The Exception that Improves the Rule*

Juliana Freire[n], Boris Glavic[i], Oliver Kennedy[b], Heiko Mueller[n]
[n]: New York University; {juliana.freire, heiko.mueller}@nyu.edu
[i]: Illinois Institute of Technology; bglavic@iit.edu
[b]: University at Buffalo; okennedy@buffalo.edu

ABSTRACT

The database community has developed a plethora of tools and techniques for data curation and exploration, from declarative languages, to specialized techniques for data repair, and more. Yet, there is currently no consensus on how to best expose these powerful tools to an analyst in a simple, intuitive, and above all, flexible way. Thus, analysts continue to rely on tools such as spreadsheets, imperative languages, and notebook style programming environments like Jupyter for data curation. In this work, we explore the integration of spreadsheets, notebooks, and relational databases. We focus on a key advantage that both spreadsheets and imperative notebook environments have over classical relational databases: ease of exception. By relying on set-at-a-time operations, relational databases sacrifice the ability to easily define singleton operations, exceptions to a normal data processing workflow that affect query processing for a fixed set of explicitly targeted records. In comparison, a spreadsheet user can easily change the formula for just one cell, while a notebook user can add an imperative operation to her notebook that alters an output "view". We believe that enabling such idiosyncratic manual transformations in a classical relational database is critical for curation, as curation operations that are easy to declare for individual values can often be extremely challenging to generalize. We explore the challenges of enabling singletons in relational databases, propose a hybrid spreadsheet/relational notebook environment for data curation, and present our vision of Vizier, a system that exposes data curation through such an interface.

1. INTRODUCTION

In spite of the availability of powerful automated curation, cleaning, and analysis tools, spreadsheets and notebook UIs (e.g., Jupyter/iPython) are still the predominant tools used by most data scientists. Their ubiquity can be attributed in part to the fact that typical users prefer a known over an unknown interface — even if the unknown interface may be better suited for the task at hand [6]. However, we argue that while both interfaces lack the scalability and power of relational databases, they offer several compelling benefits for curation workloads. Based on this observation we propose a combined spreadsheet and notebook UI for data curation over relational data that combines the best of spreadsheets, notebooks, and relational DBMS. This UI will support functionality not commonly found in either spreadsheets or notebooks, including automated curation operators [21], deployment of curation workflows over large datasets [12], declarative queries [2, 15], and support for exploratory curation tasks [17]. We first review the spreadsheet and notebook UI paradigms and then evaluate how they may be combined into a single coherent interface: a system that we call Vizier. This hybrid UI enables powerful relational queries, while still being flexible enough to permit easy data manipulation, summarization, and visualization.

Spreadsheets. Spreadsheets are a ubiquitous data processing tool. Their simplicity, generality, and adaptability make them ideal for "playing" with data through predominantly visual programming metaphors. Spreadsheets provide several important features that are useful during data curation:
– *Convenient modification of values and computations:* The user can update any cell's value or formula directly from the user interface. This enables manual curation operations like resolving missing values and correcting data errors.
– *Manual operations with inlined results:* By using formulas in cells, a user-defined computation and its result are shown together with its input data.
– *Visual mapping over data collections:* Most spreadsheet systems enable the user to take a formula (computation) and map it to a range of cells through *position-relative* references in cell formulas. For example, this can be done by copy/paste or by a fill operation. We refer to this mechanism as *adapt&apply*. This approach to bulk, set-at-a-time functionality is very useful in data curation: A fix to repair one piece of data (e.g., conversion between units) can be deployed over the whole dataset. Importantly, the formulas applied in this fashion are independent of each other creating an affordance for declaring exceptions to the bulk rule by modifying individual formulas.

