schemas, data exchange is the problem of generating a target instance that adheres to all constraints.

When data copies in our system are stored locally by applications, all new

data and updates coming in from data sources must solve the data exchange problem in some consistent manner before they can be considered by users. Unless we can agree on what constitutes a best solution the solution space in data exchange is not deterministic. Therefore, a data exchange module in our system must be careful to generate predictable solutions. The fact that the target schema has in our case directly evolved from the source schema could possibly make this problem better behaved than the most general case.

3.2.6 Update Consolidation

Every data application in our system should have an update consolidation module. This part of the system acts as a funnel for incoming data updates, and its inputs come from various sources after schema mappings have been applied. The update consolidation phase transforms them into the application schema, detects conflicting updates from different sources and finally presents a set of possible updates in human-readable form, so that the application user can choose which updates to apply to his data. If possible, updates should not be presented at the row level but at some appropriate aggregation that groups together related updates if they apply to many different rows.

3.2.7 User Mappings

The data integration scenario where a user application fetches all data at runtime and never stores the bulk data locally causes an interesting conundrum. We want and expect the user to be able to make changes to and clean his data as soon as issues are discovered. The problem is that without local storage, changes need to be applied anew everytime the data is loaded. We

propose that our system solve this issue by storing so-called user mappings in a local update table. Now when data is loaded, stored transformations from the local update table are applied to the data before the application itself sees it, therefore simulating persistant storage. User mappings have two flavors that we discuss separately. Their key difference is how generally they can be applied to previously unseen incoming new data.

3.2.8 Standard ETL Transformations

Extraction, transformation and loading (ETL) is the name given in the literature to the process used to populate data warehouses. Data is typically sourced from many transactional databases, transformed to fit a new schema, and cleaned before it is loaded into the data warehouse and its materialized views. What we refer to as standard ETL transforms are operations that research papers [17, 55] describe as happening in the Data Staging Area. These include schema level transformations, various application specific mappings, and cleaning operations. Any changes defined by a user application that are general enough to be applied to new incoming data we consider standard ETL transformations.

3.2.9 Data Cleaning Changes

We mention data cleaning specifically as a special case of user mappings because these changes will generally not apply to new data as it arrives from the sources. These changes will generally be at the row-value level, such as, deletion of duplicates or corrections of wrong input, rather than general data cleaning operations like changing the format of date fields. A core reason for separating the two is the fact that before applying these

transformations we need to correctly identify their corresponding rows as they are loaded. This complication might cause problems, and must be kept in mind and solved in a system implementation.

3.2.10 Transformation Phase

This last phase before data is loaded into a user application will often prove to be the most time consuming. This is where incoming data is carefully audited and all transformations specific to one application are designed and implemented. Transformations can be divided into two classes that roughly follow along the lines of the user mapping groups above. The first class includes all data restructuring, operations such as dropping or combining columns, pivoting rows with columns and other schema-altering changes.

In most cases the application owner will have to perform a near-manual inspection of the data to identify data quality issues. Such issues can be anything from erroneous and missing values to duplicate records and type inconsistencies stemming from integrated data. Not only is the detection of dirty data issues difficult, but the design and evaluation of transformations to successfully rectify them can be very hard. In fact, data cleaning is so tedious in practice that studies have estimated that up to 80% of development time (and cost) in data warehousing projects is spent on just discovering and correcting this issue [21].

For an implementation of the transformation phase we recommend an interactive approach, similar to the one introduced by the Wrangler project [38]. A design like that aims to make it simple for users to find and express meaningful transformations while limiting the need for manual repetition.

3.2.11 User Application

A design goal of the system architecture is sufficient modularity, so that individual parts can be interchanged or replaced as needed. Different system modules generally focus on discrete problems, many of which represent whole bodies of research. An irreplacable heart of the system remains the data application, without which the feedback loop we are looking for would be impossible.

In a traditional data project the time consuming work of the transformation phase is paid back, hopefully, by value extracted from the data in some application. A user application could be any piece of software that takes data as an input, for example clever visualization or manual spreadsheet computations made by a business analyst. Sometimes data quality issues are not discovered before the runtime of the application. Our system would have to support on-the-fly data updates on those occasions.

3.2.12 Transformation Scripts

Logging user changes during the transformation phase or application runtime serves two main purposes. It is the genesis of all user mappings to be applied at a later execution time. Secondly, it serves as input for all modules that propagate these updates back to the original data sources.

Transformations in our systems should be recorded as an actionable script in some declarative data transformation language. Again the Wrangler project [38] provides the direction by introducing a language design, based on earlier languages [44, 52], that seems well suited for our purposes. The language operates on tabular data and is composed of nine classes of transformation operators:

- *Map* transforms a data row into zero or any number of data rows. Map operators include row deletion, value updates, arithmetic and splitting into multiple columns or rows.
- *Reshape* operators perform schema-level changes by either folding multiple columns to key-value sets or unfolding by creating new column headers from data values.
- *Positional* transforms update table values by using information in neighboring rows, generating new values or shifting values in place.
- *Schema* transforms change column names, data types and semantics.
- *Lookups, joins, sorting, aggregation* and *key generation* are further classes, mostly self-explanatory, that are not of key interest in our discussion.

The set of transformations from the Wrangler project is large enough to cover most common data wrangling and cleaning tasks. In fact, it is provably sufficient to handle all one-to-one and one-to-many data transformations [44, 52]. Despite this coverage, our system must take special care when it comes to schema changes. They must be treated so as to facilitate incremental updates of schema mappings, possibly to more than one data sources.

3.2.13 Transformation Inference Engine

As discussed at some length in Section 2.4 above, transformation logs may not always exist for outstanding data copies. In those cases it is worth considering whether one can be inferred by comparing the original data copy to the modified current instance. The desired output should be similar to the transformation script described above. The feasibility of this module

is not investigated in this study but it would likely draw from a wide variety of prior work on automatic data cleaning and integration [23, 30, 32]. This problem is even further complicated if changes to the data have been made at the source, or schema changes are lost.

3.2.14 Update Propagation to Sources

Since the most valuable data cleaning revisions happen at the application level, with domain experts applying their knowledge and adding constraints, the propagation of those updates back to corresponding data sources becomes a critical part of our system design. This module is also particularly challenging because its implementation and technology would likely rely on uncharted territory in active research fields, and a carefully constructed scripting language [38]. We note two different approaches for this module below, each with nice features but distinct challenges.

Not all updates are created equal. A data source administrator is most likely not interested in knowing about every single change made to copies of their data by application users. For example, if a user changes the attribute name of a column or splits it in two this change is probably not of value to the original data source. Therefore it shouldn't get propagated back. We are only interested in sending along updates that increase the overall value of the data. Such changes are those that increase data quality for all future users of the data and lessen the load on future data cleaning operations. Few schema level changes are likely to apply, so an implementation of our system would do well to focus on only propagating *Map* transformations, as defined above, from applications to data sources.

After updates have been propagated to the sources, our system must

keep track of that fact and not attempt to resend updates later. One way to implement this would be to organize updates with a stack, so that they are not revisited after they have been popped off and propagated once. Even better, if a data source rejects an update this should be remembered so the system doesn't attempt to propagate rejected updates back and forth the next time the application fetches or loads the same data copy. This would be another bit of metadata that could be stored alongside the user mappings, since they should reflect all past application changes.

SQL Update Generation

The first approach to an update propagation module takes a transformation script from the data application as input and generates a list of corresponding SQL statements, executable on the data sources. This goal raises a number of questions that need to be addressed in a successful implementation.

For one, although it has been established that a declarative scripting language is capable of expressing all data transformations it is not certain whether a clear and unambiguous correspondence exists between transforms and SQL statements. A further complication is caused by the fact that a transformation script would likely operate on numbered lines in a data table, possibly making it harder to keep track of individual tuples as time goes on. Even if such a one-to-one mapping could be established, a translation between the two would remain a difficult task.

Another fundamental question is *where* updates need to be propagated. If the application sources data from more than one origin some mechanism is needed to pinpoint the genesis of the particular set of data affected by an

operation. Integrating data from different sources might also have altered data or combined it in a way that adds to the complexity of knowing where it came from. A possible solution for this problem may be found in the concept of data lineage. A hot topic in recent literature, data lineage [11, 13] is essentially any piece of metadata that captures the history of data, where it came from and how it has been changed. For our purposes we would likely require a low level approach, one that captures data lineage at the tuple level.

Like when new data is loaded from sources to applications, we assume that table schemas will generally diverge to some extent. If possible, those changes are kept up-to-date in the form of schema mappings. After deciding which source is to receive a specific update the SQL generator must interface with the data exchange module, retrieve relevant schema mappings, and produce source-specific SQL statements.

Error Explanation

A slightly different approach to update propagation is the one of error explanation and action prescription introduced in a data cleaning system by Chalamalla et al [14]. Instead of translating the transformation script to SQL statements and giving only a data source administrator the power to either accept or reject individual updates, the methodology is now a twofold process.

