

# CPSC 504 Project

## An investigation into an application to manage data transformations and updates

Laura Cang  
cang@cs.ubc.ca

Kailang Jiang  
jiangkl@cs.ubc.ca

Jessica Wong  
jhmwong@cs.ubc.ca

April 14, 2015

# 1 Introduction

This project builds on the theoretical framework described in the M.Sc thesis of Arni Thrasarson [13] and focuses on the first of the four scenarios detailed in the work. The first scenario investigates how changes made by a data application can be captured and propagated to a remote data source. The main workflow of this scenario involves a (possibly third party) application pulling the data from a remote data source to use locally. In order to facilitate updates to the data source, the application receives a schema mapping to identify the source table(s) of the data attributes along with a copy of the data it is requesting. The user using the application after retrieving the data may choose to add, remove, or edit the data. These changes are then logged to a transformation script which is later used to update the main copy of the data in the remote data store.

In this project, we focus on three different ways a user can use the application to fetch data. For the rest of this paper, we shall call each of the following a sub-scenario.

1. When the user chooses to view all the columns in one table
2. When the user chooses to view a subset of the columns in one table
3. When the user chooses to view a subset of columns in multiple tables

Our primary contributions are:

1. Defining a data transformation language using logical programming concepts to capture changes made to a relational data model
2. Proposing how the data source could react and handle schema changes
3. Investigating view maintenance policies for the remote data source
4. Detailing scenarios in which an application views a subset of columns from one or more tables in a remote data source
5. Deciding on the division of responsibility between the database administrator and the application user in regards to data updates

## 2 Motivation

We believe that the three sub-scenarios outlined are realistic. The first sub-scenario can be easily envisioned when thinking about a librarian trying to re-catalogue all the books in a library. The librarian would need to have all the information about a book in order to catalogue and double check the information.

As a motivating example for using a subset of one table, we consider a school administrator who is deciding to which student to award an academic merit prize. The school administrator could go to the main database storing all student information (e.g., name, student number, GPA, guardian contact information, date of birth, allergy/medication information, family doctor contact, etc.) but only wishes to access student number and GPA columns to ensure that the prize is awarded fairly.

Lastly, users might need to view a subset of columns in multiple tables if they are a journalist intern who has been tasked with compiling a report for an online journal. The report could be on differences in critic ratings and user ratings for animated movies produced in the last five years which would require the intern to look at information from a movies and user information table..

## 2.1 Sample data used in our project

In our project, we decided to borrow the dataset used in CPSC 304, an introductory relational database course at UBC, to use as the test dataset in our application. Specifically we use the “MOVIESTAR” table, which has three columns “STARID”, “NAME”, and “GENDER”. After supplementing some additional data to the dataset, namely the columns “MOVIE TITLE”, “PRODUCTION STUDIO”, and “CRITIC RATING”, we used this dataset. We then created a second table detailing user information using the column “Movie Title” as a foreign key to reference the “MOVIE TITLE” column in the Movie Information table, “SOURCE WEBSITE OF USER INFORMATION”, “USER RATING”, and “NUMBER OF LIKES”.

## 3 Solution

The basic workflow of the scenario described in Section 1 remains the same. However, as the user obtains more flexibility over what columns can be chosen to view in his/her application, the complexity of the data transformation increases due to the amount of information that must be tracked (see Section 3.2). In this project, we have decided to support the following user changes: adding a column, removing a column, renaming a column, editing a table cell’s value, removing a table cell’s value, adding a row, and removing a row. Any changes that cause a violation in schema definitions (e.g., removing a row’s primary key) will be resolved on the server side by the database administrator. This design decision was due to not wanting to restrict the types of users that would be able to use an application to explore and update data. As we have to assume that there could be users unfamiliar with schema/storage restrictions, we decided to ensure that schema violations are resolved by someone with the technical understanding of what error has occurred.

### 3.1 Types of Changes Supported

We have decided to focus on two of the eight changes mentioned in Wrangler [6]: map and reshape. Map changes deal with “[transforming] one input data row to zero, one, or multiple output rows” [6]. This essentially means that the row changes that are supported in our application (e.g., add row, delete row, edit value(s) in row(s)) all fall within this class of changes. Reshape changes deal with schema changes. Any change to the columns (e.g., add/remove/rename columns) will fall into this category. The scope of this project does not deal with changing the data type of a column.