Indeed, many curation applications require users to "break the rules" and apply one-off modifications or transformations to individual fields or records. For example, (1) hypothetical what-if scenarios require users to apply small ad-hoc updates to adjust the inputs under test; (2) repairs for data errors may be easy to define for individual cases, but far harder to define in a general case; (3) complex data transformations that need to be generalized would still be easier to define

*The authors are listed in alphabetical order.

for individual test cases than in bulk. By making it easy for users to break the rules, even if only temporarily, spreadsheets empower users to explore data, evaluate options, and better understand the effects of their curation efforts. Such violations, or *singleton* operations, are not handled gracefully by existing relational DBMS. However spreadsheets also have several drawbacks compared to a DBMS:
– *Non-Adaptive Computations over Collections:* While spreadsheets allow a computation defined for a specific set of cells to be applied to a larger collection of cells, the two dominant systems, i.e., Microsoft Excel and Google Sheets, do not support automatically extending formulas to new cells as data is added [1]. This is in stark contrast to relational databases with their declarative, data-independent query languages.
– *Collection operation intent is not explicit:* Adapt&apply allows a computation to be mapped over a collection, but there is no visual cue that indicates that a set of cells are storing formulas which were mapped in this way. That is, there is no generalization/abstraction mechanism in spreadsheets to represent higher-level bulk operations such as view queries in a database.
– *No order among operations or workflow branch tracking:* Spreadsheets use cell highlighting as a visual metaphor for dependency tracking. These visualizations are specific to single cells and do not lend themselves to tracking large curation workflows. Furthermore, these visualizations are limited to tracing one dependency at a time, making tracking transitive dependencies cumbersome.
– *Unintuitive results for adapt&apply:* As we will discuss further in Section 3, adapt&apply functionality as implemented in many systems can lead to unexpected results.

Notebook-style UIs. Systems like Jupyter expose an interactive, interpreted programming environment through a notebook-like interface where the user mixes documentation (text) with code. The output for code blocks is shown directly in the notebook — a feature that is widely used to produce data visualizations.
– *Inline documentation:* Notebooks allow users to integrate comments, descriptive text, notes, and formatting details, making it easier for others to retrace their steps.
– *Incremental development of complex workflows:* Notebooks allow users to incrementally build curation workflows, one page at a time. The (always linear) structure of the workflow is made explicit through the notebook interface.

However, some operations that are supported well in spreadsheets are harder to express in notebooks, and some disadvantages are shared among both paradigms:
– *Small edits are cumbersome:* Compared to spreadsheets, modifying individual data values requires users to write code.
– *Linear workflows and no backtracking:* Notebooks are inherently linear and do not allow users to backtrack or branch their development efforts.
– *All-at-once processing:* Both spreadsheets and notebooks operate over datasets in their entirety, something that is not feasible for large giga-/tera-byte files.

Visualizations. Both spreadsheets and notebook UIs make it very easy for users to create visualizations from data on the fly and show these visualization inline with the data. Also both paradigms allow these visualization to be tweaked and to be refreshed based on changes to their inputs. Spreadsheets in particular provide a very easy-to-use interface for selecting what data should be visualized.

Combining Spreadsheets and Notebooks. Spreadsheets permit intuitive visual interactions with data, while notebooks provide a clearer expression of the user's intent that can actually be reproduced. We propose a hybrid UI that combines elements of both interfaces, augmenting them with capabilities common to relational data processing. We discuss the challenges of developing such an integrated interface and how it facilitates data curation and exploration. We also introduce our proposed system, called Vizier, which empowers users with spreadsheet-like flexibility for transforming, visualizing, and exploring relational data, while still retaining the expressiveness and workflow capabilities of a notebook. At the heart of our approach is support for singleton operations in a relational setting, which in turn enables a bi-directional mapping between a spreadsheet-style graphical interface, and a notebook-style programmatic interface.

To enable singleton transformations within the framework of a classical relational database, we extend the notebook programming model with support for *interactive views*. An interactive view begins its life as a classical view, presented to the user in tabular form. In contrast to a classical view however, an interactive view can be edited much like a spreadsheet. Note that such updates are not propagated back to the view's inputs, but are treated as updates to the view definition. Users can modify fields, add new rows and columns, use a spreadsheet-style equation editor to define derived values, and more. As the user edits the view, the user's actions are seamlessly transformed into a program of relational(-ish) data transformation operators that derive the new, edited view. This program serves as a form of history, allowing the user to revisit and revise earlier edits, even out of order. Furthermore, the program defines a workflow, albeit one highly specialized to a specific dataset. Even this is sufficient to provide classical benefits of workflow provenance such as auditability and explainability for derived data. Once an interactive view is developed for one dataset, it can more readily be adapted to new data or to react to changes in its inputs. Recasting the user's actions programmatically allows us to leverage existing work on algebraic equivalences [13] and program rewriting [2,3] to first obtain multiple interpretations of sequences of user actions, and then to extrapolate more general expressions of the user's intent [9, 22].

