Your notebook is not crumby enough, REPLace it

Michael Brachmann[B], William Spoth[B], Oliver Kennedy[B], Boris Glavic[I],
Heiko Mueller[N], Sonia Castelo[N], Carlos Bautista[N], Juliana Freire[N]
[B]: University at Buffalo [I]: Illinois Institute of Technology [N]: New York University
{mrb24,wmspoth,okennedy}@buffalo.edu bglavic@iit.edu
{heiko.mueller, s.castelo, carlos.bautista, juliana.freire}@nyu.edu

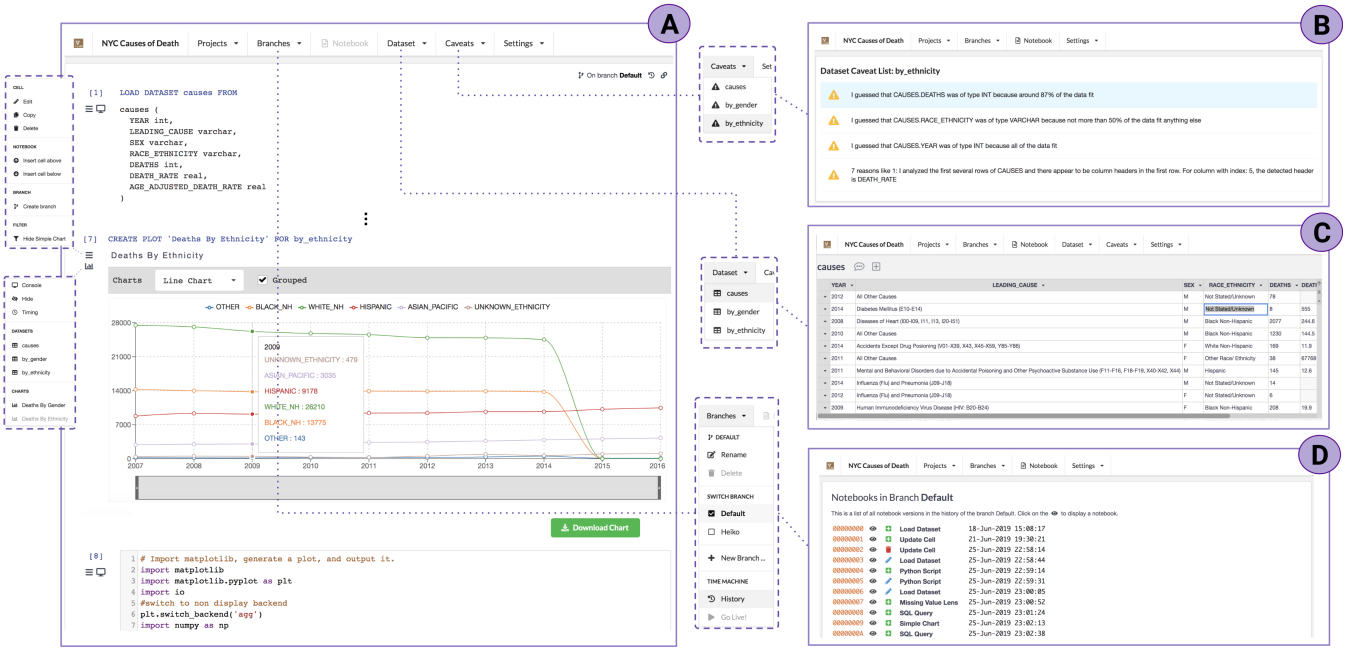


Figure 1: Overview of our system Vizier. The New York City Leading Causes of Death dataset was used. Vizier has four main views: (A) The Vizier Notebook View, (B) The Vizier Caveat View, (C) The Vizier Spreadsheet View and (D) The Vizier History View.

ABSTRACT

Notebook and spreadsheet systems are currently the de-facto standard for data collection, preparation, and analysis. However, these systems have been criticized for their lack of reproducibility, versioning, and support for sharing. These shortcomings are particularly detrimental for data curation where data scientists iteratively build workflows to clean up and integrate data as a prerequisite for analysis. We present Vizier, an open-source tool that helps analysts to build and refine data pipelines. Vizier combines the flexibility of notebooks with the easy-to-use data manipulation interface of spreadsheets. Combined with advanced provenance tracking for both data and computational steps this enables reproducibility, versioning, and streamlined data exploration. Unique to Vizier is that it exposes potential issues with data, no matter whether they already exist in the input or are introduced by the operations of a notebook. We refer to such potential errors as *data caveats*. Caveats are propagated alongside data using principled techniques from uncertain data management. Vizier provides extensive user interface support for caveats, e.g., exposing them as summaries in a dedicated error view and highlighting cells with caveats in spreadsheets.