First, when violations are discovered during data cleaning on the application side, an attempt is made to summarize them as accurately as possible using predicates on the data copy. This is called the *error explanation* phase. When errors in violating data tuples have been repaired, their lin-

eage is traced back to the sources where all tuples that made contributions to them are identified. They can now serve as input to a mechanism that would propose a course of action for the administrator to fix the perceived problem, the so-called *action prescription*. This idea is revisited in more detail in Chapter 5.

Chapter 4

Analyzing Case Studies

In an effort to further flesh out system requirements, and to show how the modular design discussed in the last chapter ties in with the standard operation of our system, we will now revisit the common use cases from Chapter 2. By showing how user action moves data through the system, and how modules interact in different scenarios, we try to motivate the current design while shedding light on problematic parts that need to be solved before an implementation can be achieved.

4.1 Scenario 1

4.1.1 Retrieving and Transforming the Data

The setup for our first main use case is as follows: A single application user checks out a data copy from one data source and stores it locally on his own server. One might think that this use case could be trivially extended to include multiple users and many sources but that leads to complications to be discussed in a separate section.

As a running example, we imagine the case of a beat reporter working for the National Basketball Association (NBA) who wants to put together an analysis on some current NBA players. To achieve this she needs to use data that is hosted in one of the NBA's databases, a sample of which

player_id	*last_name*	*first_name*	*team*	*salary*	*college*
231	james	lebron	cleveland	19.07	**null**
321	love	kevin	**minnesota**	13.67	ucla
343	duncan	tim	san antonio	**103.6**	wake forest
576	parker	tony	san antonio	12.50	**null**

Table 4.1: Sample data from a relational data source containing information about current NBA players.

can be seen in Table 4.1. The table contains information on player names, their team affiliation, salary, and what college they played for before joining the NBA. The missing data in the *college* column is caused by the corresponding players not having had a college career. The exact nature of the reporter's application is not important but for our purposes we can imagine some standard statistical analysis or visualization. Furthermore, we make no assumption on the reporter's technical aptitude when it comes to querying databases and assume that she must submit a request for the data, either through some interface or by contacting an administrator directly. We will however, assume that our reporter is very familiar with the data itself, and has enough knowledge to accurately judge the quality of the data.

The reporter is interested in checking out all the data contained in Table 4.1, except the player IDs. This subset of data could be defined by the simple query

```
SELECT last_name, first_name, team, salary, college
FROM Players;
```

assuming the name of the table. We interpret the reporter's request as being submitted by her application, and the process is followed as step 1 in Figure 4.1. In response, the database administrator will now create a

	last_name	*first_name*	*team*	*salary*	*college*
1	james	lebron	cleveland	19.07	**null**
2	love	kevin	**minnesota**	13.67	ucla
3	duncan	tim	san antonio	**103.6**	wake forest
4	parker	tony	san antonio	12.50	**null**

Table 4.2: A user defined subset of the source data. No transformations have been made.

structured data copy containing the result of the query. This process would be very similar to the creation of a materialized view, with the exception that the end result is not necessarily a relational table to be stored in a database, but more likely a simple spreadsheet-like dataset. In our example, the data copy resulting from this process will now be stored locally with the reporter and can be seen in Table 4.2. Note that, strictly speaking, the relational nature of the table has been broken and the numbered rows are only printed for reference.

For the system to work correctly, the data source must keep a record of all outgoing data copies. The semantics of the reporter's original query are given by what we have dubbed a *user data definition*: a tuple of metadata with information on the query that generated the data as well as all necessary contact details about the reporter. The user data definition is now stored with the data source, possibly in a relational table contained within its database management system.

When data is checked out of a source for use in a particular application for the first time, a one-to-one mapping will automatically exist between the schema of the data table that exists at the application side and the user data definition that is stored at the source. No changes have yet been made by either side, so the schemas remain the same. This information must

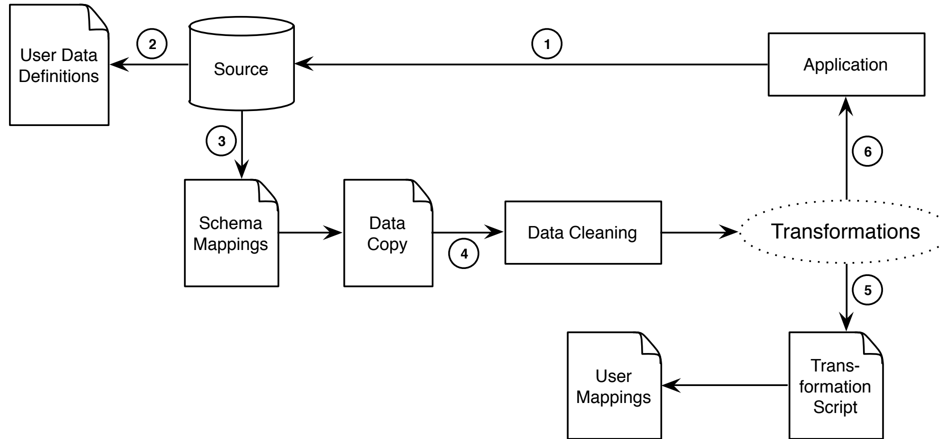

Figure 4.1: Scenario 1. The application makes a request to the data source. The data source reacts, supplies a data copy that is transformed and cleaned on the application side. All transformations are logged in a script, some are generalized as user mappings, and the data is now ready for the application.

still be encoded, and therefore the data exchange module on the application side is initialized with one-to-one schema mappings for every attribute. The recording of the user data definition and the initial schema mappings can be seen as steps 2 and 3, respectively, in Figure 4.1.

Having received the data copy of her choice, the reporter in our example can now start wrangling her data to fit the structure she needs for her application. At this point she will also start examining the data for inconsistencies or errors that need to be corrected. Our system is now in the transformation phase. The first thing that catches our analyst's eyes are the missing values in the *college* column. Being an expert in the data, she quickly realizes the reasoning behind the missing values. The reporter decides that this is insufficient for her analysis and makes a twofold change. In place of the missing college values, she inputs the name of the last team the

	last_name	*first_name*	*team*	*salary*	***before_nba***	*before_type*
1	james	lebron	cleveland	19.07	**akron**	high school
2	love	kevin	**cleveland**	13.67	ucla	college
3	duncan	tim	san antonio	**10.36**	wake forest	college
4	parker	tony	san antonio	12.50	**paris**	pro

Table 4.3: The user defined data after the transformation phase. The data can now serve as input for the user application.

corresponding player last played for before joining the NBA. Since none of the new values are names of legitimate college teams, the reporter renames the column *before_nba* and introduces a whole new column, *before_type* that explains at what level the team in the previous column plays. The data copy has now undergone a series of value updates and two schema changes, as can be seen in the last two columns of Table 4.3.

With the data now in the correct structural format for her analysis, the NBA reporter looks it over one last time and discovers a couple of anomalies. The first one is the exceedingly high salary of $103.6M listed for San Antonio's veteran star Tim Duncan. Due to restrictions on yearly salaries in the NBA this high amount is impossible. Because those restrictions are easily quantifiable and can be stated as a business rule to be applied to the data instance, they also fall into a category of data errors that are discoverable by semi-automatic data cleaning software. The report concludes that the decimal point is wrongly placed, likely due to a human input error. The other error is harder to discover automatically because that one is caused by an outdated value. The prolific power forward Kevin Love joined the Cleveland Cavaliers in a trade in August 2014 and this is not reflected in the data. Again the value of our analyst's domain knowledge is demonstrated for data cleaning. In Figure 4.1, the data cleaning and transformations made by the

reporter can be followed as step 4.

The nature in which an application user specifies his changes is important, because ideally we only want to allow robust transformations from a finite set of machine readable and reproducible operations. Since we also assume that the user interacts with the data through a spreadsheet-like interface, rather than programmatically, the goals of our transformation phase would be best met by a controlled interface where only robust changes are allowed. An example of this would be the Wrangler system interface [38], that looks like a common spreadsheet. The resulting changes are now logged in a declarative transformation language and stored on the application side. The semantics of a suitable transformation language need to be worked out; but to offer an idea, the resulting script from our example could look something like the following:

```
columnName('college').to('before_nba')
row(1).column('before_nba').setValue('akron')
row(4).column('before_nba').setValue('paris basket')
createColumn('before_type')
row(1).column('before_type').setValue('high school')
row(2).column('before_type').setValue('college')
row(3).column('before_type').setValue('college')
row(4).column('before_type').setValue('professional')
row(3).column('salary').setValue('10.36')
row(2).column('team').setValue('cleveland')
```

Logging all changes in this manner is almost sufficient, but two steps remain necessary for further housekeeping. The one-to-one mapping between the user data definition and the current table is now broken. To fix this

our system must be able to translate all schema changes in the transformation script, and update the schema mappings stored at the application side. The second necessary step, is to classify value updates into general transformations and specific transformations, the difference being that the former will apply to all new incoming data from the source in the future. General transformations get added to a special *user mapping store*. The logging of transformations corresponds to step 5 in Figure 4.1.

