DataDiff: User-Interpretable Data Transformation Summaries for Collaborative Data Analysis

Gunce Su Yilmaz
University of Chicago
Chicago, Illinois

Tana Wattanawaroon
University of Illinois
Urbana-Champaign, Illinois

Liqi Xu
University of Illinois
Urbana-Champaign, Illinois

Abhishek Nigam
University of Illinois
Urbana-Champaign, Illinois

Aaron J. Elmore
University of Chicago
Chicago, Illinois

Aditya Parameswaran
University of Illinois
Urbana-Champaign, Illinois

ABSTRACT

Interest in collaborative dataset versioning has emerged due to complex, ad-hoc, and collaborative nature of data science, and the need to record and reason about data at various stages of pre-processing, cleaning, and analysis. To support effective collaborative dataset versioning, one critical operation is *differentiation*: to succinctly describe what has changed from one dataset to the next. Differentiation, or diffing, allows users to understand changes between two versions, to better understand the evolution process, or to support effective merging or conflict detection across versions. We demonstrate DataDiff, a practical and concise data-diff tool that provides human-interpretable explanations of changes between datasets without reliance on the operations that led to the changes.

CCS CONCEPTS

• **Information systems → Data cleaning**;

KEYWORDS

versioning, differentiation

ACM Reference Format:
Gunce Su Yilmaz, Tana Wattanawaroon, Liqi Xu, Abhishek Nigam, Aaron J. Elmore, and Aditya Parameswaran. 2018. DataDiff: User-Interpretable Data Transformation Summaries for Collaborative Data Analysis. In *SIGMOD'18: 2018 International Conference on Management of Data, June 10–15, 2018, Houston, TX, USA.* ACM, New York, NY, USA, Article 4, 5 pages. https://doi.org/10.1145/3183713.3193564

1 INTRODUCTION

Due to an increasing demand for mechanisms for managing collaborative data preparation, data analytics, and data science, recent systems propose dataset versioning as a first-class primitive to support concurrent long-term isolated data science sessions for users to clean, wrangle, and integrate datasets [9, 13, 14]. Similar to versioning found in source-code version control systems (i.e. git, mercurial), dataset versioning allows users to create an isolated logical copy of the dataset, called a *branch*, that they can explicitly later share back into the system, or update their branch to reflect changes made in other branches. A branch is made up of one or more *commits*, each of which is a collection of operations—much like a long-lived transaction. Unlike temporal databases [7, 15] or snapshot isolation, dataset versioning allows commits to be created by either modifying a previous commit, or by *merging* multiple commits into a single new commit.

For users to understand the changes made in a branch and to merge branches, an efficient differentiation, or *diff* operation is critical. For a diff operation, source-code version control systems simply list out the lines that have changed between versions. Then, while merging, these version control systems utilize the results of the diff operation and automate the merge wherever possible, while also listing out line-level conflicts where two parent commits both have modified the same line of code. If any conflict is found, the user performing the merge must manually reconcile those conflicts by selectively merging a file that has annotated the areas of conflict.

While such a diff operation and manual conflict resolution may work for source-code version control, such an approach is not tenable for dataset versioning for the following reasons. First, dataset operations often result in a high number of modifications, and thus an increased number of conflicts as compared to source-code version control systems. Single line commands, such as updates or deletes with non-selective predicates or alter table statements, can modify an entire table, and cannot be succinctly described as line-by-line edits. Such broad modifications are common in data analysis for tasks such as normalizing fields, splitting attributes, interpolating values, and projecting down a dataset. Second, the intention or impact of a change can often be inferred with source code as the developer can interpret the logic of the change. With dataset versioning this intention may not always be present, as the user may only see the modified records and not necessarily the updates that created the modification. This is due to some dataset versioning systems supporting UDFs or via updates outside of the system. Third, the translation of line-level conflicts or diffs equates to record level conflicts for dataset versioning. Since modifications and reads can occur at the attribute level, this granularity of conflicts may be too coarse for both detecting conflicts and helping users understanding where conflicts exist. Lastly, and outside the scope of this demo, the definitions of a conflict with dataset versioning can expand beyond write-write conflicts.