An important challenge that we consider is how to minimize unexpected side effects. Unlike a relational database where query semantics are explicit, visual interactions in a spreadsheet trigger many implicit behaviors. Existing spreadsheet software is carefully designed to ensure that these behaviors follow a consistent, intuitive pattern. Many of these implicit behaviors are a consequence of the spreadsheet's coordinate system: User actions that move cells frequently trigger implicit secondary updates to formulas and other cells.

Contributions. Our core contributions are as follows: (1) We present Vizier, a hybrid relational notebook/spreadsheet exploration-based data curation framework and outline its capabilities, (2) We discuss the challenges of mapping actions back and forth between the different components, (3) We define a data model for Vizier that allows us to precisely characterize the side-effects of user's actions on a spreadsheet. (4) We apply this model through a case study on existing spreadsheet software, and show how user actions in spreadsheet software follow a specific heuristic that we be-

[1] We note that Apple Numbers does exhibit desired behavior.

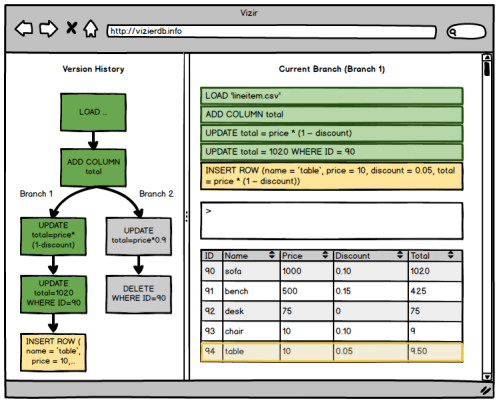

Figure 1: An example of Vizier's UI

```
LOAD 'lineitem.csv'
ADD COLUMN total;
UPDATE total = price * (1 - discount)
UPDATE total = 1020 WHERE ID = 90;
INSERT ROW ( name = 'table', price = 10,
             discount = 0.05, total = 9.5 )
```

Figure 2: An example VizUAL script

lieve captures the principle of least surprise. (5) We outline future research directions, including a readability-enhancing optimizer and generalization of singleton-based workflows.

2. INTERFACE DESIGN

Figure 1 illustrates the interface for Vizier, our proposed tool for data curation and exploration. This interface combines elements of both notebooks and spreadsheets.

2.1 The Notebook UI

Notebook interfaces like Jupyter's use an analogy of pages in a notebook that consist of a block of code and an output for the block, e.g., a table, visualization, or documentation. Blocks are part of a continuous program, allowing a user to quickly probe intermediate states by creating new visualizations or views of the data, or to safely insert hypothetical, exploratory modifications by adding or disabling pages.

Each page in a Vizier notebook can be thought of as a block of SQL DML/DDL code that imperatively manipulates a single relation, which is displayed as a table or visualization. Pages are evaluated in sequential order. Code defining later pages may reference preceding pages as if they were (materialized) views and edits to a page may result in cascading changes to pages that depend on it. We refer to this SQL-based language as the Vizier user action language (VizUAL). In spite of its imperative flavor, operators in VizUAL form a monad that can be compiled down to a generalized form of relational algebra [2, 3]. Due to space constraints, we will only sketch the language through examples in this paper; a full description is left to future work.

Figure 2 shows an example VizUAL script that loads a CSV file, extends it with a new column named `total`, defines a value for the column (derived from the remaining attributes), and applies two minor *singleton* edits to the result (a single value update and a row pasted into the result). VizUAL operates over ordered relations, and its language primitives are based on SQL's DDL and DML. As a result, while operations in VizUAL appear imperative, they actually define a sequence of declarative transformations on the data imported by the `LOAD` operation in the first line. The entire script can be rewritten into a SQL query:

```
SELECT *, total = CASE WHEN ID = 90 THEN 1020
                       ELSE price*(1-discount) END
FROM LOAD(lineitem.csv)
UNION ALL
SELECT name = 'table', price = 10,
       discount = 0.05, total = 9.5
```

Imperative-flavored declarative language syntax has been repeatedly found to be more user-friendly than classic declarative syntax [15, 18]. Here however, it also serves to highlight the compositional nature of interactive views: each user action that changes the view's schema or contents is reflected in the script by a new statement appended to its end. Thus, we aim for — in principle at least — a bi-directional mapping between user actions and statements in VizUAL.