WOODSTOCK '97 El Paso, Texas USA

ACM ISBN 123-4567-24-567/08/06…$15.00
DOI: 10.475/123_4

Keywords

Notebooks, Data Science, Provenance, Workflow System

1. INTRODUCTION

Notebook tools like Jupyter have emerged as a popular programming abstraction for data exploration and curation, model-building, and rapid prototyping. Notebooks promise re-use and reproducibility as well as interactive refinement of code with immediate feedback to support iterative construction of pipelines for data curation and analysis. However, a recent study by Pimentel et. al. [56] found only 4% of notebooks sampled from GitHub to be reproducible, and only 24% to be directly re-usable. These unfortunate statistics stem from Jupyter's heritage as a thin facade over a read-evaluate-print-loop (REPL). Many existing notebooks, like Jupyter, are not designed as a historical log — as one would want for reproducibility and iterative exploration — but rather as library managers for code snippets (i.e., cells). Reproducibility, re-usability, and iterative pipeline construction require active effort from users to organize cells, keep cells up to date, and manage inter-cell dependencies and cell versions (e.g., using tools like git) [43].

As been observed repeatedly (e.g., [47]), data scientists spend most of their time on the complex tasks of data curation and exploration. As a prerequisite for analysis, a data scientist has to find the right datasets for their task and then iteratively construct a pipeline of curation operations to clean and integrate the data. Typically, this is not a linear process, but requires backtracking, e.g., to fix a problem with a curation step that causes errors in later stages of the pipeline. The lack of support for automatic dependency tracking in existing tools is already detrimental, as problems with later stages introduced as a result of changing an earlier stage might never be detected. However, dependency tracking alone is not sufficient to aid an analyst in iterative refinement of a dataset. As the name curation implies, data problems are often repaired one bit at a time: In early stages of data exploration, data quality is less of a concern than data structure and content, and investing heavily in cleaning could be wasteful if the data turns out to be inappropriate for the analyst's needs. Similarly, in the early stages of data preparation, it is often necessary to take shortcuts like heuristic or manual repairs that, although sufficient for the current dataset and analysis, may not generalize. As progressively more critical decisions are made based on the data, such deferred curation tasks are typically revisited and addressed if necessary. However, deferring cleaning tasks requires analysts to keep fastidious notes, and to continuously track possible implications of the heuristic choices they make during curation.

There is substantial work on detecting data errors (e.g., [1, 22]) and streamlining the cleaning process (e.g., [22, 26]). However, effective use of error detection still requires *upfront* cleaning effort, and with only rare exceptions (e.g., [10]) automatic curation obscures the potential implications of its heuristic choices. Ideally a data exploration system would supplement automatic error detection and curation with infrastructure for tracking errors that the data scientist does not address immediately, and for highlighting the potential implications of automatic data curation heuristics.

In short, we target four limitations of existing work:

• **Reproducibility.** The nature of most existing notebook systems as wrappers around REPLs leads to non-reproducible analysis and unintuitive and hard to track errors during iterative pipeline construction.

• **Direct Manipulation.** It is often necessary to manually manipulate data (e.g., to apply simple one-off repairs), pulling users out of the notebook environment and limiting the notebook's ability to serve as a historical record.

• **Versioning and Sharing.** Existing notebook and spreadsheet systems often lack versioning capabilities, forcing users to rely on manual versioning using version control systems like git and hosting platforms (git forges) like github.

• **Uncertainty and Error Tracking.** Existing systems do not expose, track, or manage issues with data and deferred curation tasks, nor their interactions with data transformations and analysts

In this paper, we present Vizier,[1] a notebook-style data exploration system designed from the ground up to encourage reproducibility, workflow and dataset re-use, and proper error/uncertainty management. Vizier eschews REPLs in favor of a more powerful state model: A versioned database and workflow. This allows Vizier to act as a true workflow manager [6], precluding out-of-order execution, a common source of frustration[2] and cause of non-reproducible workflows [56]. To aid reproducibility, Vizier maintains a full version history for each notebook. We supplement Vizier's notebook interface with a tightly coupled "spreadsheet mode" that allows reproducible direct manipulation of data. Vizier facilitates debugging and re-usability of data and workflows by tracking potential data errors through a principled form of incompleteness annotations that we call *caveats*. Vizier can propagate these annotations through operations of a notebook based on a principled, yet lightweight, uncertainty model called UA-DBs [27]. While some aspects of Vizier such as automated dependency tracking for notebooks, versioning, and workflow provenance tracking are also supported by other approaches, the combination of these features and the support for caveats leads to a system that is more than the sum of its components and provides unique capabilities that to the best of our knowledge are not supported by any other approach.