4.1.2 Propagating Updates to the Source

Now that all data pre-processing work is done, our reporter is ready to make her analysis, and our focus switches to the update propagation protocol. The task now is to salvage all updates that might be of value to other users of the data, and send them back to the sources. A walkthrough of this process can be seen in Figure 4.2. Again, some classification of the application changes is in order. We perform a rough pruning of the transformation script to leave behind changes that are unlikely to be relevant outside the scope of the specific application. Transformations that would be of global interest are mostly the ones resulting from data cleaning, so the division here is likely to be the same as in the earlier generation of user mappings. Changes that simply massage the structure of the data to fit the application needs, or involve some general calculations and functions would probably be left behind. However, this might not always be the case and needs a closer look in the future.

Once the system has picked the transformations that are to be propagated, the lineage of the tuples involved, stored through some mechanism on the application side, is examined to see what source it is affiliated with.

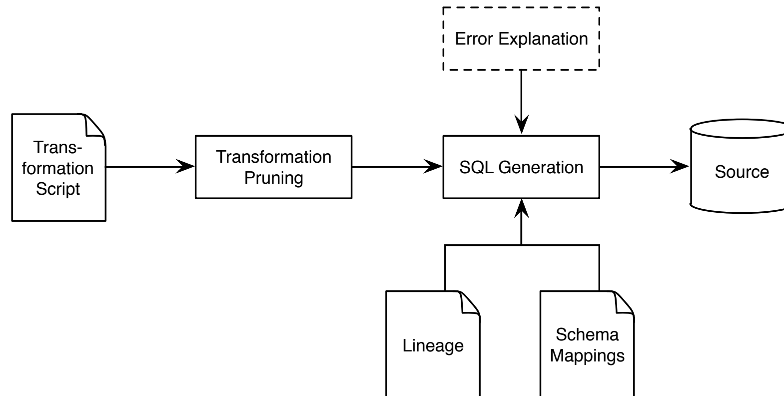

Figure 4.2: Scenario 1. Propagating updates to the source. First, the transformation script is pruned. Next, the remaining transformations are translated to SQL using the existing schema mappings and data lineage. The result is presented to the source. An alternative to SQL generation would be to attempt *error explanation*, as discussed in Section 3.2.12.

Lineage, also called provenance, is any metadata that captures the origin and history of a particular piece of data. We defer discussion on provenance to the next chapter, but stress the importance of keeping track of it for exactly this purpose. The NBA data in our example originates in one source, so in our case this objective is fairly straightforward. Even so, complications can quickly arise if the application data is the result of an aggregate query, and contributing tuples on the source side need to be pinpointed.

Chapter 3 introduced two fundamentally different approaches to how the system should proceed with update propagation. Now, knowing where particular transformations need to be sent back to, the first approach involves a module that would take as input the transformations in question as well as the necessary schema mappings to perform a data exchange operation with the source relations as targets. This module would generate

an executable SQL script for each source and notify its administrator, so that he can choose which parts of it to apply to the source. The goal of the second method we mentioned, would be closer to one of error explanation and action prescription. Instead of giving the source administrator a binary choice of updates to accept, we will now try to highlight and explain problematic tuples in the source, so that a more comprehensive course of action can be taken. The implementation of the second module needs some extra thought, but it would most certainly require even more detailed records of data provenance.

4.2 Scenario 2

We have dealt with the simple scenario of one source, one application, and the backward propagation of valuable updates. Now we consider a similar scenario from the perspective of new updates being made at the data source. This function of the system could be described as *forward propagation*. The motivation for this scenario was laid out in Chapter 2, and now we shall see how different modules of the system might interact to make it happen. We continue with our example from the previous section, and imagine that the NBA data source makes updates to its data at some point in time after all the work already described has been done.

Specifically, we will consider the following changes, as seen in Table 4.4. For illustration, we only imagine two basic changes at this point, but any number of them could have been made. We notice that the input error in the *salary* column for Tim Duncan has been corrected, presumably resulting from our earlier backwards propagation. The second change is the addition of a tuple representing the NBA's top draft pick in 2014, new player Andrew

player_id	*last_name*	*first_name*	*team*	*salary*	*college*
231	james	lebron	cleveland	19.07	null
321	love	kevin	*minnesota*	13.67	ucla
343	duncan	tim	san antonio	**10.36**	wake forest
576	parker	tony	san antonio	12.50	null
772	**wiggins**	**andrew**	**cleveland**	**4.59**	**kansas**

Table 4.4: The NBA sample data source after applying the highlighted updates. An earlier data error has been corrected, and a brand new tuple added representing Andrew Wiggins.

Wiggins. The astute reader might also notice the fact that the NBA analyst's update of Kevin Love's current team has been rejected by the data source administrator, possibly because the transfer has not been finalized, thus demonstrating the factor of choice in the application of presented updates.

Our task is the same as before, notifying all interested parties of changes made to data in the system, but now in the opposite direction. Simply reversing the roles of the source and application users is not sufficient because the heterogeneity of each party's technology causes issues that need to be addressed separately. An obvious first solution to this problem would be to regenerate every application user's data copy in its entirety, replace his current copy and reapply all his changes. This could potentially be a cumbersome operation if sources are updated frequently. It would also not function smoothly in the data integration case we will consider later, so we decide to choose a different path in search of a solution.

Since the overall purpose of our system is not to maintain full continuous synchronization between users, but rather to make meaningful updates globally for everyone to apply at their own discretion, we will not attempt a design to incrementally propagate each update. Instead, we propose that every data source maintain a separate *update buffer*, a queue of available up-

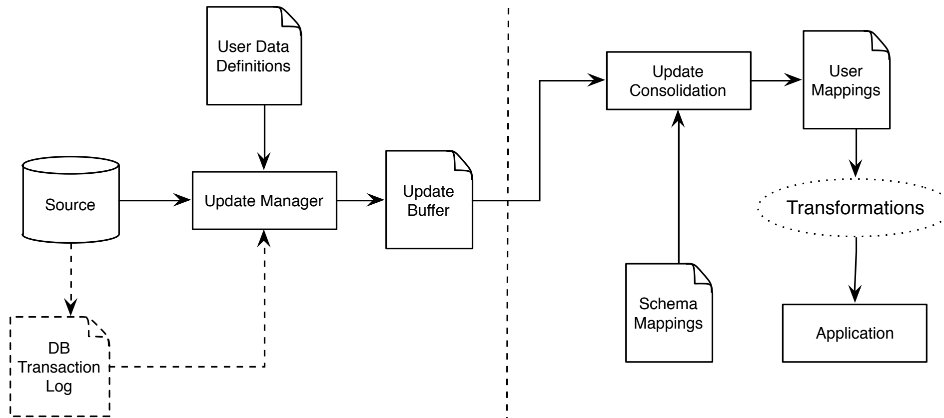

Figure 4.3: Scenario 2. Propagating new data and updates from a data source to the application. The update manager monitors the source for changes and loads the update buffers. The changes are then loaded, transformed, and examined on the application side.

dates for consideration, for each active user application. Once fresh transactions start to accumulate in the update buffer a flag of some sort is given to the corresponding application's manager so that he can at any point choose to refresh his data accordingly.

The protocol for loading updates into the update buffer requires some consideration. Again, a possibility presents itself to process each data source transaction on the fly, but the expected performance drop in throughput would be less than ideal. If possible, a cleaner solution could involve scanning the transaction logs of the database at regular intervals and processing them in bulk. However, deferring the work in that way does not come without its own problems. Changes to user data definitions in the period between database transactions and when updates are logged would affect the accuracy of the update buffer logs. For example, if an application user checks out the most recent copy of data before updates have been logged

	last_name	*first_name*	*team*	*salary*	*college*
1	duncan	tim	san antonio	10.36	wake forest
2	wiggins	andrew	cleveland	4.59	kansas

Table 4.5: The table of updated tuples in the buffer after the Update Manager has processed the fresh transactions on the NBA database. The table schema fits the original user data definition.

the update buffer would become inconsistent. One possible solution to this would be to associate timestamps with all user application actions and the update buffer.

The role of the *update manager*, as seen in Figure 4.3, is to process all the database transactions, and manage the update buffers on a per tuple level. Its input is received either after every transaction is committed or in bulk from scanning the transaction logs. For every affected tuple, the update manager must now consult the user data definition store to determine all the user application update buffers it belongs to. For each user data definition that a tuple is included in, it is now added to the corresponding update buffer in the same format as the resulting table of the original query. In our example, the changes made to the NBA database are processed by the update manager and a table of modified tuples, akin to Table 4.5, is added to the update buffer.