In addition to enabling singletons and being easy to integrate with spreadsheets, the imperative flavor of VizUAL also enables a form of backtracking and branching. As illustrated in Figure 1, users can quickly try out hypothetical changes by checkpointing program state and applying a variant sequence of edits. Vizier will support a comprehensive suite of branching and merging capabilities for both data [14] and workflows [17].

2.2 The Spreadsheet UI

As a user edits tables and visualizations directly, these edits are reflected in the page where the table resides and they are also propagated to subsequent pages that depend on it. The user's edits, whether applied via the spreadsheet or notebook UI, are recorded as a form of workflow provenance [1, 7, 8, 17]. Note that our goal is not to reproduce the full interface of a spreadsheet, but rather to replicate as many of the flexible data and schema manipulation features as possible within a more structured framework. Concretely, Vizier's UI allows users to:

– *Overwrite arbitrary cells with constants, formulas, or regular expressions:* Users may click on any cell in the output to overwrite its contents (as in a spreadsheet).
– *Cast cells to a new type:* Dropdown menus allow the user to apply general transformations like typecasting. The bulk transformation is applied to all cells in a selected region.
– *Copy/Paste cells:* Users can copy and paste regions of cells. The formula of the copied cell(s) is replicated in the target region through adapt&apply. If the target region is larger than the source, cells in the source region are tiled to scale over the entire target region.
– *Add/Delete/Reorder columns or rows:* Users may drag columns or rows to reposition them. A tab at the bottom and right edges of the displayed table allows users to widen or lengthen the table, adding new columns or rows respectively. Other interface elements allow users to insert rows (resp., columns) before or after any existing row (column).
– *Sort data:* A dropdown menu allows users to sort data according to values in one or more columns.
– *Filter data:* A dropdown menu allows users to filter out rows according to a formula defined over the row.

Many of these operations (e.g., paste, typecast) require the user to define a target, normally specified as rectangular area selected by clicking and dragging with the cursor; We also propose to support declarative regions, as discussed below.

2.3 Spreadsheet to Notebook and Back

To create a seamless interface between the spreadsheet

and notebook UIs, we need to map operational semantics and effects between the two interaction models. We now sketch solutions to several of the resulting challenges.

Identifying Singletons. To allow singleton operations, VizUAL must be able to uniquely identify specific rows and columns of the dataset, including rows and columns introduced in the code itself. More importantly, these identifying markers must persist through the program: A user edit applied to the row 10 of `'lineitem.csv'` must continue to be applied to the 10th row, even if an insert operation occurs between rows 8 and 9. We address this challenge through provenance: Each operation that creates rows generates a unique tag for each row, column, and cell, which persists through the lifetime of the row, column, or cell.

Positional vs Qualitative Semantics. Spreadsheets allow formulas to reference other cells by relative position. For example, a cell's formula might compute a cumulative average over all rows up to that point. To capture these semantics, bulk update operations must permit a form of implicit windowing, semantics that can be unintuitive if handled incorrectly. We address positional semantics as part of VizUAL's data model.

Readability. Interactive spreadsheet interfaces encourage many small transformations. In contrast, code promotes abstraction and terse expressions that precisely convey the user's intent. As a result, directly translating visually generated operations into code is likely to produce a large, hard-to-follow, unreadable mess. We address this by proposing a source-to-source readability-optimizing compiler.

Formula Extraction. An important challenge arises in the reverse direction as well. When a user clicks on a formula to edit it, we need to reconstruct the formula that derived the cell's value. However, obtaining the precise formula may not be as simple as tracing the provenance of the cell's value, since operations (e.g., reordering rows) may alter dependencies. We address this specific issue as part of our data model.