Many aspects of Vizier, including parts of its user interface [28, 44], provenance models [4, 6], and caveats [66, 27], were explored independently in prior work. In this paper, we focus on the challenge of unifying these components into a cohesive, notebook-style system for data exploration and curation, and in particular on the practical implementation of caveats and Vizier's spreadsheet interface.

A Trail of Breadcrumbs. A common use of notebooks is to incrementally build a workflow to ingest, curate, and analyze a dataset. The notebook acts as a form of documentation, automatically preserving a record of a user's process as she explores a dataset: A trail of breadcrumbs.

EXAMPLE 1. *Alice the data engineer is exploring a newly acquired dataset in a notebook. She breaks her exploration down into discrete steps — each performing an isolated load, transformation, summarization, or visualization task. She iterates on each task within a single cell, revising its code as needed before moving to a new cell for the next step. As she encounters errors, she revises earlier steps as needed.*

Such *breadcrumbing* can be very powerful if performed

[1] `pip2 install vizier-webapi` https://vizierdb.info

[2] https://twitter.com/jakevdp/status/935178916490223616
https://multithreaded.stitchfix.com/blog/2017/07/26/nodebook
https://github.com/jupyter/notebook/issues/3229

correctly. Results produced from the notebook can be reproduced on new data; intermediate state resulting from earlier cells can be re-used for different analyses; or the notebook itself can be used as a prototype for a production system like a dashboard or data curation pipeline. However, breadcrumbing in a REPL-based notebook requires extreme diligence from the user. She must manually divide tasks into independent logical chunks. She must mentally track dependencies between cells to ensure that revisions to earlier steps do not break downstream cells. She must also explicitly design around cells with side effects like relational database updates or file writes.

1.1 A Notebook-Style Workflow Manager

The fundamental challenge is that REPL-based notebooks are stateful, and this state is managed independently of the notebook by the REPL. Using a REPL to manage state makes it difficult to preserve the linear order of execution implied by the presentation of the notebook as an ordered list of cells [43] for two main reasons. First, particularly in the presence of native code (e.g., for popular libraries like NumPy), it is difficult to rewind a REPL to an earlier state. Thus, if an existing cell is edited, it will be executed on the REPL's current state, requiring the *user* to ensure that it is operating on a consistent input state (e.g., by manually ensuring that each cell is idempotent). Second, treating the state as an opaque blob makes it difficult to automatically invalidate and recompute dependencies of a modified cell. Dependency tracking through REPL state is possible. However, such dependency tracking is often limited. For example, Nodebook [68] tracks data dependencies across Python cells and caches cell outputs, but makes the strong assumption that all variables are immutable. Even worse, the user is held responsible for enforcing this assumption.

Vizier addresses both problems through a simple, but robust, solution: isolating cells. In lieu of a REPL, cells execute in independent interpreter contexts. Our approach is similar to Koop's [43] proposal of a dataflow notebook where notebook cells can refer to specific outputs of specific cells, but without the need to manually manage inter-cell dependencies and with strong isolation guarantees for cells. Specifically, cells can communicate by consuming and producing *datasets*. This well-defined API enables efficient state snapshots, as well as dependency analysis across cell types (e.g., a Scala cell may depend on an SQL cell's output). Whenever Alice updates a cell, Vizier automatically re-executes all cells that directly or indirectly depend on the updated cell.

In Vizier each change to a notebook cell or edit in a spreadsheet creates, conceptually, a new version of the notebook (and of the results of all cells of the notebook). Vizier maintains a full history of these versions. With Vizier, if Alice's notebook evolves in a non-productive direction, she can always backtrack to any previous version. Furthermore, any version of a notebook and dataset has a unique URL that she can share with collaborators.

1.2 Caveats

Automated cell dependency tracking in notebook systems (e.g., [43]) is not new, but has historically focused on improving execution performance (e.g., by parallelizing independent cells) and aiding reproducibility and re-use. Vizier employs novel fine-grained annotations called caveats designed to help users to detect, document, and track the effects of heuristic assumptions and incompletely addressed data errors. Concretely, a caveat is an annotation attached to one or more cells or rows of a dataset (i.e., data elements), consisting of: (i) A human-readable description of a shortcut, error, or concern that could invalidate the annotated data elements and (ii) A reference to the workflow cell where the caveat was attached. Intuitively, a caveat indicates that an element is *potentially* erroneous (i.e., uncertain).