Every application user is free to fetch available updates for his data copy at his own discretion. Once a data update is initialized, the contents of the update buffer are transferred to the application user and the buffer is cleared. The new data is at first processed by the *update consolidation* module, on the application side, whose job it is to conform it to apply any up-to-date schema mappings necessary for data exchange, as well as to resolve any conflicting updates. Consulting Figure 4.3, we see how the updates are

	name	*team*	*salary*	*before_nba*	*before_type*
1	tim duncan	san antonio	10.36	wake forest	**null**
2	andrew wiggins	cleveland	4.59	kansas	**null**

Table 4.6: The available data updates as presented to the application user, after user mappings have been applied. Note how data in the last column has been lost.

now subjected to all general user mappings that have been prescribed to incoming data, resulting in a data instance conforming fully to the current state of the original data. These tuples are now presented to the application user to decide whether they should be merged with his current data copy. Table 4.6 demonstrates how the updates are represented at this time in the case of our running example.

To help the application user determine which of the available updates to keep and which to discard, it would be helpful if all tuples clearly indicated whether they constitute new data, deleted data or altered data. Once the changes to be applied have been chosen, the remaining data essentially enters another round in the transformation phase as described in the previous section. Whether manual or with the help of some data cleaning software, new tuples especially must undergo an inspection to correct for possible data quality issues. For our purposes, the only change made during this phase would be to replace the *null* in Andrew Wiggins' tuple with the class *college*. Generally, if any issues are discovered, the application user makes the appropriate transformations and they get logged as before, prompting a new round in our system's propagation carousel.

At this point, the data updates need to be merged with the current data instance. This is easy in the case of new data tuples, they are simply appended to the table. A harder problem comes with deleted tuples, since

the matching tuples have to be found and removed. An even harder problem still is the case of updated tuples. The matching tuples must not only be found but merged and consolidated with the updates arriving from the source. Care must be taken to interchange the data values that constitute the update, while at the same time retaining earlier transformations. In some cases, data might even have been lost, like we see in the Tim Duncan tuple of our example. Because the value of the *before_type* column was added on the application side and not propagated back, the source has never come in touch with that value before. For our system to be effective, this merging of updates with existing data in the tabular world of the application side must be solved.

The motivating case study for this scenario in Chapter 2 touched on a number of problems and loose ends that our simple example here doesn't encounter specifically. We mentioned the analogy that exists for this task with the maintenance of materialized views in data management systems. The important difference of not insisting on full synchronization, in conjunction with the loss of a strict relational schema on the application side make our problem a different animal. Similar issues still have to be addressed. For one, we have not formulated any update strategy for either party in our system, and leave that for future work. An even harder problem not addressed by our example is when user data definitions contain any form of aggregation. This introduces a flurry of issues that need to be resolved in an implementation of a system inspired by our design.

4.3 Scenario 3

The basic operation of our system can mostly be demonstrated by following the example of one data application working with a data copy from one source. Some important parts of the system and their associated challenges only come to light when a more complicated scenario is considered. Increasing the number of data applications in the system doesn't change much, and has mostly been addressed in describing the system architecture at the source. However, when the number of data sources is increased from one, complications stemming from data integration must be solved. We will now describe the functional operation of that scenario. Just like the corresponding scenario in Chapter 2, we will also drop the assumption of full local storage on the application side, now only leaving space for necessary metadata.

For this data integration scenario, we assume that all data sources involved are related through some already established universal schema. If this is not the case, with disparate sources that only make sense in the context of the application, the work of constructing the universal schema is laid on the application user's shoulders. For our purposes, we shall only consider the case where such a schema exists and represents the only interface the user has with the sources as a whole. The initial request for data is now made by the user through this interface, and a *user data definition* is stored like before, only now it is made to the universal schema and is maintained on behalf of all data sources at once.

Like in any data integration scenario, the relation between the multiple sources and the universal schema is stored as a set of schema mappings for each source. This approach is included in our system, but its data integra-

tion role should be contrasted with the schema mappings maintained for data exchange between the source and application in our earlier example. Since none of the actual data is now stored locally on the application side, structural changes by the user can now be included as general user mappings whenever data is loaded from the sources. If schema mappings at this second stage become necessary, they should be defined so as to express the relationship between the application data and the universal schema.

Limiting local storage means that all of the actual data copy is fetched from its sources every time an application is run. The only things to reside on the application side are definitions of user mappings and schema mappings, data provenance, transformation scripts, and other necessary metadata to ensure seamless operation. This storage design has both advantages and new challenges that must be faced. On the one hand, updates and new data coming from the sources need not be maintained at the source and propagated in a special process as in our earlier scenario. Fresh data and updates in the data integration scenario now arrive naturally with already seen data. However, we contend that our system should still keep track of updates at the sources for each user application. This is so that every time the data is fetched, an application owner can be presented with a segmented section of unseen updates and new data like before. Given that view of his data, the user can now inspect it for data quality issues more easily and prescribe appropriate transformations.

The tradeoff for this new convenience is that a user application can in no way manipulate data in a persistent manner. This is especially problematic since this scenario is fairly common in practice and, without a system like we propose, the options to make corrections are slim to none. We get around this caveat by keeping transformation scripts and user mappings as before,

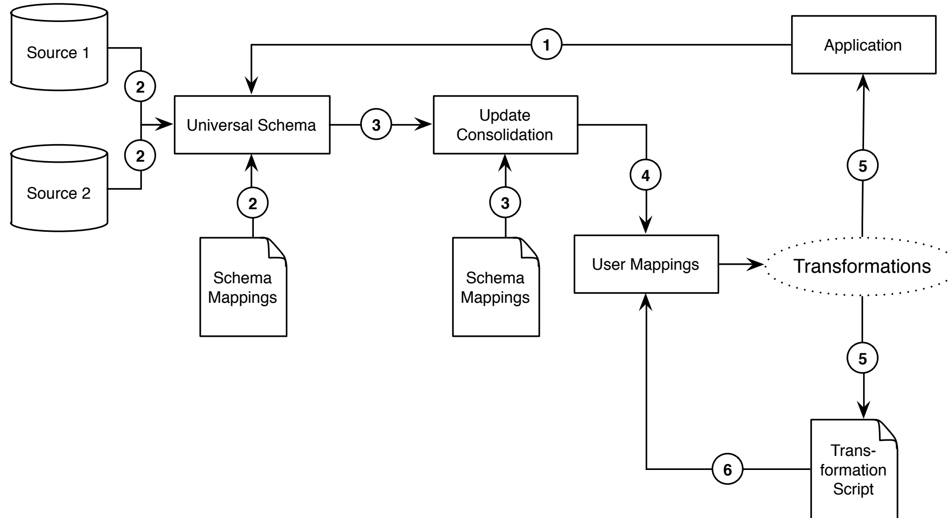

Figure 4.4: The data integration scenario. The lack of local storage on the application side complicates schema mappings and update consolidation. The forward propagation from sources is relatively easier, but user mappings must be applied at runtime. Step 2 represents the data integration system, replacing the single data source of Scenarios 1 and 2.

and applying them to the whole bulk of data every time the application needs it. Before, user mappings were thought to include mostly generic transformations that apply equally to all new incoming data. Now we need user mappings to be more specific and able to apply all previous user defined changes.

With the problem of persistent transformations taken care of, we have addressed the main issue with the lack of local storage. Still left for discussion are complications on the application side introduced by the addition of multiple data sources. This is where the *update consolidation* module shoulders much of the work. Its main task is to resolve conflicting updates that might be coming from different sources, and in doing so, making sure

that any updates that involve the same row of the application data play nicely together. This should be more problematic when data is integrated and kept in local storage but in our scenario care must still be taken so that user defined transformations and mappings are not negatively affected by changes made at some data sources.

The slightly more involved system schema needed to handle this scenario can be seen in Figure 4.4. Since our interest here is still confined to only a single data application, the backward propagation of updates to the sources has not changed. Hence, this will largely follow the pipeline from Figure 4.2, with the only change being the addition of a data integration system, consisting of one or more data sources.

4.4 Scenario 4

The careful reader will remember that our case studies in Chapter 2 included a fourth scenario. In that scenario, the task was to infer unknown changes, made to a data copy, when no transformation script had been recorded. As we discussed in Chapter 2, such a task is quite intricate and out of the scope of our preliminary system design. Since our design does not have a provision to solve this problem, we will not include a section on it here, and leave it as an important topic for future research.

Chapter 5

Related Work

5.1 Collaborative Data Sharing Systems

Sharing data across interconnected communities is a difficult problem and increasingly visible in data integration research. Building on earlier work on peer data management systems [29] (PDMS), the paradigm of *collaborative data sharing systems* [25, 36, 37, 40] addresses the current challenges of scientific data sharing. As part of the CDSS project, its authors have developed the Orchestra system as a prototype.