3. THE VizUAL DATA MODEL

The fundamental unit of data in VizUAL is a *cell*, a 3-tuple: $C = \langle\ id, f, v\ \rangle$, consisting of a globally unique identifier id, a formula expression f, and a value v. The identifier of a cell is assigned to it when it is first accessed and is immutable — even if the cell is moved to a different position in the spreadsheet. By storing both a formula f and its result v, a cell maintains data provenance akin to a provenance-aware data management system, where each record is associated with metadata describing how it was computed. Here, this metadata serves two purposes. First, as noted above, we need to be able to reliably materialize the formula backing each cell so that it can be edited. We need to ensure that each operator defines precise semantics for how it affects formulas. Second, and perhaps more importantly, we track both values and the formula used to derive them as a way to define operational semantics that minimize user surprise. As we discuss shortly, one specific update to a spreadsheet may have many secondary, incidental effects on the spreadsheet's formulas and/or values. By tracking both, we can better understand these effects and minimize the complexities and unexpected side-effects of each operation.

Coordinate System. Cells are arranged into a 2-dimensional grid of rows and columns indexed by a coordinate system, a function $s : \mathbb{N} \times \mathbb{N} \to id$ that maps positions in the grid to the cell occupying that position. The function s need not be complete, but must be one-to-one: a cell may only appear in one position in the spreadsheet.

Formulas. A formula is a primitive-valued expression that may include references to the values of other cells, identified by the cell's global id or by absolute coordinates (explicit and absolute references, respectively). A formula evaluated in the context of a cell may also specify coordinate references as being relative to the cell (relative references). Columns are usually denoted by letters and rows by numbers. A *state* is a 2-tuple $\langle\ C, s\ \rangle$ consisting of a set of cells $C = \{C_i\}$ and a coordinate system. We say that a formula f evaluates to a value v in the context of a given state ($f \mapsto_{\langle\ C,s\ \rangle} v$) if, after replacing all references (coordinate references using s and C, and explicit references using C), the formula reduces to v [2]. We say that a state $\langle\ C, s\ \rangle$ is *valid* [3] if each cell's formula evaluates to the cell's value:

$$\forall \langle\ id_i, f_i, v_i\ \rangle \in C\ :\ f_i \mapsto_{\langle\ C,s\ \rangle} v_i$$

User *actions* in VizUAL transform a state $\langle\ C_1, s_1\ \rangle$ into a new state $\langle\ C_2, s_2\ \rangle$. We call the semantics for an action correct if they ensure that if the input to an action is valid, then the action's output is also valid.

3.1 Unsurprising Inconsistencies

User actions on a spreadsheet have both direct, intended effects, and may also have indirect, *incidental* effects. Examples include changing a formula (dependent formulas are recomputed), repositioning a row (formulas depending on the row are modified), or sorting (formulas are recomputed based on the new, sorted coordinate system). In commercial spreadsheet systems, indirect effect semantics can sometimes be inconsistent. Take, for example, two mechanisms for rearranging rows in the table given in Figure 3a. A user might manually drag row 3 to a position between rows 1 and 2, effecting a swap of rows 2 and 3. Microsoft Excel, Apple's Numbers, and Google's Sheets [4] all have identical behavior, each resulting in the table shown in Figure 3b. Note that the formulas for C2 and C3 have changed to ensure that each cell retains its original value under the transposed coordinate system. In other words, the user's `MOVE` action treats formula references as being *explicit* references. Conversely, a user might sort the rows of the table in descending order on Column B. The resulting table in all three systems is identical, and shown in Figure 3c. Here, the formulas in column C are changed only in appearance; each continues to reference the cells immediately to the left and above. However the values of each cell have changed as a result. In other words, the user's `SORT` action treats formula references as being *relative* references.

Clearly, in spite of the superficial similarity between these two operations, their semantics are quite different. How-

[2]Similar operational semantics were previously proposed by Krishnamurthi and Ramakrishnan [10].

[3]Note that this definition does not preclude direct or indirect circular references as long as the computations defined by the cell formulas have a fixpoint. However, such a fixpoint computation may be hard to understand for a user and, thus, we disallow circular references for now.

[4]These and other behaviors described were evaluated on Excel for Mac version 15.20, Numbers version 3.6.1, and Google Sheets as of April 2016.