## 3.2 Transformation Language

During the design of the transformation language, we realized that it did not make sense to start with the simplest of our three sub-scenarios as the transformation language did not scale well with increasing complexity. Therefore, we are basing our decisions around the most complicated of our three sub-scenarios: the situation where an application has pulled a subset of columns from more than one table. To ensure that the transformations worked for the simpler sub-scenarios, every step was aligned with all consequences in mind.

When designing the data transformation language, we decided to emulate some of the logical programming concepts from Prolog (none too biased by our instructor’s well-known preference for Datalog). We primarily borrowed from the syntactical style and the easy-to-understand format. The addition of a timestamp was in preparation for a possible future extension of the project where multiple concurrent users would make updates to the data source. Table 1 has a summary of how the structure of each data transformation will look like for each function.

User Action	Data Transformation Syntax
Add Row	addRow(tableName(columnName1:valuesUserPutIn1, columnName2:valuesUserPutIn2, ..., columnNameN:valuesUserPutInN), timestamp)
Remove Row	removeRow(tableName(primaryKey1:primaryKeyValue1, primaryKey2:primaryKeyValue2, ..., primaryKeyN:primaryKeyValueN), timestamp)
Add Column	addCol(tableName, nameOfNewCol, timestamp)
Remove Column	removeCol(tableName, nameOfColToRemove, timestamp)
Rename Column	renameCol(tableName, oldNameofCol, newNameOfCol, timestamp)
Edit Value	editValue(tableName(originalPrimaryKey, ..., originalPrimaryKeyN, columnNameOfEditedValue1:editedValue1, columnNameOfEditedValue2:editedValue2, ..., columnNameOfEditedValueN:editedValueN), timestamp)

Table 1: The data transformations logged to the transformation script as the user makes changes to the data.

The rest of this paper will use the following tables:

- Movie Information(Movie Title: String, Primary Key; Movie Debut Year: Int; Production Studio: String, not null; Total Gross: double; Critic Rating: String)

- User Information(Movie Title: String, Foreign Key; Website that user information was obtained from: String, Primary Key; User Rating: String; Number of Likes: double)

Once an application has pulled data from a remote database, the resulting set of columns shown in the application will be referred to as a view. For the journalist intern example mentioned in Section 2, the columns being viewed are “MOVIE TITLE”, and “CRITIC RATING”, both of which come from the Movie Information table, “USER RATING”, “NUMBER OF LIKES”, and “SOURCE WEBSITE OF USER INFORMATION” columns, which come from the User Information table. The Movie Information table has “MOVIE TITLE” as the primary key while the User Information table has “MOVIE TITLE” as its foreign key and “SOURCE WEBSITE OF USER INFORMATION” as its primary key. Table 2 shows what the displayed view after the user has pulled from the remote database. Note that schema mappings are also pulled to the application along with the data from the remote database. The schema mappings will be used to help determine what tables each of the respective columns in the view belong to.

Movie Title	Critic Rating	User Rating	Number of Likes	Website that user information was obtained from
Despicable Me 2	5/5	5/5	220 000	IMDB
Up	5/5	5/5	225 000	Facebook
Kung Fu Panda 2	4/5	5/5	500 000	Rotten Tomatoes
Mars Needs Moms	3/5	3/5	103 000	Rotten Tomatoes

Table 2: An example user-application display prior to the user making any changes to the data.

A constraint on the view is that the primary key(s) of each table must be displayed. This is a necessary detail as the primary key is needed to be able to uniquely identify the value being changed by the user (see Section 3.2.6 for more details). Also, when a user decides to add a new row to the table (see Section 3.2.1 for more details), there needs to be values for the primary key(s) in order for a row to be added successfully. Allowing the user to choose which rows he/she wants does not guarantee that the primary key is chosen and this may result in a seemingly inexplicable loss of certain application functions. Therefore, when pulling information from each table, the application is required to pull all primary and foreign keys of the table. While we recognize that this will cause a lot of activity on the network and possibly take up more local memory, we could not think of a better solution. Due to this design decision, the example shown in Table 2 does not include any details about whether the columns shown are the primary keys of their respective tables as we expect that the primary keys are already shown. There has been some discussion regarding how to make it easier for the user to determine primary key columns as opposed to other columns but we did not feel that non-technical users could understand the difference intuitively. For users that can understand why the idea of different columns belonging to different tables, it would certainly make it easier for them to perform actions such as add or remove a row (see Sections 3.2.1 and 3.2.2) but for the very non-technical users, it would only make it more distracting and confusing. Also, as we are possibly working with third party applications,

we can only restrict what information gets pulled, we cannot expect anything in terms of how the information is visualized to the users.