As described by Green et al. [25], a CDSS is a data integration system that consists of a set of peer databases, each with its own independent schema and local data instance. Peers are connected to each other by compositional schema mappings that explain how their instances are derived from each other and differ. This architecture, originally devised for a PDMS, alleviates the need for a universal schema common in data integration and thus facilitates schema evolution. The CDSS paradigm differs from a PDMS by materializing all data locally at the peers. Users of a database only make local queries and only a database administrator, the curator, is concerned with integrating data from other sources. Local materialization allows the peer administrator to cherry-pick data and updates from other parts of the system based on trust conditions.

The CDSS supports local edits by peers to facilitate data disagreements within the community. To implement this, while still ensuring schema mapping consistency, deleted tuples are stored in a *rejection table* and newly inserted tuples in *local contribution tables*. If local a curator discovers mistakes in the data, corrections can be propagated back to original data sources through bidirectional mappings. When a peer administrator requests an update exchange from the wider system, the peer's local edit log is published globally. An incremental update translation is then performed using all other globally available edit logs.

The premise of sharing data between peers in a CDSS is quite analogous to the interaction model between users that we propose with our system. A key difference is that a CDSS assumes that all data is stored locally. Our approach is more lightweight, with data generally stored at designated sources and logical copies of it used by data applications. Other differences are that in a CDSS the peers are homogeneous in their functionality, and they are all relational databases and potential sources for other peers. Our system has a collection of data sources on one side and a set of data applications on the other. These two types of "peers" in our system are technically different and must be approached as such. In addition, a data application is likely solving a different problem than a data source, thus having a contrasting perspective on the data while still having the goal of increasing its overall quality.

Two ideas are directly applicable from the CDSS design to our system. One is the way peer changes are stored in local edit tables and only shared globally when requested by other peers. Our transformation script store could follow this design. The other key contribution is the mechanism for incremental update exchange, including schema mapping handling, that ad-

dresses a number of problems in upstream and downstream propagation, and can be efficiently implemented with the use of provenance information.

5.2 Data Transformation Systems

The value in the system we envision, is mostly created in the data transformation phase, after an application user has successfully checked out or loaded a copy of his data from the sources. This is where the important data cleaning happens, as well as all structural manipulations that serve the application. After the transformation phase, our whole system is only in the business of making sure these changes are maintained and made available to other users.

The need to transform and clean data before using it for analysis or as application input is ubiquitous in an increasingly data centric industry and a real effort has been made to meet the needs for effective solutions to the problem. From a large body of research on transformation languages [1, 18, 38, 44, 44, 52], we will highlight two publications that set the foundation for a project we see as the most likely solution so far.

To improve the interoperability of heterogeneous relational databases with regards to querying and data exchange, Lakshmanan et al. [44] constructed the SchemaSQL transformation language. A natural extension to SQL, SchemaSQL supports the restructuring of databases to fit different schemas, partly by treating schema information and other metadata in the same way as the actual data.

In an effort to integrate exploratory data cleaning efforts and transformation scripting into a graphical interface, Raman and Hellerstein introduced Potter's Wheel [52]. They implement a set of common transformations for

both values and rows, and prove that this collection can cover all possible one-to-many row mappings.

A recent project, directly related to Potter's Wheel, vastly improves on its predecessor. The first result of the project was Wrangler [38], a system where users can build transformation scripts in a spreadsheet-like interface to manipulate data, or "wrangle" it as the authors call it. Intended to reduce the need for manual editing and individual scripts, Wrangler lets users specify robust transformations that result in a editable and reusable script. The authors define a declarative transformation language that extends the language from Potter's Wheel, and make the claim that it allows for near complete coverage of all practical data wrangling transformations. In a continuation of the project, the team introduced Profiler [39], a data cleaning system that combines automatic anomaly detection with a visual summarization feature. The Wrangler project and subsequent work constitutes the most complete solution we know of for an implementation of the data transformation phase of our system. The Wrangler system might therefore even be made to serve as a module in our system, that is the transformation module that lies between data sources and applications. As an alternative to the Wrangler transformation system, we can briefly mention Google Refine [33]. Like Wrangler, Google Refine takes tabular data as input, but its graphical command capabilities are more limited. Google Refine does however include some useful data cleaning features, such as entity resolution and discrepancy detection.

A slightly different approach is taken by Stonebraker et al., in an architecture design for what they call a *data curation system* [53]. Their main goal is data integration, but in doing so they address issues with data cleaning and transformations. Although sharing the emphasis on non-programmatic

interfaces with the Wrangler project, the authors argue that scalability can only be reached through automation with the help of machine learning. This line of reasoning is promising but still at a very early stage in research. Our system assumes mostly manual data edits and does not yet include a focus on automation.

5.3 Data Provenance

For the past two decades, the topic of data provenance and lineage has garnered growing attention from groups in the data management research community. Simply put, provenance is any information about where data originated, how it has been transformed since then, and why it appears in its current context and place [4]. A good survey of early provenance research is given by Buneman et al. [12]. Buneman has continued where he left off and summarized more recent efforts [10], stressing the relevance of provenance and calling for increased attention to the issue. Despite being increasingly important for scientific research and other practical implications, provenance ideas are still at a very early implementation stage in enterprise solutions [16].

This work of Buneman is rehashed at length by Cheney et al. in a recent survey [19]. It expands on the previous survey by covering important new topics, most notably the notion of how-provenance. Originating with the Orchestra project [36], the idea of how-provenance as put forth by Green at al. [25] aims to explain what transformations a piece of data has been subjected to. The authors do this by proposing a new general model for provenance, one based on a framework of *semiring-annotated relations*, and demonstrating how this model encompasses previously proposed models as

special cases while encoding more information. The work by Green et al. [25] also demonstrates the flexibility of provenance information by using it as a basis for a system of trust policy enforcement between data sources, as well as a means of efficiently implementing update propagation. Both ideas could prove fruitful in our system, with the latter demanding serious consideration in any implementation. Being a system of update propagation, it is clear that how-provenance should be a first class citizen in our system's metadata management.

One application that is sometimes highlighted in the literature is the capability of querying provenance [16, 25]. This notion is implemented in the Orchestra system to help end-users and data curators explore the often complicated provenance graphs. Karvounarakis et al. [41] developed the ProQL query language for provenance to support this implementation. The ability to query provenance information directly should be readily applicable in our system, notably for decision support in components like the update consolidation store.

By definition, as soon as any provenance information is maintained, data quality has been improved. Aside from that, provenance as a tool can help solve a flurry of problems ranging from data integration and query languages to debugging schema designs and software. In most cases, the task at hand will dictate the need for one of many different provenance models. They differ fundamentally in the granularity of provenance chosen, with the spectrum covering task-based workflow at one end down to a single value level at the other. Buneman and Davidson [11] give an overview of provenance models for different tasks while arguing the need for model unification in future research.

Early implementation attempts stored provenance information as meta-

data, quickly requiring a vast amount of storage for fine-grained models with many transformations. Provenance storage has been shown in practice to grow larger than the base data, sometimes by many factors [16]. Woodruff and Stonebraker suggested an early remedy [57] to this problem, by computing provenance lazily instead of using complete storage and only having knowledge of minor details of transformations undergone by the data. A more recent effort by Chapman et al. [16] to solve this problem has resulted in a number of algorithms, based on ideas of factorization and inheritance, that managed to reduce provenance storage by a factor of 20 in experiments. Bose and Frew have written a survey [9] on provenance in scientific workflow systems where many lineage system implementations are chronicled.

In context of our system design, it is clear that provenance information will play a key role. It should be incorporated from the ground up and serve as the main artifact of what data sources a data tuple belongs to and who maintains a copy of it. To serve this purpose, we contend that a model must be chosen that works at least at the tuple level of granularity. In addition, it is also feasible that how-provenance be maintained that encodes past transformations at the same level. We recognize that finding an efficient provenance model and implementation is a major challenge that remains unsolved in the wider propagation system.

5.4 Data Exchange

Data exchange is a problem fundamental to many data management tasks, most notably data integration. The problem is defined as taking data structured under a source schema, and translating it to accurately fit some other pre-existing target schema. The restructured data should be the best possi-

ble representation of the data under the original schema. An overview of the theory underlying data exchange is given Fagin et al. [24]. The translation from one schema to the other is achieved by following so-called *source-to-target dependencies*, expressed in some logical formalism. The source-to-target dependencies encode how the schemas relate and what constraints the structure of the data instances must follow. They are the rules to the game and must be maintained as the schemas change.

The foundation of data exchange systems draws on elements from research on data integration [28, 45, 46]. Data integration relates data sources and their schemas to a global schema that queries can be posed against. Data exchange can be modeled as data integration from one source to the global target schema, or as integration in both directions. A difference between the two is that a target schema typically exists beforehand in data exchange whereas it is specially constructed for data integration. The target schema in data integration is also most often virtual, but the data instances are physically materialized in data exchange.