	A	B	C
1	Alice	10	=B1 (10)
2	Bob	4	=B2+C1 (14)
3	Carol	8	=B3+C2 (22)
4	Dave	9	=B4+C3 (31)

(a) Initial State

	A	B	C
1	Alice	10	=B1 (10)
2	Carol	8	=B2+C3 (22)
3	Bob	4	=B3+C1 (14)
4	Dave	9	=B4+C3 (31)

(b) After swapping rows 2 and 3

	A	B	C
1	Alice	10	=B1 (10)
2	Dave	9	=B2+C1 (19)
3	Carol	8	=B3+C2 (27)
4	Bob	4	=B4+C3 (31)

(c) After sorting on column 'B'

Figure 3: Examples of both swapping rows and sorting rows in commercial database systems.

	Stability		
Action	**Excel**	**Numbers**	**Sheets**
Cut/Paste	V	F	V
Drag Cell/Row/Col	n/a	V	V
Insert Row/Col	V	V	V
Sort	F	F	F
Filter	V	V	V

Figure 4: Interface actions and whether they are **F**ormula-stable, or **V**alue-stable. Excel does not support dragging.

ever, viewed through the lens of VizUAL's data model, this design choice emerges as a form of the principle of least surprise [16]. Concretely, for purely structural operations (i.e., operations that simply manipulate the coordinate scheme), it is still necessary to propagate incidental effects to formulas and/or values. The `MOVE` action translates cell formulas into the new coordinate scheme — retaining stable values at the cost of changing formulas. Meanwhile the `SORT` action re-evaluates cell-formulas under the new coordinate scheme — retaining stable formulas at the cost of changing values. By enforcing one of these two forms of stability (value or formula) for each user action, spreadsheet designers are guarding against hard to follow "magic" semantics. We also note that VizUAL's data model admits both forms of stability.

We tested a range of structural actions, and all consistently exhibited one of these types of stability: either on values (formulas are translated), or on formulas (new values are computed). Our results are shown in Figure 4. Virtually all action semantics favor value stability — clearly the simpler case in general. Semantics that enforce formula stability are used primarily in sorting, which applies a non-intuitive, effectively random coordinate transform. The other outlier is Numbers, where the cut operation removes data immediately, compared to Excel and Sheets, where the cut operation simply marks data to be moved on the subsequent paste. This distinction allows Numbers to provide consistent paste semantics between cut and copy, while Excel and Sheets treat cut/paste as a special `MOVE`-like operation. Specific tradeoffs aside, each system exhibits a preference for value-stability, falling back to formula-stability for non-intuitive coordinate transforms.

3.2 Regions to Relations

Many operations in Vizier operate over sets or collections of cells. For example, aggregates in formulas, the 'paste' operation, and type conversions all target or reference entire regions of cells. In a typical spreadsheet, such regions are specified as rectangular regions of cells in the current coordinate system (e.g., `[A3 : B99]` or `[A : A]`). Conversely, in a relational setting, sets of target values are specified qualitatively through selection predicates.

The former semantics are critical for enabling the spreadsheet interaction model, while the latter is important for generalizing the curation workflow beyond the initial dataset. Existing data curation systems focus on the latter approach; Even Wrangler [12], which does allow users to initially write curation operators as singletons, still forces users to define a generalized predicate before moving on.

In VizUAL, regions combine both semantics. Concretely, a *region* is defined through a 3-tuple $\langle X, Y, f \rangle$, where X is a (possibly infinite) set of columns, Y is a (possibly infinite) set of rows, and f is a boolean-valued formula defining a predicate over cells in the specified range. All cells in the intersection of X and Y fulfilling f are part of the region.

4. WORKFLOW REWRITING

As the user makes edits in the spreadsheet interface, the corresponding actions are recorded in the notebook as a VizUAL script. Although these scripts do encode the evaluation logic that generates the spreadsheet being displayed, they also serve as an audit trail, tool for reverting or altering older edits, and vector for generalizing the same curation process to new data. As such, VizUAL is subject to a different set of optimization goals than most programming languages. Rather than optimizing for performance or resource usage as in a normal optimizer, VizUAL needs an optimizer that prioritizes both *readability* and *generality*.