The need for caveats arises, because decisions made during error detection and cleaning typically are uncertain or depend on assumptions about the data. That is true no matter whether these decisions are made by the user or by an automated data curation method. For example, data values may be incorrectly flagged as errors or automated data cleaning techniques may chose an incorrect repair from a space of possible repair options. A way to model this uncertainty is to encode the space of possible repairs as an incomplete or probabilistic database. For example, work by Beskales et al. [10, 11] is based on this idea and applies probabilistic query processing to propagate uncertainty through operations. Probabilistic and incomplete data management provide a principled foundation for dealing with the uncertainty inherent in data or introduced by curation operations. However, more lightweight methods for uncertain data management are needed to make this feasible in practice [27, 66].

Creating Caveats. Notebook systems used for data curation should be able to model and track the uncertainty inherent in error detection and data cleaning. Curation may involve both automated error detection and repair methods as well as one-off heuristics applied by the user. Vizier supports caveat creation for both types of transformations.

We emphasize that caveats are orthogonal to any specific error detection or cleaning schemes. A wide range of such tools can be can be wrapped to expose their heuristic assumptions and any resulting uncertainty to Vizier, integrating them into the Vizier ecosystem. Vizier then automatically tracks caveats and handles other advanced features such as versioning and cell dependency tracking. In our experience, extending methods to expose caveats is often quite straight-forward (e.g., see [66] for several examples), and Vizier already supports a range of caveat-enabled data cleaning and error detection operations (see Figure 2 for a list of currently supported cell types). Similarly, Vizier's data load operation also relies on caveats as a non-invasive and robust (data errors do not block the notebook) way to communicate records that fail to load correctly. To support one-off curation transformations, Vizier also allows users to create caveats manually through its spreadsheet interface or programmatically via `Python`, `Scala`, or `SQL` cells.

Example 2. *Alice is analyzing a dataset of price lists from her company's suppliers. However, several suppliers manually enter data for the price field. As a result, her dataset includes many non-integer values like* `'21 USD'` *or* `'$19.00'`. *Alice knows that all of her company's suppliers are based in the United States and always quote prices in dollars. She quickly adds to her data preparation script a new step that uses a regular expression (e.g.,* `s/[^0-9.\-]+//g`*) to remove all non-digit characters from the price field.*

Heuristic transformations are significantly easier and faster to deploy than more general solutions. In some cases (like the above example) enumerating and addressing all possible

corner cases is simply not feasible, and a heuristic transformation is the only option. Unfortunately, heuristics tailored to one dataset or analysis may not generalize to other data or to other questions. Thus, neither the data preparation scripts implementing those heuristics, nor the resulting prepared data can be safely re-used in the future.

EXAMPLE 3. *Months later, Alice's company has grown significantly and her colleague Bob revisits Alice's analysis on new data. The company now sources parts from suppliers across the world and Alice's original assumption that all prices are in US Dollars is no longer valid. Bob would like to leverage Alice's existing data preparation script in his new analysis. However, to do so safely, he needs to (i) understand all of the heuristic assumptions Alice made, (ii) evaluate which are still applicable, and (iii) update parts of the script based on assumptions that are invalid for his analysis.*

Current best practices suggest documenting heuristic data transformations out-of-band. For example, Alice might note her assumption that "all prices are in US Dollars" in a README file attached to the dataset, or in a comment in her data preparation script. However, even when such documentation exists, it can get lost in a pile of similar notes, many of which may not be relevant *to a specific user asking a specific question.* Moreover, the documentation is not directly associated with the individual data values that it refers to. Out-of-band documentation makes it difficult for a user making changes to understand how the assumptions affect their code and data. Using Vizier, Alice can ensure that her assumptions are documented and will transition when her workflow is re-used in the future.