### 3.2.1 Adding a row

When the user uses his/her application to add a row in the view, it is unclear whether or not the user desires to include an additional row in all tables that have columns in the view. One of the design factors we have decided on is to include a new row in the tables that have a value filled in at the column.

In Table 3, the user has just added in a new row and populated the “MOVIE TITLE” and “CRITIC RATING” columns with a value. If the user decides to not populate the row any further and push his/her changes, the remote database would have the Movie Information table add a new row with Megamind as the “MOVIE TITLE” value and 3/5 as the “CRITIC RATING” value. However, as the Movie Information table has other schema restrictions, namely that the “PRODUCTION STUDIO” column must be populated, there will be an error generated when the user chooses to push to the remote database. We have decided to let the database administrator deal with issues like these at the remote database end. The alternative solution is to impose another restriction upon the application such that it will have to also pull columns that have schema specifications like “unique” or “not null”. However, we felt that it would be too confusing for the end-user to have such columns included. With primary/foreign keys, the information captured in those columns gives the requisite context and meaning to rest of the information in that row. Columns that have schema specifications may not have such context and meaning and thus would be confusing to impose at the application end.

In Table 4, the user has added more information about the movie “Megamind”. As columns from the Movie Information table and the User Information table have been filled with values, there will be a new row added to the Movie Information and User Information table. Any issues with adding to the Movie Information table (e.g., missing a value for a column denoted not null in the database) will have to be resolved by the local database administrator. Changes that are definitely going to be rejected (e.g., adding in a row without a value for the primary key(s) will be rejected by the data source without needing input from the database administrator).

Movie Title	Critic Rating	User Rating	Number of Likes	Website that user information was obtained from
Despicable Me 2	5/5	5/5	220 000	IMDB
Up	5/5	5/5	225 000	Facebook
Kung Fu Panda 2	4/5	5/5	500 000	Rotten Tomatoes
Mars Needs Moms	3/5	3/5	103 000	Rotten Tomatoes
Megamind	3/5			

Table 3: The user is trying to add a new row to the table where only columns from the Movie Information table is filled.

Movie Title	Critic Rating	User Rating	Number of Likes	Website that user information was obtained from
Despicable Me 2	5/5	5/5	220 000	IMDB
Up	5/5	5/5	225 000	Facebook
Kung Fu Panda 2	4/5	5/5	500 000	Rotten Tomatoes
Mars Needs Moms	3/5	3/5	103 000	Rotten Tomatoes
Megamind	3/5		200 000	IMDB

Table 4: The user is trying to fill in a new row that she has added. Columns from the Movie Information and User Information table have values added to them.

When the user adds a row, the change that is logged to the transformation script has the following structure:

```
addRow(tableName(columnName1:valuesUserPutIn1,
  columnName2:valuesUserPutIn2,
  ...,
  columnNameN:valuesUserPutInN), timestamp).
```

### 3.2.2 Removing a row

There are two methods for a user to delete a row in the data store. The first method involves selecting the "Remove Row" button while the second method is to delete all values from cells associated with that row in the table. Imposing only cell-at-a-time deletion when manipulating from the table helps avoid incidents where users accidentally delete whole rows when they just want to delete a single cell. We thought that this reflects how a user would normally use the system—if a user really wanted to delete a row, the whole row would be deleted anyways. As we felt that this accurately reflected a normal usage situation, we did not consider this as a constraint that imposed extra annoyance upon the user.

When the user removes a row by the "Remove Row" button, the change that is logged to the transformation script has the following structure:

```
removeRow(tableName(primaryKey1:primaryKeyValue1,
  primaryKey2:primaryKeyValue2,
  ...,
  primaryKeyN:primaryKeyValueN),
  timestamp).
```

We use the primary key to identify which specific row to delete. The database system will handle delete propagation (i.e., deleting the rows that other tables have which use values referencing the ones in the deleted row).

Removing values of all cells related to that row may also effectively delete the row. Further value-editing details are described in Section 3.2.6.