Therefore to support the effective branching, an efficient *data diff* tool must exist to aid the user in understanding how two versions

SIGMOD'18, June 10–15, 2018, Houston, TX, USA

ACM ISBN 978-1-4503-4703-7/18/06…$15.00
https://doi.org/10.1145/3183713.3193564

differ from each other by examining the data only. Such a tool is useful for understanding how versions evolve and for understanding conflicts between versions. To understand how data diff can be implemented, one must understand how updates can be applied to dataset versioning systems. Some dataset versioning systems, such as Decibel [18], only allow for modifications to happen through the system DSL or DML. Here, provenance could be used to help track the operations that lead to a change. However, other systems such as OrpheusDB [14] allow for users to export a dataset to a shared format (i.e. CSV) and reimport the dataset into a child commit. This allows for external tools (i.e. Pandas) to modify the dataset and result in diff tools only to rely on data-only based solutions. In this demo, we propose DataDiff, a system to support concise diffing between versions without access to the underlying operations. We leverage recent development of theory [5] to develop the underlying algorithms, while also developing a user interface with the goal of providing concise explanations of complex changes efficiently.

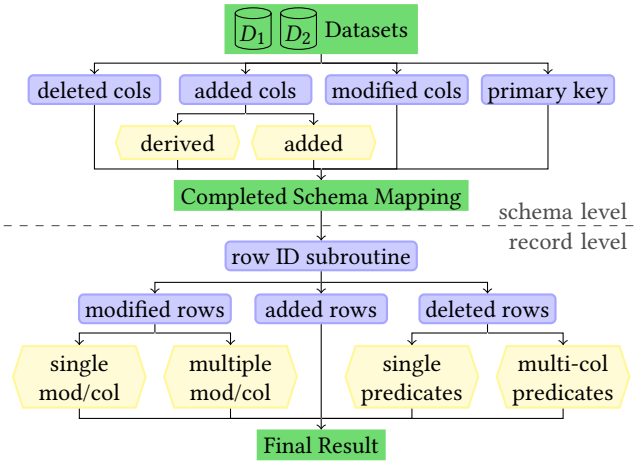

Figure 1: DataDiff **Workflow**

2 BACKGROUND

While a large body of related work exists, we limit our discussion here to versioning for databases and synthesizing view definitions.

Data Versioning. Temporal databases support versioning operations only for a linear chain of versions [17, 21], as opposed to branching and merging that are more crucial in collaborative data science. DB2 [22] and Teradata [8] supports time-travel functionality by developing temporal capabilities on top of traditional databases. Other research [17, 21] focuses on branched temporal databases, allowing multiple linear chains of versions, but without supporting arbitrary branching and merging. Jiang et al. [16] describe a new index for tree-oriented versioned data without merges.

`git` and `svn` are version control systems designed for source code; thus they cannot scale to large datasets and have limited querying capabilities [18]. Noms [4] extends `git` version control functions and supports decentralized version control over large structured datasets. LiquiBase [3] manages, keeps track of and restores database schema changes. DBV [2] logs the schema change operations made in one branch and is able to reapply these operations to datasets in other branches.

Related projects explore versioning as a first class primitive [9, 11, 13, 18], allowing the management of branched datasets, while supporting querying across versions and data, but none of these systems provide any rich functionality for explaining diffs between versions other than showing users the records that are in conflict, and do not provide any support for complex conflict resolution.

View Synthesis. DataDiff is related to *view synthesis*, wherein the goal is to find a *view definition* that succinctly describes one database instance as a view of another instance [12, 23]. Recent work has also extended view definitions to a broader class of queries and interfaces [6, 10, 19, 24]. The key difference between DataDiff and view synthesis is that since DataDiff allows modification of data, which is non-commutative even for simple modifications, the problem becomes NP-Hard even for a finite number of attributes [5].

3 THE DATA-DIFF PROBLEM AND SOLUTION

DataDiff takes as input two datasets, the *origin dataset* and the *destination dataset*, and provides a succinct description of the changes between the two datasets in the form of a sequence of operations that be used to transform the origin dataset into the destination dataset. Operations can affect:

- the schema: columns added, removed, or renamed, and
- the data: tuples added, removed, or has values changed.

We focus on SQL-style changes for the purpose of displaying the results to the user. (Of course, the changes made to the dataset could be made within a database or outside of it, and therefore need not conform to an SQL query, as we described in Section 1.) The schema may be modified with `ALTER TABLE` commands. Adding a tuple can be done with `INSERT`, whereas deleting tuples or modifying values can be done in batches using `DELETE` and `UPDATE`, with the change affecting all tuples that match the specified `WHERE` conditional clause. Hence, it is possible that a change affecting the majority of tuples is described succinctly using only one command.

Like present dataset versioning tools such as OrpheusDB [14] (which we build on) and Decibel [18], we focus on structured, relational datasets. We also assume that the datasets share a primary key. Without a primary key, it can be difficult to determine whether a new tuple in the destination dataset is entirely new or actually modified from another tuple in the origin dataset.