EXAMPLE 4. *As Alice creates her data preparation script, she ensures that it marks cells affected by her regular expression data cleaning step with a caveat indicating her assumption. Later, when Bob re-uses Alice's script, any prices containing non-digit characters will be marked by caveats.*

Automatic Propagation of Data Caveats. Documenting errors and uncertainty is important. However, for complex workflows the user needs to understand how caveats for one dataset affect data produced by cells that directly or indirectly depend on this dataset. Vizier supports this functionality by automatically propagating caveats through data transformations based on uncertain data management principles. Formally, we may think of a data value annotated with a caveat as representing a *labeled null* (the correct value is unknown) complemented with a guess for the unknown value (the annotated data value). Then a dataset with caveats can be thought of as an approximation of an incomplete database [39] where the values encode one possible world and the caveats encode an under-approximation of the certain values in the incomplete database: any row not marked with a caveat is a *certain answer*. Wherever possible, Vizier preserves this property when propagating caveats. We note that unlike traditional incomplete (or probabilistic) databases, we can not assume that users will be able to (or have the time) to completely and precisely characterize the uncertainty in their data and workflows. Thus, precisely characterizing the set of certain answers is in general not possible. We apply our conservative approximation from [27] to propagate caveats. For certain cell types no techniques for propagating incompleteness in this fashion are known. Thus, Vizier propagates caveats based on fine-grained provenance (data-dependencies) when supported (e.g., for SQL queries) or based on coarse-grained provenance when fine-grained provenance is not supported for a cell type (e.g., a Python cell). The rationale is that a value is most likely affected by caveats associated with values it depends on[3].

Caveats in Vizier's UI. Vizier highlights caveatted values in the spreadsheet view of a dataset [44] to indicate that more attention is required. Caveat propagation is done on an as-needed basis, and only caveats applied to values or records that could affect the output are propagated. Vizier features an error view that shows a dataset-specific summary of all caveats affecting the dataset (Section 4). Vizier can also generate caveat reports for any result value, summarize caveats affecting the value or its component elements.

EXAMPLE 5. *When Bob repurposes Alice's data preparation script, Vizier produces a list of caveats affecting his data. Bob can choose to immediately address these, or continue filtering and refining the data. When he is ready, Vizier generates a report summarizing which caveats could affect the final dataset he produces for his analysis. When drilling down to investigate individual values affected by Alice's heuristics, Bob may find that values with other currencies, e.g,* `'21 Euro'` *have been incorrectly translated into USD. Bob can then modify Alice's workflow to fix this problem. For example, he could write code to check for other currency names and apply appropriate exchange rates.*

1.3 Overview and Contributions

Concretely, this paper makes the following contributions. (i) We outline the design of Vizier, a notebook-style workflow system (Section Section 2); (ii) We explore the challenges of integrating spreadsheet and notebook (Section Section 3) interfaces; and (iii) We introduce caveats, a practical implementation of uncertainty annotations [27] (Section 4), and discuss their implementation in Vizier.

2. THE VIZIER NOTEBOOK

Vizier is an interactive code notebook similar to Jupyter[4] or Apache Zeppelin[5]. An analytics workflow is broken down into individual steps called cells. Workflow cells are displayed in order, with each cell shown with code (e.g., Python or SQL) implementing the step, and any outputs (e.g., console output, data visualizations, or datasets) produced by this code. Each cell sees the effects of all preceding (upstream) cells, and can create effects visible to subsequent (downstream) cells. The user may edit, add, or delete any cell in the notebook, which *triggers re-execution of all dependent cells appearing below it in the notebook.*

Vizier workflows allow different cell types to be mixed within the same notebook. Figure 2 lists the cell types presently supported by Vizier. Classical notebook cell types like Python, SQL, and Markdown are all supported. Vizier additionally supports cells that use point-and-click interfaces to streamline (1) Common notebook tasks like data ingest/-export and visualization; (2) Spreadsheet-style dataset ma-

[3]Of course, this is not guaranteed to be the case, e.g., if a missing value with caveats should have been included.
[4]https://jupyter.org/
[5]https://zeppelin.apache.org/

Category	Cell Type Examples	API
Script	Python, Scala	Workflow
Query	SQL	Dataflow
Documentation	Markdown	n/a
Point/Click	Plot, Load Data, Export Data	Workflow
Spreadsheet DML/DDL	Add/Delete/Move Row, Add/Delete/Move/Rename Column, Edit Cell, Sort, Filter	Dataflow
Cleaning	Infer Types, Repair Key, Impute, Repair Sequence, Merge Columns, Geocode	Dataflow

Figure 2: Cell Types in Vizier

nipulation (Section 3); and (3) Data curation and cleaning (Section 4).

2.1 Cells and Workflow State

Dataflow in a typical workflow system (e.g., VisTrails [59, 6, 25]) is explicit. Steps in the workflow define outputs that are explicitly bound to the inputs expected by subsequent steps. Conversely, data flow in a notebook is implicit: each cell manipulates a global, shared state. For example, in Jupyter, this state is the internal state of the REPL interpreter itself (variables, allocated objects, etc...). Jupyter cells are executed in the context of this interpreter, resulting in a new state visible to subsequently executed cells.