4.1 Rewriting for Readability

User actions on the spreadsheet are expected to be small, isolated changes. Recording them directly in this form is likely to produce long, hard to follow VizUAL scripts. Thus, it will be necessary for Vizier to dynamically rewrite scripts being modified in a principled way that optimizes for readability. We consider readability to be a tradeoff between minimizing two measures: size and complexity. For example, consider a sequence of 10 update actions with the form:

```
UPDATE A = 3 WHERE ROWID = ?
```

with `?` taking values from 1 to 10. Instead, we could express all 10 updates in a single expression using a `BETWEEN` predicate that (a) more concisely represents the same concept, with (b) a similar level of complexity, and (c) is semantically equivalent. Similar transformations appear in optimizing compilers — the above equivalence inverts a common compiler optimization called loop unrolling. Although there has been substantial research effort on obfuscating compilers, we are not aware of any source-to-source compilers designed to increase code readability. Vizier will not just record a VizUAL script for a workflow, but also keep track of what user operations each operation in the script is based on. This information can be exploited during rewriting. A set of formulas created by an adapt&apply operation is a good candidate for rewriting, because we know that all formulas in such a set follow the same pattern.

4.2 Generalizing Singletons

Singletons allow users to try out hypotheticals, explore cleaning solutions, and conduct small-scale tests. It is often easier for users to perform one-off curation steps initially, repairing errors in the data as singletons, rather than expending the mental effort to generalize the repair upfront. However, when the user needs to adapt their preliminary

data cleaning solution to new data, to a larger dataset, or to an updated dataset, these singleton operations can become a burden. Although they put more control over the curation process in the user's hands, singleton actions increase the size and complexity of a VizUAL script, with no benefits beyond the initial dataset. In addition to considering readability-enhancing rewrites that preserve semantic equivalence, it will be necessary for Vizier to evaluate how singleton actions can be generalized — effectively a form of query (or curation, in this case) by example [22]. Concretely, given a set of similar statements with singleton targets, we would like to propose to the user a set of rewrites that apply a single update to many (or all) of the singletons at once.

5. RELATED WORK

Spreadsheet-style interfaces for relational data have been of interest to researchers and practitioners alike for some time now as a desperately needed form of direct data manipulation [11]. Tyszkiewicz [19] demonstrated an embedding of SQL into spreadsheet formula semantics. Excel provides database integration capabilities, and there is a spectrum of attempts at hybrid environments [4, 5, 12, 13, 20].

Spreadsheets with Workflows. Trifacta/Wrangler [12] have features that are similar to Vizier. As in Vizier, users generate curation workflows by directly editing data. However, unlike Vizier, there is no support for singleton operations in the workflow language – user edits must be generalized immediately through a recommendation interface. Query-by-Excel [20] (QBX) provides support for cube-style operations in a spreadsheet-like environment. Although the goal is different, the mechanism is quite similar: QBX allows singleton outputs in the cube query, encoding them as `UPDATE` operations on the query output. However, QBX treats only query outputs as mutable, while source data is fixed; VizUAL is free of this limitation.

Relational Spreadsheets. SheetMusiq [13] uses semantics for relational queries over spreadsheets. Though superficially similar to VizUAL, it assumes static data, and does not attempt to preserve formula semantics through queries. Related Worksheets [4] provide a spreadsheet UI for structured relational data, focusing on enabling strongly-typed data and foreign key references. However, although editing cells is permitted, the work does not address cell formulas. DataSpread [5] extends spreadsheets with relational database functionality: structured query support and a scalable relational engine for a backend. By comparison, VizUAL starts with a structured relational data model and extends it with the illusion of freeform editing.

Inspiration. The idea of generalizing singleton operations is based on Query by Example [22] and Query by Explanation [9]. As individual operations are grouped together, the system can learn to describe what the user is attempting to accomplish. We plan to draw heavily on work in this area to develop Vizier's generalization engine. As the basis for the notebook-style interface and script provenance, we leverage work on scientific workflows [7, 8, 17], and we borrow ideas from reenactment [2, 3] as the basis for VizUAL scripts.