### 3.2.3 Adding a column

When adding a column, we considered two main possibilities: having a column added to all tables that have projections in the current view or adding to only one table. Originally we chose to have the column added to all tables that have representation in the current view but quickly realized that it would lead to a lot of update problems including potentially increasing the number of otherwise null-valued rows. There would also be the problem of using extra memory. In the end, we decided to have the column added to the table with the most columns in the current view. If tables have an equal number of columns in the view, final destination for the column can be chosen in one of three ways:

- Arbitrarily/at random
- Based on the name of the table (e.g., add the column to the table with the name that is listed first when table names are sorted alphanumerically)
- Based on the statistics of table usage that the database stores (e.g., the column could be added to the table that is queried more often). However, this would involve some back and forth dialog between the database and the remote application which may not be a good idea for performance or database security measures

We decided to choose the second method to resolve any possible conflicts between table choice when adding columns. The add column change is logged as follows:

```
addCol(tableName, nameOfNewCol, timestamp).
```

### 3.2.4 Removing a column

While discussing this function, there was some concern about whether or not users should be allowed to remove a column and if they were, whether they should be allowed to remove primary key columns. In the end, we decided to only allow the removal of non-primary key columns for database security reasons. It could be argued that the user should be able to do whatever he/she wishes upon the data as that is the point of enabling users from all backgrounds to use and update data but upon weighing the pros and cons of this issue, it was decided that it was too easy for the user to cause major damage to the data source if primary key removal was allowed. This would severely limit the availability and use of the data when scaled to allow for concurrent users. Thus we decided on a requiring that users who wish to remove a primary key column would require the aide of the database administrator. The "Remove Column" change would be logged into the transformation script as follows:

```
removeCol(tableName, nameOfColToRemove, timestamp).
```

### 3.2.5 Renaming a column

Although the situation where a user would rename a column is uncommon, we decided to enable support for this use case regardless. The transformation script syntax for this change is:

```
renameCol(tableName, oldNameofCol, newNameOfCol, timestamp).
```



### 3.2.6 Editing a value

When editing a value, a user can make one of two types of changes, defined as follows:

- 1-1 —change from one value to another
- 1-0 —value is deleted

As with the add row functionality (see Section 3.2.1), if a user ends up violating schema constraints when editing a value, the database administrator will have to handle the changes. Some of the possible schema violations include changing a unique value to a non-unique one when the column specifies that it needs to have unique values or deleting a value from a column that has forbidden null values. The "Edit Value" change would be logged into the transformation script as follows:

```
editValue(tableName(originalPrimaryKey1,..., originalPrimaryKeyN,  
    columnNameOfEditedValue1: editedValue1,  
    columnNameOfEditedValue2:editedValue2,  
    ...,  
    columnNameOfEditedValueN:editedValueN).
```

The originalPrimaryKey1 to originalPrimaryKeyN variables hold the values of the primary keys for the row prior to any changes. The initial primary key values are needed to identify which row is to be changed; if one of the primary key values have been edited, we would not be able to correctly identify the rows to be changed without the initial primary key values.