In designing the state model for Vizier, we considered four requirements. Of these, only one is satisfied by an opaque, REPL-based state model.

1. **Enforcing in-order execution**: We needed a state representation that could be correctly, efficiently checkpointed to allow replay from any point in the notebook. A typical REPL's state is a complex graph of mutable objects that can be checkpointed completely or efficiently, but not both simultaneously.

2. **Workflow-style execution**: We needed a state representation that permits at least coarse-grained dependency analysis between cells. Operations in a REPL can have arbitrary side effects, making dependency analysis challenging.

3. **Fine-grained provenance**: To support provenance-based features of Vizier like caveats, we wanted a state representation that supported declarative state transformations. A typical REPL for an imperative language like Python makes such analysis challenging.

4. **Big data support**: We needed a state representation that could manage large, structured datasets. This criterion is met by Python REPLs, which offer limited support for large, in-memory datasets through libraries like NumPy and Pandas.

A versioned relational database satisfies all four criteria, providing efficient checkpointing, instrumentable and declarative updates, as well as natural parallelism for scalability. Thus, relational tables provide the foundation for Vizier's state model: A notebook's state is a set of named relational tables called datasets[6].

[6]Support for primitive values and BLOB state is in progress.

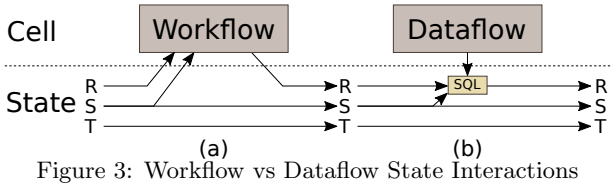


Figure 3: Workflow vs Dataflow State Interactions

Cells access, manipulate, create, or destroy datasets through one of two instrumented APIs, illustrated in Figure 3. As we discuss below, the behavior of some cell types can be expressed declaratively, making fine-grained instrumentation for provenance and caveat tracking at the level of individual data values possible. Implementations of such *dataflow* cells access datasets through a functional API that allows new datasets to be defined as SQL views over existing datasets, i.e., datasets are treated as immutable from the viewpoint of the code using the API. For example, the dataflow cell in Figure 3 is translated into a SQL view definition that redefines the dataset named `R` in terms of the dataset `S` and the previous version of `R`. Other cell types admit only coarse-grained instrumentation. These *workflow* cells are supported through language-specific read/write APIs that allow datasets to be queried and replaced. For example, the workflow cell in Figure 3 reads from datasets `R` and `S` to create a new version of the dataset `R`.

Workflow Cells. The workflow API, illustrated in Figure 3.a, targets cells where only collecting coarse-grained provenance information is presently feasible. This includes `Python` and `Scala` cells, which implement Turing-complete languages; as well as cells like `Load Dataset` that manipulate entire datasets. To discourage out-of-band communication between cells (which hinders reproducibility), as well as to avoid remote code execution attacks when Vizier is run in a public setting, workflow cells are executed in an isolated environment. Vizier presently supports execution in an fresh interpreter instance (for efficiency) or a docker container (for safety). Vizier's workflow API is designed accordingly, providing three operations: *Read dataset* (copy a named dataset from Vizier to the isolated execution environment), *Checkpoint dataset* (copy an updated version of a dataset back to Vizier), and *Create dataset* (allocate a new dataset in Vizier). A more efficient asynchronous, paged version of the *read* operation is also available, and the *Create dataset* operation can optionally initialize a dataset from a URL, S3 Bucket, or Google Sheet to avoid unnecessary copies.

Dataflow Cells. The dataflow cell API, illustrated in Figure 3.b is aimed at cell types that implement declarative dataset transformations. Dataflow cells are compiled down to equivalent SQL queries. Updated versions of the state are defined as views based on these queries. We emphasize that most dataflow cells do not require the user to write SQL. SQL is the language used by the implementation of such a cell to communicate with Vizier. For example, spreadsheet operation cells created as a consequence of edits in the spreadsheet and interactively configured cleaning operations are both dataflow cells.

2.2 Version Model

In addition to versioning state, Vizier also versions the notebook itself, through a branching version history illus-

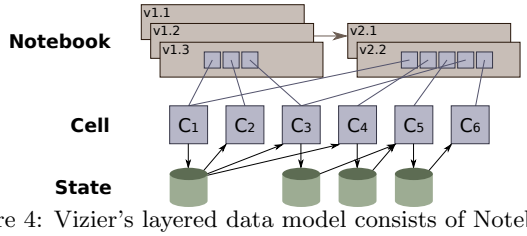


Figure 4: Vizier's layered data model consists of Notebook, Cell, and State versions.

trated in Figure 4. This version graph tracks three types of objects: Notebooks, Cells, and Datasets.