Data integration and data exchange are the two main tasks in what has been dubbed data interoperability [27]. Underlying both is the need to express relationships between different schemas. This can be done at two contextually different levels. In *schema matching* [6, 49] syntactic correspondences are generated that establish relationships between elements of one schema and elements of another schema. A more operational level is the one of *schema mappings* [27, 31, 47], where the focus is on moving instances of data from a source to a target. The source-to-target dependencies used in data exchange are a form of schema mappings. Fagin et al. [24] designate the semantics of data exchange on one hand and query answering on the other as the two main challenges. For our purposes, we are almost solely

interested in the former, namely moving data from source to target.

The first system to generate schema mappings in a semi-automatic manner between two relational databases was the *Clio* system [24, 27, 47]. Clio takes schema matching correspondences as input, and generates source-to-target dependencies as their interpretation. The schema mappings are then fed into a query generation tool that produces an executable transformation script in some language, SQL for example.

In our system, the data application users would either load data from sources in a data integration scenario, or check out their own copy for local storage in a data exchange setting. An implementation to solve either scenario would require schema mappings to be generated and maintained. For data exchange we would need mappings established between a source schema and a target schema. In the data integration scenario, where one data application loads data from multiple sources, we would require mappings from every source to a single virtual universal schema. If however an application only loads data from one source without local storage it is possible that schema changes can be inferred from user mappings, without knowing exact schema mappings. A system like Clio seems to be a good fit as a tool to generate schema mappings in our system. Its functionality as a black box is what we are looking for, but an added complexity in our case is that schemas on the application side are not necessarily relational. However, Clio was designed as a practical tool for industry from the start, and besides relational schemas it can now handle XML schemas [27]. It remains to be examined what constraints must be enforced on data application schemas in our system so that a schema mapping tool like Clio can be used in the context we need.

Manipulating schemas of data sources and the mappings that relate them

are common operations in data integration systems. To define and investigate a general set of operators for these tasks, in different contexts and data models, is the goal of Model Management [5, 7]. Aside from all operators that help with the data exchange and integration parts of our system, special attention should be paid to operators devised for schema differencing. They could bring possible value to the table in situations where schema mappings are incomplete or have been lost. If a transformation script is not in place schema differencing could help shed light on structural changes that a data application user has made.

5.5 Probabilistic Databases

Interest in probabilistic databases resurged with a publication by Dalvi and Suciu [20]. Drawing their idea from the way queries are defined in information retrieval systems [3], they wanted to address the problem of answering structurally rich SQL queries with possibly uncertain predicates, and ranking the results. Uncertain predicates lead to uncertain matches, so a probability value must be assigned to every tuple in the database. The problem at hand is now one of evaluating queries over database tables with uncertainty, and the authors devise efficient algorithms to do just so.

Research on probabilistic databases is expanded on, and integrated with ideas about provenance, in the Trio project [4, 56]. They present a new data model, dubbed ULDB, that treats *uncertainty* and *lineage* as first class citizens in a database management system. Uncertainty is encoded in the system at both the attribute and the tuple level, representing confidence levels of all possible data instances. The relationship between different tuple-alternatives and how they are derived is then stored as lineage. The

end product is the *Trio* prototype system that sits as a layer on top of a traditional, relational DBMS. Queries in the system are posed with a strict extension of SQL that the authors describe in the same work.

The idea of combining two different hot topics of research into one system, where they can complement each other in an effort to solve some classic problems of data management, is the key reason for our interest in the Trio system. Like Trio, which combines provenance and uncertainty, our goal is to design a composite system that combines data exchange and integration with transformations and lineage. Differences between the two systems are that ours does not yet involve ideas on uncertain data, and is at a very early stage in its conception. The takeaway is the direction of combining interesting topics into composite systems that can be used for solving practical problems.

5.6 Data Cleaning

Data sets are very often plagued by various issues, anomalies, and errors of all different kinds. These errors often originate at the source, whether that source is some scientific sensor equipment or simply an input error made by a human. Common sources of anomalies found in practice are the inconsistencies that can be introduced by data integration. Efforts to enumerate common data errors have been widely published [21, 23, 30, 42, 50]. Among many problems that have been addressed, techniques have been introduced to detect outliers [15, 32], duplicate records [23], and key violations [30]. Reducing redundancy in databases by entity resolution [8] is another important topic.

The onus to create better data cleaning systems is great. Studies have

estimated the cost of correcting data cleaning issues in large data projects to hover around 80% of total project cost [21], and the cost of flawed analysis due to errors in data runs in billions of dollar per year [22]. Although current solutions have made huge strides, most of them are still semi-automatic in the sense that they are designed to flag potential issues. These issues must then be checked by a human and corrected by manipulating the appropriate data, possibly covering some unflagged tuples when dealing with derived data.

Many systems have been implemented to solve different tasks in data cleaning [27, 35, 38, 39, 52]. The need to solve different tasks is dictated in part by the flavour of data management project the data cleaning effort is a part of. Many systems, often focusing on duplicate detection, are purposely built to help with data integration. They will certainly be needed in our context for just that task, but of even greater interest is the direction taken by the Profiler system [39]. With a main emphasis on issues occurring within a single relational table, Profiler flags anomalies that include missing and erroneous data, inconsistent or extreme values, and key violations. These are the same issues we expect our application user to be dealing with in the data cleaning phase of our system. The seamless integration of Profiler with the Wrangler transformation system [38]—the two having been developed together—make the two an interesting possibility as modules in the wider system we imagine.

5.7 Extraction, Transformation, and Loading

As the role of data management systems increasingly shifted to meet the need for decision support, data warehouses were invented. Data ware-

houses [17] and on-line analytical processing (OLAP) gained popularity in the 90s and have only grown in importance since. A data warehouse is a "subject-oriented, integrated, time-varying, non-volatile collection of data that is used primarily in organizational decision making." [34] As such, it emphasizes summarized historical data over the transactional variation of regular databases, therefore facilitating temporal or categorical analysis.

Data warehouses are typically populated by data from one or more on-line transactional databases, the operational databases that serve on the frontlines of data retrieval and storage. The process of moving data from operational data sources, changing it appropriately and using it to populate a data warehouse is called *extraction, transformation, and loading* (ETL) [55]. The two biggest challenges faced in the ETL process are the integration of data from multiple sources on one hand and performing data cleaning and all necessary transformations on the other.

Extraction is the task of defining the subset of data in a source that is to be imported to the warehouse and querying for it in bulk. This must be done with the least amount of interference in the standard operation of the source, and is therefore often performed at night. Comparing snapshots of data to pinpoint incremental updates is a way to lessen the load of the process [43]. The *transformation* step is where the data can undergo the range of changes needed for the particular warehouse operation. These include data cleaning, solving data integration issues, and other transformations at schema-, instance-, and value-levels that we have described in detail above. The changes also include ones that are uniquely common in data warehouses, such as replacing keys with surrogate keys to improve performance and adding redundancy by pivoting tables and attributes. The last step of the ETL process is *loading* the data into an existing warehouse.

This usually involves a bulk operation and must consider both the merging of existing data with updates as well as maintaining all materialized views and indexes of the warehouse.

The ETL pipeline has many parallels with moving data back and forth in our system. When a user application imports data from one or more sources to be kept in local storage, that is a complete ETL process. Important differences still remain. The first one is our data integration scenario where the user application never keeps a copy of data in its storage. Another difference is that the ETL process only works in one direction. Data warehouses and their views are designed to be end products, their data is rarely updated and doesn't send feedback to data sources. Furthermore, data warehouses are generally large and expensive enterprises. They are costly to design and maintain and are usually deployed for analysis in larger data projects. Our system is designed to handle smaller, lighter data application projects and should be general enough that a difficult setup process is not needed.

5.8 View Maintenance

A materialized view is the stored result of a query on some database that is either kept remotely or locally in a redundant fashion. This is different from logical views, whose virtual tables are never actually stored. The purpose of materializing views is usually to increase query performance. When large tables need to be queried at high cost, it can be useful to instead have access to a pre-computed table requiring less work. This is particularly useful in speeding up analytical queries on data warehouses [54].

In addition to needing additional storage, the largest overhead of materialized views comes with maintaining them. Whenever the underlying

source data changes, the materialized view must be updated accordingly to perfectly reflect its original query. This is called *view maintenance*. One approach would be to recompute the whole view, but that wouldn't scale too well. A better solution would be to opt for incremental view maintenance. Surveys can be found on both materialized views [26] and view maintenance algorithms [2].

In Chapter 2, we mentioned how the relationship between a data source and an application's local data copy was akin to the one between a database and a materialized view. Propagating updates from the source would then be analogous to maintaining a materialized view. A key difference is that in our case, we do not strive for full synchronization. Nevertheless, the incremental update techniques of view maintenance are likely to be of value to our system.