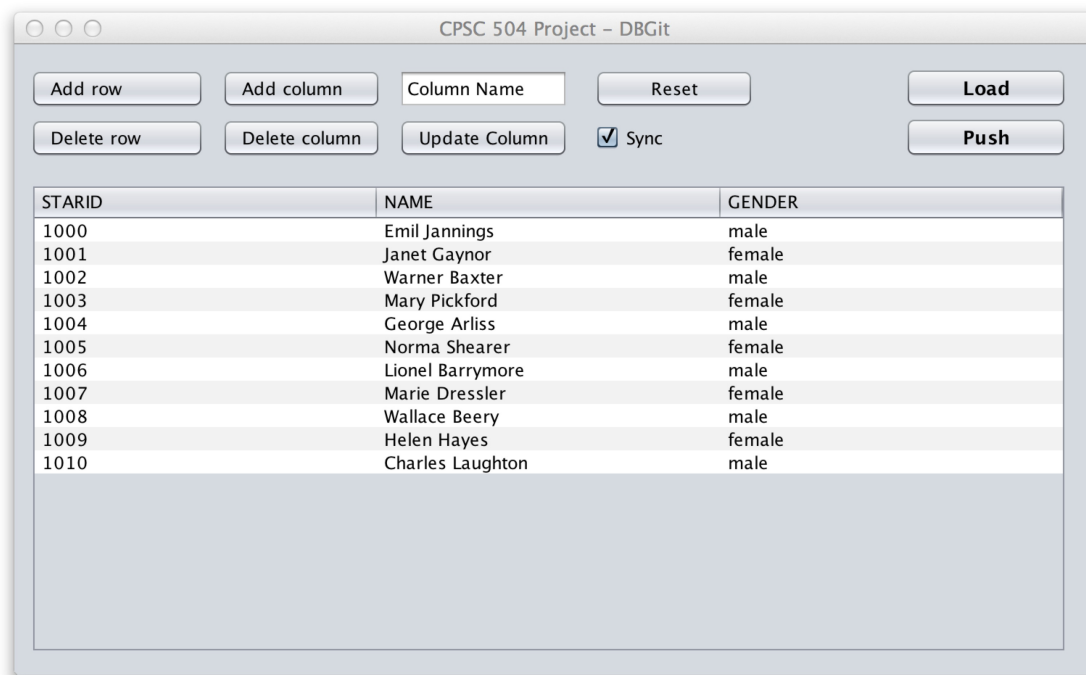
## 3.3 View Maintenance

View maintenance is also something that factored into our considerations throughout the project. There were long discussions about whether or not we should allow the user to choose when updates would be pushed to the remote database or if the application should choose. In the end, we decided on a mixed-initiative approach where the user would push if certain types or a certain number of changes have occurred while also allowing the user to choose when to push changes. The primary concern was that schema changes were important enough to push right away (see Section 3.2.1, 3.2.2, 3.2.3, and 3.2.4) especially since we acknowledged that as the wait period grew, the potential effects of bad information grew accordingly. If this project is later extended to include multiple concurrent users, the longer schema changes are withheld, the greater the potential is for affecting a large number of users. Value changes are considered less important; for value changes that conflict from user to user, we elected to refer to the timestamp for resolution decision. The application can push to the remote data source after a threshold number of changes. We have left the definition of the automatic push threshold as a future work.

### 3.4 Proof of Concept

After designing the transformation language, we decided to implement the scenario from [13] in order to show that our transformation language and design decisions could work to update a data source. A critical feature that we added was to allow for the user to choose which changes to sync to the remote data source. We created an interface using Netbeans that enabled users to load data from data source, make changes (while selecting whether to sync those changes), and push selected changes to remote data source.

Figure 1 shows the original MOVIESTAR table data we extracted and loaded from the data source. We are only showing three columns: the first is a unique id of a movie star (i.e. the primary key of the table), the second is the name of the movie star, and the third is the gender of the actor. Currently, we only load the specific data and columns, but this can be easily extended to letting users choose which columns and rows to fetch. Every time “Pull” is clicked, we discard all the un-pushed changes and load the data from the remote data source. Keeping the remote data source distinct from the application view means that a “Pull” without a “Push” is effectively a local “Undo”.



STARID	NAME	GENDER
1000	Emil Jannings	male
1001	Janet Gaynor	female
1002	Warner Baxter	male
1003	Mary Pickford	female
1004	George Arliss	male
1005	Norma Shearer	female
1006	Lionel Barrymore	male
1007	Marie Dressler	female
1008	Wallace Beery	male
1009	Helen Hayes	female
1010	Charles Laughton	male

Figure 1: Original data loaded from remote data source

Now, we can try to make some changes. We will track all “Map” changes (i.e., adding/deleting a row, and editing the value) while “Reshape” changes (i.e., creating/delete/renaming a column) are only logged to the transformation script if the user wishes for the changes to be synced to the remote data source. Figure 2 shows a few changes (add column “height” and “weight”, and rename the first column to be “ID”) we made to the original data that we chose to not sync and keep locally. Figure 3 shows some changes (add column “age”, add a row, and update some cells) that we chose to log and sync.

CPSC 504 Project - DBGit

☐ Sync

ID	NAME	GENDER	Height	Weight
1000	Emil Jannings	male	182	150
1001	Janet Gaynor	female		
1002	Warner Baxter	male		
1003	Mary Pickford	female	168	
1004	George Arliss	male		
1005	Norma Shearer	female		
1006	Lionel Barrymore	male		130
1007	Marie Dressler	female		
1008	Wallace Beery	male		
1009	Helen Hayes	female		
1010	Charles Laughton	male		