Complexity Results. In a companion paper, we formally define the problem of identifying a succinct data modification script, determining the complexity and tractability of the problem under different settings of attribute characteristics (read, write), possible `WHERE` conditions (equality, at-most, range), and update modifiers (assignment, increment, etc.) [5]. Even when restricted to single-attribute conditions, this problem is intractable. For example, finding the shortest sequence of updates with range conditions and increment modifiers is NP-Hard. When the condition spans multiple attributes, the problem is hard even for conditions as simple as equality. It can be shown that the problem is a generalization of the view synthesis problem (Section 2), and is therefore harder. The difficulty arises from the need to break down compound operations, and sometimes the non-commutativity of the operations.

Note, that the DataDiff problem is much more general than the problem of finding a succinct data modification script—which tries to characterize how a specific set of tuples have been modified,

when there is no change to the schema. In addition to tuple modifications, in the DataDiff problem, tuples could be deleted—for which the work on view synthesis as described in Section 2 provides partial solutions. Finally, DataDiff could also have attributes being modified, added, deleted, or renamed, and tuples being added. Due to all of these reasons, DataDiff is a substantially harder problem, and developing a system for DataDiff that provides reasonable results in interactive time is a challenge.

System Description. We implement DataDiff as a module in OrpheusDB [14], a "bolt-on" versioning layer for PostgreSQL. OrpheusDB supports versioning, by creating mappings of versions to tuples through non-atomic data types. Versions are materialized to a table or CSV from partitioned relations that contain records from one or more versions. Versioning metadata is stored in auxiliary data structures. When a user invokes a diff in OrpheusDB, the system calls the DataDiff submodule to derive a concise diff explanation. Figure 1 highlights the workflow to determine both the schema level and the data level changes. This workflow output also matches the diff UI demonstrated in Section 4. The list below summarizes the changes that DataDiff currently supports:

1. Detects column differences between datasets and prompts the user to confirm differences. Given an origin dataset with known primary key column and a destination dataset, DataDiff predicts the column mappings from origin to destination, and the primary key column of the destination. It designates a null-to-column mapping for added columns and a column-to-null mapping for removed columns. DataDiff also provides a clear interface for the user to evaluate and modify these mappings.

2. Detects deleted rows via selection queries. Once the column mappings are identified, DataDiff adds selection queries examining one column predicates, two column predicates, and three column predicates (all excluding the primary key), as long as they provide no false-positives, until all of the tuples that need to be deleted are indeed deleted. In the worst case, DataDiff will use the primary key to delete the remaining tuples that need to be deleted but were not. In general, this problem is related to the view synthesis problem (see Section 2) and is therefore intractable.

3. Detects added rows. DataDiff simply adds the rows that were inserted in the destination, in one large `INSERT` statement.

4. Detects modifications to a single column where one modification was performed on all the modified rows. DataDiff first assumes a single modification was made to a column and uses a random search algorithm to perform a regression between the values of two versions of a column. If the assignment is successful, it adds the change to the set of modifications.

5. Detects modifications to a single column where a split point was used to perform one modification to rows with a value less than the split point, and another modification was performed on the rest of the rows. If the regression assignment fails with the assumption of single modification, DataDiff then tries to find a split point in the origin column and runs the same regression procedure twice: one for between the values less than the split point in the origin column and their destination counterparts, and one for between the values greater than the split point in the origin and the corresponding destination entries.

6. Provides a concise explanation of the data level changes. DataDiff returns two sets of results, one, a list of queries that takes the origin dataset as input and outputs the destination, and two, the list of tuples that were modified, added, removed, or updated. The first set of results can help users track the logical changes between the versions while the second set, the list of changed records, can help track the merge conflicts from diffing sibling versions.

4 DEMONSTRATION DESCRIPTION

We have built an interface that allows users to interpret the differences between dataset versions in an interactive manner. We now describe this interface as well as the demonstration experience.