Chapter 6

Conclusions and Future Work

In this thesis we proposed a novel framework for update propagation in collaborative data sharing systems, and suggested a high level system architecture as a first step in implementing that framework. As data gathering efforts continue to expand and data applications become ever more important across society, we are motivated by a strong need for greater sophistication in data management systems.

We classified three use cases that generalize to countless situations that should ultimately be covered by a system following our design. Composed of a number of separate modules, each carrying the load of an important function, the overall system draws from ideas and prior work that span a significant part of current data management research topics. We explained the role and importance of each system function in the context of its relevant prior work, and mapped out requirement guidelines and possible issues for future implementation efforts.

The high level system overview we described here only represents the early stages of our group's focus on update propagation and we can identify a number of challenges that must be overcome for the system actually to be realized in some form. A particularly sensitive assumption in our work

is the need to somehow force application users to only perform meaningful transformations on the data. This could be enforced through a specialized version of spreadsheet software, and has been done by others as we have seen, but a much preferred option would still be an integration with standard spreadsheet software while somehow limiting the user's freedom to make random changes that are difficult to parse programmatically.

Another major challenge that we have not discused at length, is the added complexity of propagating updates between players when an application user has checked out a data copy through an aggregate query. These by definition make a single row of application data dependent on any number of data rows at the source. This can lead to difficult problems in update propagation on both sides of the table. When propagating to the source, a syntactic meaning of changing the aggregate value must be interpreted on the original rows, and on the flip side, value changes or additions/deletions of data rows at the source must be reflected in recomputed aggregate values at all relevant applications.

The fact that we assume all data stored locally at the application side is only in a structured relational format, and not in a full-fledged database table, needs to be considered closely before implementation and could cause some unforeseen difficulties. As an example, we have discussed the need for incoming data updates from the source to identify with the correct rows in the application data copy for consolidation. We mentioned this in passing, but in situations where these rows are now only recognized by a row number this matching could be quite difficult. It is also easy to see how careless and inconsistent management of such row numbers can lead to numerous issues in the system operation over time.

A somewhat related issue ties in with how the system should segment

between general ETL transformations and more tuple-specific data cleaning transformations. We defined the former as any changes that are generally applicable to all incoming new or reloaded data on the application side. We suspect that the boundary between the two could at times become somewhat fuzzy and this would have to be addressed in the system. In the data integration scenario, when data is fetched from the sources at application runtime, the correct matching between all prior user mappings and the incoming data needs to be established. Implementing the functionality is likely to suffer from this concern.

Two more system modules have thus far only been laid out in rough drafts and might be challenging when it comes to integration. The first is the schema mapping store. Although schema mapping storage between two related schemas is common in data integration, arbitrary changes on either side in our system would call for incremental updates to schema mappings. Since such changes could potentially be triggered by different modules of the system, and affect mappings between more than one schema, this could prove to be a tough task. The second module whose operation needs more consideration, is the one handling update consolidation. This module should adjust all incoming data from sources to the correct schema and form of the application data. In addition, we want this phase to consolidate any conflicting updates—updates that might arrive from different sources in a data integration scenario and affect the same data row. Presumably, merging updates should not be a big problem but design choices need to be made about how the system should handle conflicting updates to the same data values in a particular row.

The last leap of faith we mention that will prove challenging forms one of the key steps in update generation. Given a transaction log of all user

defined changes, one suggested way of propagating updates involved generating an executable SQL script, in the source schema, and presenting it to its administrator. First of all, interpreting the transformation script into SQL is no easy task, but great care must also be taken that errors are not introduced in this phase since the purpose of its output is to alter the original source data. Although it is a logical and necessary step in one of the update propagation schemes, it remains to be seen whether this module will be a feasible option in implementation.

As our first effort in this direction we feel we have laid out a vision and necessary groundwork for the continuation of this project. Much work is still needed before a working prototype of our system can be achieved. Obviously, meeting the challenges drafted above will be a big factor in getting to that goal but we can identify further steps that need to be taken and will now conclude this report by listing some possible milestones of future work.

A big step necessary for the realization of such a system is to develop a suitable provenance model. We have described how in other work provenance research has been molded in recent years to fit particular objectives, and how formalisms at different granularities have been used in applications. Many of the challenges we have described could be met by a well designed provenance model as well as other questions that might arise. One would be deciding how the system should deal with rejected updates once they have been suggested. Should they be kept track of for a later recommendation, or should they be discarded? What about updates made later to values affected by earlier rejected updates? What if an update affecting more than one user is rejected by one and accepted by the other? Those are examples of questions that could affect choices about how data history should be kept in provenance records.

Another task at hand is to develop update strategies for different actors in our system. The means and ways of choosing when data is loaded, or when new updates are either fetched or pushed, can have large effects on the system operation and design. It is necessary to map out the subtle effect different update strategies might have. In doing so, it is also important to determine exactly what connection an implementation of our system would have with the notion of materialized views in database systems. We have discussed some preliminary similarities but just as obvious are important differences. Nonetheless, it is likely that ideas and algorithms from view maintenance research would be applicable to similar problems in our framework.

In Chapter 3, we described two different formalisms on propagating updates. One involved generating an executable SQL script from the transformation logs and presenting it to the data source, while the other is an attempt to identify erroneous tuples in the data source and describe the issues it might have. Both methods should be examined in future work to determine which is more suitable to the objectives of our system.

In addition, we floated a couple of new features that would improve the usability of the end product, if implemented. The first would be functionality to rank the results of update propagation suggestions. This might be possible by imposing an order of importance on different types of updates or implementing varying trust conditions on different update sources from the receiver's perspective. Again, the feasibility of both would rely on the choice of provenance model. The second improvement we propose, is the means of grouping updates into composite aggregations in parts of the system and describing them in human readable form. This would improve user comprehension, and could possibly be implemented both when data trans-

formations are prescribed as well as before update suggestions are presented to a user.

The way we described the data sharing pipeline in earlier chapters would imply that a data application owner is able to examine and clean all his data prior to inputting it to the application. This should often be the case but we must still consider that certain data errors might not be discovered until runtime by the application user. To accommodate this fact it would be helpful to look into how our system could interface with the application to handle real-time feedback about data quality issues.

At last, we reiterate the unsolved problem of the transformation inference engine. Given data from a source and a modified copy of some subset of that data, is it possible to infer an executable script to transform the latter copy to the former? This is obviously a big question to ask and would require much effort to answer fully. However, if possible, an inference engine like that would bypass the requirement of an existing transformation script before propagating valuable updates to other users. This would prove to be beneficial to many existing data sharing interactions since our system would not need to be in place right from the start.

The topics we have specifically mentioned here as possible next steps in no way represent a complete collection of future research avenues. Many still remain, and future challenges are bound to surface. Once further progress is made, we foresee a system that could benefit a great number of applications and would significantly reduce the cost of repeated, manual labour in many data processing pipelines.

Bibliography

[1] Serge Abiteboul, Sophie Cluet, Tova Milo, Pini Mogilevsky, Jerome Siméon, and Sagit Zohar. Tools for data translation and integration. *IEEE Data Eng. Bull.*, 22(1):3–8, 1999.

[2] D. Agrawal, A. El Abbadi, A. Singh, and T. Yurek. Efficient view maintenance at data warehouses. *SIGMOD Record*, 26(2):417–427, June 1997.

[3] Ricardo Baeza-Yates and Berthier Ribeiro-Neto. *Modern information retrieval.* ACM press New York, 1999.

[4] Omar Benjelloun, Anish Das Sarma, Alon Halevy, and Jennifer Widom. ULDBs: Databases with uncertainty and lineage. In *Proceedings of the 32nd international conference on Very Large Data Bases*, pages 953–964. VLDB Endowment, 2006.

[5] Philip A Bernstein. Applying model management to classical meta data problems. In *CIDR*, volume 2003, pages 209–220, 2003.

[6] Philip A Bernstein, Jayant Madhavan, and Erhard Rahm. Generic schema matching, ten years later. *Proceedings of the VLDB Endowment*, 4(11):695–701, 2011.

[7] Philip A Bernstein and Sergey Melnik. Model management 2.0: ma-

nipulating richer mappings. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, pages 1–12. ACM, 2007.

[8] Indrajit Bhattacharya and Lise Getoor. Collective entity resolution in relational data. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 1(1):5, 2007.

[9] Rajendra Bose and James Frew. Lineage retrieval for scientific data processing: a survey. *ACM Computing Surveys (CSUR)*, 37(1):1–28, 2005.

[10] Peter Buneman. The providence of provenance. In *Big Data*, pages 7–12. Springer, 2013.

[11] Peter Buneman and Susan B Davidson. Data provenance–the foundation of data quality, 2013.

[12] Peter Buneman, Sanjeev Khanna, and Tan Wang-Chiew. Why and where: A characterization of data provenance. In *Database Theory-ICDT 2001*, pages 316–330. Springer, 2001.