Figure 2: Changes that users choose not to sync

CPSC 504 Project - DBGit

☒ Sync

ID	NAME	GENDER	Height	Weight	Age
1000	Emil Jannings	male	182	150	
1001	Janet Gaynor	female			
1002	Warner Baxter	male			
1003	Mary Pickford	female	168		
1004	George Arliss	male			
1005	Norma Shearer	female			
1006	Lionel Barrymore	male		130	
1007	Marie Dressler	female			
1008	Wallace Beery	male			
1009	Helen Hayes	female			93
1010	Charles Laughton	male			
1011	Ethan Hawke	male			44

Figure 3: Changes that will be logged and synced

Synced changes will not apply to the remote data source immediately. Instead, we generate a transformation script, and log every synced change to it using our transformation language. For example, operations in Figure 3 will be logged in our script and once the user selects the “Push” button, we will translate and apply the transformation script to the remote data source. If we then click on “Load”, the tool will load data in from the data source again (see Figure 4). We can see that the data loaded in from the data source is slightly different from the data we saw before reloading. This is because only part of the changes made to our local data will be synced to the remote data source. If we load again, some changes we made to the local data copy will disappear.

STARID	NAME	GENDER	Age
1000	Emil Jannings	male	
1001	Janet Gaynor	female	
1002	Warner Baxter	male	
1003	Mary Pickford	female	
1004	George Arliss	male	
1005	Norma Shearer	female	
1006	Lionel Barrymore	male	
1007	Marie Dressler	female	
1008	Wallace Beery	male	
1009	Helen Hayes	female	93
1010	Charles Laughton	male	
1011	Ethan Hawke	male	44

Figure 4: Modified data re-load from remote data source after push

And also, the execution result of our transformation script and our translation of these scripts are proved to be correct by comparing these results to the data set before we “Push” and re-“Load”.

## 4 Related Work

### 4.1 Transformation Languages

While there has been a large body of work done on various data transformation tools [6, 11, 7], these tools often focus on visualizing the data transformation rather than the transformation language used to represent the data changes behind the scenes. In fact, most of the work in the data transformation language domain seems to be classified into one of two categories. Either the work focuses on trying to help users visualize the data transformations that have

occurred [6, 11, 7] or it focuses on modifying existing transformation languages to help with the data transformation process [5, 8]. Due to the lack of literature about the specifics of data transformation languages, we decided to cast our net further and examine data transformation language frameworks. There was a minimal amount of literature on using logical programming concepts to create a data transformation language [3]. The idea was that people find declarative languages like logical programming languages easier to understand as it focuses on the relationships between the data models rather than the method of how to locate a certain piece of information to work with [3, 12]. There has been research working with declarative data transformation languages for different data models, highlighting the flexibility requirements in order to keep current with today’s less structured data, [3, 9, 12] so we decided to try and model our data transformation language accordingly.

## 4.2 Supported Changes

Another part of the system that we had to figure out was the actual scenarios that would be executed on our system. We decided to do a literature search to try and figure out what our system had to support [4, 6, 7, 10, 11]. From this requirements solicitation, we determined that our data transformation language had to focus on the logical transformations rather than the physical (this is handled by the fact that our data transformation language borrows from logical programming concepts) and that the types of changes we had to support were map and reshape changes (see Section 3.1) [6].

## 4.3 View Maintenance

Due to our use of a remote data source that doesn’t know about update details made by the application, we began our view maintenance research in data warehousing environments. We considered the *Eager Compensating Algorithm* (ECA) [15] which described triggering events to update the warehouse or the application’s materialized view. While this approach may be helpful in correcting for inconsistencies arising from decoupling the data source and application, it requires quite a bit of overhead to regularly connect and creates the opportunity for anomalies (consistency issues that arise when updating views while base data is being changed) [15].

We implemented an application that takes a mixed-initiative approach to updating the remote data source making it important for us to be aware of how often to push any possible schema mapping/schema changes. This seems to be a comparable problem to how materialized views need to push its changes to a remote database [1, 2, 14]. Through looking at how views and indices are maintained in a large distributed database [1], we noted that the model suggested in the papers validated many of the design decisions we made for our transformation language (see Section 3.2). For example, the discussion on the different ways to push updates and how to handle update discrepancies (source versus the application) [1] is reflected in our implementation.

## 5 Future Work

Future extensions of this work can look into testing the transformation language on other data formats like XML to see if it can effectively capture the changes that happen in an XML document. We can also look into how to handle situations where there are multiple concurrent users drawing upon the same data source and what policies should be adopted into order to achieve the best balance between performance and data consistency. Updating the data source too often could cause performance overhead from locking up the data source while updating too infrequently could cause inconsistent data to be propagated to many different users. The amount of effort and operational overhead involved with many people making the same changes would be undesirable.