Notebook Versions. A single version of the notebook is an immutable sequence of references to the immutable cells that specify a workflow. Every edit a user makes to a notebook, whether adding a new cell, editing an existing cell, or deleting a cell creates a new notebook version, preserving a full, reproducible history of the notebook. Following the workflow provenance model from [17], we record for each notebook version the operation performed to create the version, and the previous, *parent version* of the notebook. The result is a version graph, a tree-like structure that shows the notebook's evolution over time. Under typical usage, edits are applied to a leaf of the tree to create chains of edits.

Editing a prior version of the notebook creates a *branch* in the version history. Vizier requires users to explicitly name branches when they are created; This explicit branch management makes it easier for users to follow the intent of a notebook's creator. Internally however, each notebook version is identified by a 2-tuple consisting of a randomly generated, unique branch identifier and a monotonically increasing notebook version number.

Cell Versions. A cell version is an immutable specification of one step of a notebook workflow. In its simplest form, the cell version stores the cell's type, as well as any parameters of the cell. This can include simple parameters like the table and columns to plot for a `Plot` cell, scripts like those used in the `SQL`, `Scala`, and `Python` cells, as well as references to files like those uploaded into the `Load Dataset` cell. In short, the cell configuration contains everything required to deterministically re-execute the cell[7]. A cell version is identified by an identifier derived from a hash of its parameters.

Alongside the cell version, Vizier also stores a cache of results derived from executing the cell. Unlike the immutable cell version itself, this cache is mutable, and is updated every time the cell is executed. The cell cache specifically includes: (i) Outputs for presentation to the user, including console output and HTML-formatted data plots; (ii) Coarse-grained (workflow) provenance information, including a list of datasets the cell read from and wrote to; and (iii) A reference to each dataset version produced by the cell.

Dataset Versions. A dataset is presented to the user as a mutable relational table identified by a human readable name. Internally however, a dataset version is an immutable Spark dataframe identified by a randomly generated, globally unique identifier. Keeping dataset versions immutable makes it possible to quickly recover notebook state in between cells and to safely share state across notebook branches. To preserve the illusion of mutability, Vizier maintains a *scope* that maps human-readable dataset names to the appropriate identifiers. When a cell is executed, it receives a scope that maps dataset names to dataset version identifiers. To create or modify a dataset, a cell first initializes a new dataset version: uploading a new dataframe (workflow cells) or creating a Spark view (dataflow cells). The scope entry for the corresponding dataset is updated accordingly, and passed to the next cell. The result is the illusion of mutability, but with state checkpoints between each cell. Of course, it may be more efficient to compute deltas between versions instead taking full copies or even to trade computation for storage (recompute the dataset when required). For instance, such ideas have been applied in Nectar [36] and DataHub [14]. We leave such optimization to future work.

[7]Of course, we assume here that the computation of the cell itself is deterministic. For cells with non-deterministic computation , e.g., random number generators, we cannot guarantee that multiple execution of the same cell yield the same result.

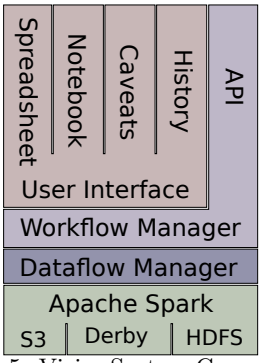


Figure 5: Vizier System Components

2.3 Dependency Models

As illustrated in Figure 6, Vizier manages two graphs of dependencies: one at the dataflow and one at the workflow level. Workflow dependencies form a coarse-grained view of which cells depend on which other cells, and are used to manage cell execution order. When a cell is executed, dataset accesses are instrumented to compute a **read set** that records the user-facing names of datasets accessed by the cell and a **write map** that records user-facing names of dataset versions created by the cell with the new version's identifier (or `NULL` if the cell deleted the dataset). For each dataset in a cell's read set, Vizier creates a workflow edge to the most recent cell to modify that dataset (i.e., the last cell with that dataset in its write set). For example, in Figure 6, the `Python` cell creates a new version of dataset `R` that is read by both the `Merge` and `Insert Row` cells. Thus both of the latter cells depend on the former.