[13] Peter Buneman, David Maier, and Jennifer Widom. Where was your data yesterday, and where will it go tomorrow? data annotation and provenance for scientific applications. In *Position paper for NSF Workshop on Information and Data Management (IDM 2000): Research Agenda into the Future, Chicago IL*, 2000.

[14] Anup Chalamalla, Ihab F Ilyas, Mourad Ouzzani, and Paolo Papotti. Descriptive and prescriptive data cleaning. In *SIGMOD*, pages 445–456, 2014.

[15] Varun Chandola, Arindam Banerjee, and Vipin Kumar. Anomaly detection: A survey. *ACM Computing Surveys (CSUR)*, 41(3):15, 2009.

[16] Adriane P Chapman, Hosagrahar V Jagadish, and Prakash Ramanan. Efficient provenance storage. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, pages 993–1006. ACM, 2008.

[17] Surajit Chaudhuri and Umeshwar Dayal. An overview of data warehousing and OLAP technology. *ACM Sigmod record*, 26(1):65–74, 1997.

[18] Weidong Chen, Michael Kifer, and David S Warren. Hilog: A foundation for higher-order logic programming. *The Journal of Logic Programming*, 15(3):187–230, 1993.

[19] James Cheney, Laura Chiticariu, and Wang-Chiew Tan. Provenance in databases: Why, how, and where. *Foundations and Trends in Databases*, 1(4):379–474, 2009.

[20] Nilesh Dalvi and Dan Suciu. Efficient query evaluation on probabilistic databases. *The VLDB Journal*, 16(4):523–544, 2007.

[21] Tamraparni Dasu and Theodore Johnson. *Exploratory Data Mining and Data Cleaning*. John Wiley, 2003.

[22] Wayne W Eckerson. Data quality and the bottom line. *TDWI Report, The Data Warehouse Institute*, 2002.

[23] Ahmed K Elmagarmid, Panagiotis G Ipeirotis, and Vassilios S Verykios. Duplicate record detection: A survey. *Knowledge and Data Engineering, IEEE Transactions on*, 19(1):1–16, 2007.

[24] Ronald Fagin, Phokion G Kolaitis, Renée J Miller, and Lucian Popa. Data exchange: semantics and query answering. *Theoretical Computer Science*, 336(1):89–124, 2005.

[25] Todd J Green, Grigoris Karvounarakis, Zachary G Ives, and Val Tannen. Provenance in ORCHESTRA. *IEEE Data Engineering Bulletin*, 33(3):9–16, 2010.

[26] Ashish Gupta and Iderpal Singh Mumick. *Materialized views: techniques, implementations, and applications.* MIT press, 1999.

[27] Laura M. Haas, Mauricio A. Hernández, Howard Ho, Lucian Popa, and Mary Roth. Clio grows up: from research prototype to industrial tool. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, pages 805–810, 2005.

[28] Alon Halevy, Anand Rajaraman, and Joann Ordille. Data integration: the teenage years. In *Proceedings of the 32nd International Conference on Very Large Data Bases*, pages 9–16. VLDB Endowment, 2006.

[29] Alon Y Halevy, Zachary G Ives, Dan Suciu, and Igor Tatarinov. Schema mediation in peer data management systems. In *Proceedings of the 19th International Conference on Data Engineering*, pages 505–516. IEEE, 2003.

[30] Joseph M Hellerstein. Quantitative data cleaning for large databases. *United Nations Economic Commission for Europe (UNECE)*, 2008.

[31] Mauricio A Hernández, Renée J Miller, and Laura M Haas. Clio: A semi-automatic tool for schema mapping. *ACM SIGMOD Record*, 30(2):607, 2001.

[32] Victoria J Hodge and Jim Austin. A survey of outlier detection methodologies. *Artificial Intelligence Review*, 22(2):85–126, 2004.

[33] David Huynh and Stefano Mazzocchi. Google refine.

[34] William H Inmon. *Building the data warehouse*. John Wiley & Sons, 2005.

[35] Zachary Ives, Craig Knoblock, Steve Minton, Marie Jacob, Partha Talukdar, Rattapoom Tuchinda, Jose Luis Ambite, Maria Muslea, and Cenk Gazen. Interactive data integration through smart copy & paste. In *CoRR*, 2009.

[36] Zachary G Ives, Todd J Green, Grigoris Karvounarakis, Nicholas E Taylor, Val Tannen, Partha Pratim Talukdar, Marie Jacob, and Fernando Pereira. The ORCHESTRA collaborative data sharing system. *ACM SIGMOD Record*, 37(3):26–32, 2008.

[37] Zachary G Ives, Nitin Khandelwal, Aneesh Kapur, and Murat Cakir. Orchestra: Rapid, collaborative sharing of dynamic data. In *CIDR*, pages 107–118, 2005.

[38] Sean Kandel, Andreas Paepcke, Joseph Hellerstein, and Jeffrey Heer. Wrangler: interactive visual specification of data transformation scripts. In *CHI*, pages 3363–3372, 2011.

[39] Sean Kandel, Ravi Parikh, Andreas Paepcke, Joseph M. Hellerstein, and Jeffrey Heer. Profiler: integrated statistical analysis and visualization for data quality assessment. In *AVI*, pages 547–554, 2012.

[40] Grigoris Karvounarakis, Todd J Green, Zachary G Ives, and Val Tannen. Collaborative data sharing via update exchange and provenance. *ACM Transactions on Database Systems (TODS)*, 38(3):19, 2013.

[41] Grigoris Karvounarakis, Zachary G Ives, and Val Tannen. Querying data provenance. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, pages 951–962. ACM, 2010.

[42] Won Kim, Byoung-Ju Choi, Eui-Kyeong Hong, Soo-Kyung Kim, and Doheon Lee. A taxonomy of dirty data. *Data mining and knowledge discovery*, 7(1):81–99, 2003.

[43] Wilburt Labio and Hector Garcia-Molina. Efficient snapshot differential algorithms in data warehousing. *Proceedings of the International Conference on Very Large Data Bases*, pages 63–74, 1996.

[44] Laks V S Lakshmanan, Fereidoon Sadri, and Subbu N Subramanian. SchemaSQL: An extension to SQL for multidatabase interoperability. *ACM Transactions on Database Systems (TODS)*, 26(4):476–519, 2001.

[45] Maurizio Lenzerini. Data integration: A theoretical perspective. In *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of Database Systems*, pages 233–246. ACM, 2002.

[46] Alon Levy, Anand Rajaraman, and Joann Ordille. Querying heterogeneous information sources using source descriptions. In *Proceedings of 22th International Conference on Very Large Data Bases*, pages 251–262, 1996.

[47] Renée J Miller, Laura M Haas, and Mauricio A Hernández. Schema

mapping as query discovery. In *Proceedings of the International Conference on Very Large Data Bases*, volume 2000, pages 77–88, 2000.

[48] Raymond T Ng, Patricia C Arocena, Denilson Barbosa, Giuseppe Carenini, Luiz Gomes, Jr, Stephan Jou, Rock Anthony Leung, Evangelos Milios, Renée J Miller, John Mylopoulos, et al. Perspectives on business intelligence. *Synthesis Lectures on Data Management*, 5(1):1–163, 2013.

[49] Erhard Rahm and Philip A Bernstein. A survey of approaches to automatic schema matching. *The VLDB Journal*, 10(4):334–350, 2001.

[50] Erhard Rahm and Hong Hai Do. Data cleaning: Problems and current approaches. *IEEE Data Eng. Bull.*, 23(4):3–13, 2000.

[51] Raghu Ramakrishnan and Johannes Gehrke. *Database management systems*. Osborne/McGraw-Hill, 2003.

[52] Vijayshankar Raman and Joseph M. Hellerstein. Potter's wheel: An interactive data cleaning system. In *Proceedings of the International Conference on Very Large Data Bases*, pages 381–390, 2001.

[53] Michael Stonebraker, George Beskales, Alexander Pagan, Daniel Bruckner, Mitch Cherniack, Shan Xu, Verisk Analytics, Ihab F. Ilyas, and Stan Zdonik. Data curation at scale: The data tamer system. In *In CIDR 2013*.

[54] Michael Teschke and Achim Ulbrich. Using materialized views to speed up data warehousing. *Technical Report, IMMD 6. Universitat Erlangen-Nurnberg*, 1997.

[55] Panos Vassiliadis and Alkis Simitsis. Extraction, transformation, and loading. In *Encyclopedia of Database Systems*, pages 1095–1101. Springer, 2009.

[56] Jennifer Widom. Trio: A system for data, uncertainty, and lineage. *Managing and Mining Uncertain Data*, pages 113–148, 2008.

[57] Allison Woodruff and Michael Stonebraker. Supporting fine-grained data lineage in a database visualization environment. In *Proceedings of the 13th International Conference on Data Engineering*, pages 91–102. IEEE, 1997.