Another extension of the project could be investigating how the data transformations could be integrated into data provenance to help with automatic data conflict resolutions [13]. As the project currently stands, the remote application can push any type of data to the data source. If the data being pushed to the data source violates any table constraints (e.g., if a user tries to push a non-unique value into a column that has specified that it only takes unique values), the database administrator has to manually resolve that violation. It is possible to investigate whether or not provenance information can be leveraged to help make these decisions thus lowering the workload on the database administrator. A possible example of how to use the provenance information could be using the previous history of the sorts of changes have been accepted for that column or table to determine the likelihood of the current data violation being accepted. Provenance could also be used to determine the likelihood of correctness by the app or user who made the change; as information about who has changed values can be tracked, it is not inconceivable that the statistics of who has made the best or the most correct/acceptable changes can be used in some way.

This problem can also be investigated from an information visualization/HCI perspective. The best way to present information to the user to help facilitate use and understanding of how the data can be manipulated could be examined. However, as stated before, this is more on the application side of this problem which we are not specifically dealing with in the scope of this project.

The update policy of this system can be examined more closely to see if we can draw upon more of the principles and ideas discussed in the distributed database-application system PNUTS [1].

## References

- [1] D. Agrawal, A. El Abbadi, A. Singh, and T. Yurek. Efficient view maintenance at data warehouses. In *ACM SIGMOD Record*, volume 26, pages 417–427. ACM, 1997.
- [2] P. Agrawal, A. Silberstein, B. F. Cooper, U. Srivastava, and R. Ramakrishnan. Asynchronous view maintenance for vlcd databases. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 179–192. ACM, 2009.

- [3] F. Bry and S. Schaffert. Towards a declarative query and transformation language for xml and semistructured data: Simulation unification. In *Logic Programming*, pages 255–270. Springer, 2002.
- [4] H. Galhardas, D. Florescu, D. Shasha, E. Simon, and C. Saita. Declarative Data Cleaning : Language, Model, and Algorithms. Research Report RR-4149, 2001. Projet CARAVEL.
- [5] R. Goldman, J. McHugh, and J. Widom. From semistructured data to xml: Migrating the lore data model and query language. In *ACM SIGMOD Workshop on The Web and Databases ( WebDB 1999)*, 1999.
- [6] S. Kandel, A. Paepcke, J. Hellerstein, and J. Heer. Wrangler: Interactive visual specification of data transformation scripts. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 3363–3372. ACM, 2011.
- [7] S. Kandel, R. Parikh, A. Paepcke, J. M. Hellerstein, and J. Heer. Profiler: Integrated statistical analysis and visualization for data quality assessment. In *Proceedings of the International Working Conference on Advanced Visual Interfaces*, pages 547–554. ACM, 2012.
- [8] L. V. Lakshmanan, F. Sadri, and S. N. Subramanian. Schemasql: An extension to sql for multidatabase interoperability. *ACM Transactions on Database Systems (TODS)*, 26(4):476–519, 2001.
- [9] M. Lawley and J. Steel. Practical declarative model transformation with tefkat. In *Satellite Events at the MoDELS 2005 Conference*, pages 139–150. Springer, 2006.
- [10] E. Rahm and H. H. Do. Data cleaning: Problems and current approaches. *IEEE Data Eng. Bull.*, 23(4):3–13, 2000.
- [11] V. Raman and J. M. Hellerstein. Potter’s wheel: An interactive data cleaning system. In *VLDB*, volume 1, pages 381–390, 2001.
- [12] P. Tarau. An embedded declarative data transformation language. In *Proceedings of the 11th ACM SIGPLAN conference on Principles and practice of declarative programming*, pages 171–182. ACM, 2009.
- [13] A. M. Thrastarson. Managing updates and transformations in data sharing systems (m.sc thesis), 2014.
- [14] J. Zhou, P.-A. Larson, and H. G. Elmongui. Lazy maintenance of materialized views. In *Proceedings of the 33rd international conference on Very large data bases*, pages 231–242. VLDB Endowment, 2007.
- [15] Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom. View maintenance in a warehousing environment. *ACM SIGMOD Record*, 24(2):316–327, 1995.