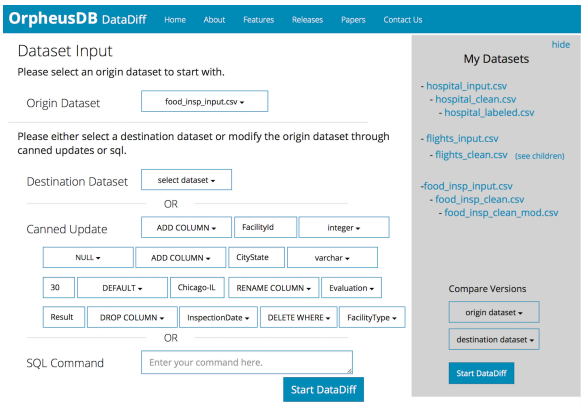

Figure 2: DataDiff Dataset Modification

In addition to the critical and sometimes problematic merge operations between two children of the same parent, dataset versioning between a parent and its child is also very common. To address both cases, DataDiff takes an "origin dataset" and a "destination dataset" and reports back the most succinct summarization (e.g. the shortest list of queries) that allows us to derive the destination dataset from the origin dataset. The "Dataset Input" component of our interface, shown in Figure 2, is the first step of the demo process, consisting of forms to upload the origin and the destination datasets. To make the demo experience flexible yet succinct, we provide two extra modification input methods to be used instead of the destination dataset input: parameterized (e.g. canned) updates and SQL. This interface will create a destination dataset, store it, and allow use of this new version in the future DataDiff operations. To store and reuse these dataset versions, the interface provides the users with a local repository called "My Datasets". We will have prepared source and destination datasets for shorter demonstrations, including cleaned and dirty versions of the City of Chicago Food Inspection dataset [20].

To run DataDiff, the user inputs the origin and the destination datasets. DataDiff will then direct the user to the "Confirm Column Mappings" page, as depicted in Figure 3. This shows a summary of the added, removed, the modified column names of the origin dataset and a mapping table to confirm the column mappings from the origin to the destination dataset. Each row in the table is a tuple that contains a column name from the origin dataset, its mapping in the destination dataset, and a boolean value indicating whether the mapped column is the primary key in the destination dataset. A tuple with a non-null first entry and a non-null second

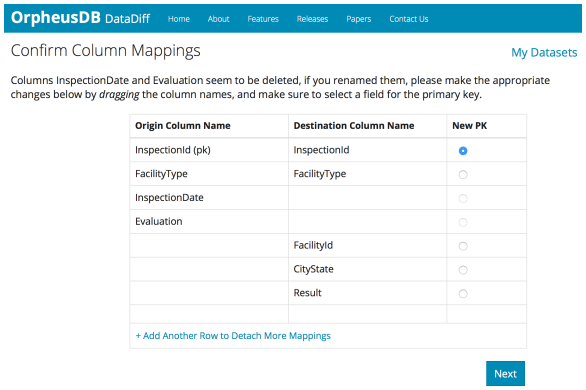

Figure 3: DataDiff **Column Mappings**

entry indicates an unchanged or renamed column in the destination dataset. A tuple with the first two entries null and non-null indicates an added column, and a tuple with first two entries non-null and null indicates a deleted column in the destination dataset. DataDiff will predict the mapped columns and prompt the user to fix any mappings by dragging the column names in the second column of the mapping table to the appropriate place. If a user wants to detach a predicted mapping, they can add an empty tuple to the mapping table and then use the dragging feature accordingly. Although we ask for manual input in this step, we present a simple interface for diffing schema modifications.

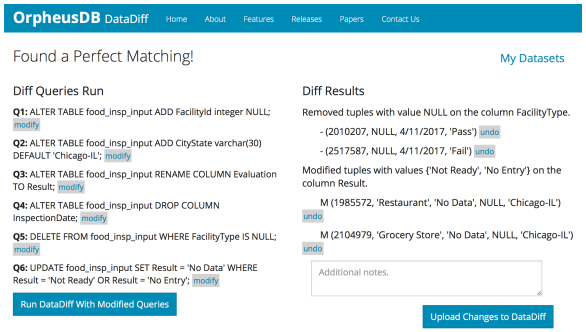

Figure 4: DataDiff **Output**

When the schema checking is completed, DataDiff will finally compare the versions and output two sets of results, depicted in Figure 4. The first set lists the SQL queries DataDiff has applied to the origin dataset in order to get the destination dataset. If a user wants to store their own version of the query list, for a more efficient or more human-readable version, they can modify each query and test if the modified queries applied to the origin dataset will perfectly match the destination dataset. The second set of results is a sample of tuple-based representation from the entire diff between the two versions—displaying the added, removed, and the modified tuples in the origin dataset.

To enable merge, starting from the result of the diff, DataDiff will also allow the user to undo operations on certain tuples by selecting examples listed in the diff results. This allows users to generate a new dataset with some of the operations between origin and destination undone. If the user wants to persist a SQL schema for the destination dataset, there is an optional step "Finalize Schema", displayed in Figure 5. DataDiff will check if the destination dataset for the merge is in compliance with the data types and constraints that the user has provided.