6. CONCLUSIONS

We present our vision for Visier, a data curation system which exposes powerful curation operations through a UI that is a hybrid between the spreadsheet and notebook interface paradigms. In this work we focus on the user interface as well as present the initial design of a language VizUAL that can serve as the underlying computational model for operations in the system.

7. REFERENCES

[1] Y. Amsterdamer, S. B. Davidson, D. Deutch, T. Milo, J. Stoyanovich, and V. Tannen. Putting Lipstick on Pig: Enabling database-style workflow provenance. *PVLDB*, 5(4):346–357, 2011.

[2] B. Arab, D. Gawlick, V. Krishnaswamy, V. Radhakrishnan, and B. Glavic. Reenacting transactions to compute their provenance. Technical Report IIT/CS-DB-2014-02, Illinois Institute of Technology, 2014.

[3] B. Arab, D. Gawlick, V. Krishnaswamy, V. Radhakrishnan, and B. Glavic. Formal foundations of reenactment and transaction provenance. Technical Report IIT/CS-DB-2016-01, Illinois Institute of Technology, 2016.

[4] E. Bakke, D. Karger, and R. Miller. A spreadsheet-based user interface for managing plural relationships in structured data. In *SIGCHI*, 2011.

[5] Mangesh Bendre, Bofan Sun, Ding Zhang, Xinyan Zhou, Kevin Chen-Chuan Chang, and Aditya Parameswaran. DataSpread: Unifying databases and spreadsheets. *PVLDB*, 8(12):2000–2003, 2015.

[6] Y. E. Chan and V. C. Storey. The use of spreadsheets in organizations: Determinants and consequences. *JIDM*, 31(3):119 – 134, 1996.

[7] F. Chirigati and J. Freire. Towards integrating workflow and database provenance. In *IPAW*, pages 11–23. 2012.

[8] S. B. Davidson, S. Cohen-Boulakia, A. Eyal, B. Ludäscher, T. McPhillips, S. Bowers, and J. Freire. Provenance in Scientific Workflow Systems. *IEEE DEB*, 32(4):44–50, 2007.

[9] Daniel Deutch and Amir Gilad. Learning queries from examples and their explanations. *ArXiV*, 2016.

[10] M. Erwig and M. Burnett. *Practical Aspects of Declarative Languages*, chapter Adding Apples and Oranges, pages 173–191. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002.

[11] H. V. Jagadish, A. Chapman, A. Elkiss, M. Jayapandian, Y. Li, A. Nandi, and C. Yu. Making database systems usable. *SIGMOD*, 2007.

[12] S. Kandel, A. Paepcke, J. Hellerstein, and J. Heer. Wrangler: Interactive visual specification of data transformation scripts. In *SIGCHI*, 2011.

[13] B. Liu and H. V. Jagadish. A spreadsheet algebra for a direct data manipulation query interface. In *ICDE*, 2009.

[14] X. Niu, B. Arab, D. Gawlick, O. Kennedy Z. H. Liu, V. Krishnaswamy, and B. Glavic. Provenance-aware versioned dataworkspaces. In *TaPP*, 2016.

[15] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: A not-so-foreign language for data processing. In *SIGMOD*, 2008.

[16] J. H. Saltzer and M. F. Kaashoek. *Principles of computer system design: an introduction*. Morgan Kaufmann, 2009.

[17] C. E. Scheidegger, H. Vo, D. Koop, J. Freire, and C. T. Silva. Querying and Re-using Workflows with VisTrails. In *SIGMOD*, 2008.

[18] B. Sowell, A. Demers, J. Gehrke, N. Gupta, H. Li, and W. White. From declarative languages to declarative processing in computer games. Technical report, ArXiV, 2009.

[19] J. Tyszkiewicz. Spreadsheet as a relational database engine. In *SIGMOD*, 2010.

[20] A. Witkowski, S. Bellamkonda, T. Bozkaya, A. Naimat, L. Sheng, S. Subramanian, and A. Waingold. Query by Excel. In *VLDB*, 2005.

[21] Ying Yang, Niccolò Meneghetti, Ronny Fehling, Zhen Hua Liu, and Oliver Kennedy. Lenses: An on-demand approach to etl. *PVLDB*, 8(12):1578–1589, 2015.

[22] M. M. Zloof. Query by example. In *AFIPS*, 1975.