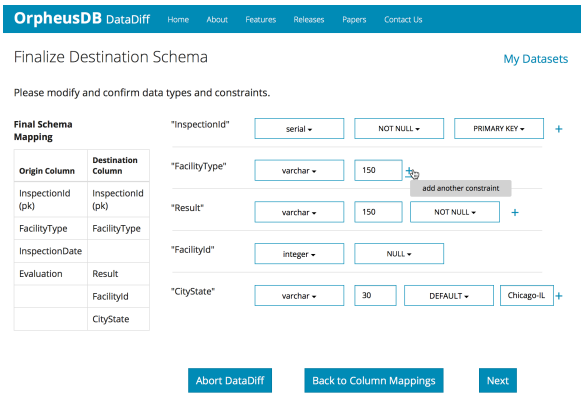

Figure 5: DataDiff **Final Destination Schema**

This demonstration experience will highlight how DataDiff enables easy conflict resolution, provides concise diff explanations, and illustrates the challenges in building an efficient data-diffing tool in the presence of arbitrary updates.

REFERENCES

[1] [n. d.]. Dat. http://datproject.org/. ([n. d.]).
[2] [n. d.]. dbv. https://dbv.vizuina.com/. ([n. d.]).
[3] [n. d.]. Liquibase. http://www.liquibase.org/. ([n. d.]).
[4] [n. d.]. Noms. https://github.com/attic-labs/noms. ([n. d.]).
[5] Available at: http://data-people.cs.illinois.edu/papers/datadiff.pdf. Towards a Theory of DATA-DIFF : Optimal Synthesis of Succinct Data Modification Script. In *Technical Report*.
[6] Azza Abouzied et al. 2013. Learning and verifying quantified boolean queries by example. In *PODS*. ACM, 49–60.
[7] Ilsoo Ahn et al. 1986. Performance evaluation of a temporal database management system. In *ACM SIGMOD Record*, Vol. 15. ACM, 96–107.
[8] Mohammed Al-Kateb et al. [n. d.]. Temporal query processing in Teradata. In *EDBT'13*. ACM, 573–578.
[9] Anant Bhardwaj, Souvik Bhattacherjee, Amit Chavan, Amol Deshpande, Aaron J Elmore, Samuel Madden, and Aditya G Parameswaran. 2015. Datahub: Collaborative data science & dataset version management at scale. *CIDR* (2015).
[10] Angela Bonifati et al. 2016. Learning join queries from user examples. *TODS* 40, 4 (2016), 24.
[11] Amit Chavan and Amol Deshpande. 2017. DEX: Query Execution in a Delta-based Storage System. In *SIGMOD*. ACM, 171–186.
[12] Anish Das Sarma et al. 2010. Synthesizing view definitions from data. In *ICDT*. ACM, 89–103.
[13] Joseph M Hellerstein et al. 2017. Ground: A Data Context Service.. In *CIDR*.
[14] Silu Huang, Liqi Xu, Jialin Liu, Aaron J. Elmore, and Aditya G. Parameswaran. 2017. OrpheusDB: Bolt-on Versioning for Relational Databases. *PVLDB* (2017).
[15] Christian S Jensen and Richard T Snodgrass. 1999. Temporal data management. *IEEE Transactions on Knowledge and Data Engineering* 11, 1 (1999), 36–44.
[16] Linan Jiang, Betty Salzberg, David B Lomet, and Manuel Barrena García. 2000. The BT-tree: A Branched and Temporal Access Method.. In *VLDB*. 451–460.
[17] Gad M Landau et al. 1995. Historical queries along multiple lines of time evolution. *The VLDB Journal* 4, 4 (1995), 703–726.
[18] Michael Maddox, David Goehring, Aaron J Elmore, Samuel Madden, Aditya Parameswaran, and Amol Deshpande. 2016. Decibel: The relational dataset branching system. *VLDB* 9, 9 (2016), 624–635.
[19] Kiril Panev and Sebastian Michel. 2016. Reverse Engineering Top-k Database Queries with PALEO.. In *EDBT*. 113–124.
[20] Theodoros Rekatsinas et al. 2017. HoloClean: Holistic Data Repairs with Probabilistic Inference. *PVLDB* 10, 11 (2017), 1190–1201.
[21] Betty Joan Salzberg and David B Lomet. 1995. *Branched and Temporal Index Structures*. College of Computer Science, Northeastern University.
[22] Cynthia M Saracco, Matthias Nicola, and Lenisha Gandhi. 2010. A matter of time: Temporal data management in DB2 for z/OS. *IBM Corporation, New York* (2010).
[23] Q. T. Tran, C. Y. Chan, and S. Parthasarathy. 2009. Query by Output.
[24] Meihui Zhang et al. 2013. Reverse engineering complex join queries. In *SIGMOD*. ACM, 809–820.