Dataflow dependencies provide a fine-grained view of how a dataset version was derived and are used primarily for propagating caveats. For dataflow cells, Vizier simply stores the SQL query used to generate the dataset version. For example, dataset version $\texttt{S}_1$ is derived from both dataset versions $\texttt{R}_1$ and $\texttt{S}_0$. For workflow cells like the `Python` cell, exact derivation information is not available. Instead, Vizier creates coarse-grained provenance records based on the cell's read and write sets: dataset versions emitted by the cell are linked to every dataset version the cell read from.

2.4 Execution Model

Figure 5 overviews the Vizier system, including the its two core components: The workflow manager and the dataflow manager. The workflow manager is a simplified version of

the VisTrails [6] workflow system, and is responsible for managing cells, inter-cell dependencies, scheduling workflow cell execution, and the version history. The dataflow manager is implemented using the Mimir incomplete database [66, 27, 44], and is responsible for fine-grained provenance, data loading, Vizier's "lens" [66] data curation operators (the cleaning cell types supported by Vizier), and functionality required to support caveats. Mimir, in turn, is implemented as a query-rewriting front-end over Apache Spark, which handles query evaluation. At the other end of the Vizier stack is a user interface that provides a range of modalities for interacting with and analyzing the workflow.

2.4.1 The Workflow Manager

The workflow manager is responsible for managing cells, inter-cell dependencies, scheduling workflow cell execution, and the version history. It exposes a REST API that allows clients like Vizier's user interface to query state and manipulate notebooks. Clients can create or delete notebooks; add, update, or delete cells; or create new notebook branches. The workflow manager tracks notebook and cell versions, as well as a dependency graph between cells.

The workflow manager executes cells asynchronously. As a *workflow* cell is added or updated, it is scheduled for execution through either a lightweight worker process, or through a celery distributed task queue[8] if one is configured. Both execution engines establish a connection to the dataflow manager through which the cell can access existing datasets or upload new dataset versions. When a *dataflow* cell is added or updated, the workflow manager allocates a new, globally unique view identifier and synchronously creates the view in the dataflow manager.

Execution Scheduling. Anytime a cell is added, updated, or deleted, its transitive dependencies must be re-executed as well. The challenge is that, at least for workflow cells, dependencies are not inferred statically — they are observed from execution traces. We require cell execution to be deterministic: If a cell's dependencies are unchanged, the cell does not need re-execution. When a cell does need re-execution, we pessimistically assume all downstream datasets could be affected until execution completes. Scheduling and execution strategies are presented in simplified forms in Algorithms 1 and 2, respectively. When the notebook cell at index i changes, Algorithm 1 marks it as `dirty` and marks all downstream cells as `waiting` (i.e., potentially dirty).

Algorithm 1 `schedule_updates(`$\mathcal{N}$`, `i`)`

Require: $\mathcal{N}$: A notebook; i: Index of updated cell.
  $\mathcal{N}$`[`i`].state` $\leftarrow$ `dirty`
  for $j \in i+1 \ldots$ `len(`$\mathcal{N}$`)` **do**
    $\mathcal{N}$`[`j`].state` $\leftarrow$ `waiting`

The asynchronous evaluation engine, summarized in Algorithm 2 waits until one or more cells are marked `dirty` and executes the first cell so marked. The algorithm then marks any cells that read from a dataset touched by the executed cell as `dirty`. `waiting` cells upstream of the first remaining `dirty` cell (if any) are now guaranteed to be unchanged, and can be safely marked as `ready`. Although not presently supported by Vizier, we note that this naive algorithm can be optimized when cell dependencies can be derived statically (e.g., in the case of dataflow cells).

[8]http://www.celeryproject.org/

Algorithm 2 `async_eval(`$\mathcal{N}$`)`

Require: $\mathcal{N}$: A notebook
  loop
    $\mathcal{D} \leftarrow \{\, i \mid \mathcal{N}$`[`$i$`].state = dirty` $\}$ ▷ Find dirty cells
    if $\mathcal{D} \neq \emptyset$ **then**
      $i \leftarrow$ `min(`$\mathcal{D}$`)`; `eval(`$\mathcal{N}$`[`i`])` ▷ Eval first dirty cell
      for $j \in 1 \ldots i$ **do** ▷ Mark cells up-to-date
        $\mathcal{N}$`[`j`].state` $\leftarrow$ `ready`
      for $j \in i+1 \ldots$ `len(`$\mathcal{N}$`)` **do** ▷ Mark dependencies
        if `reads(`$\mathcal{N}$`[`j`])` $\cap$ `writes(`$\mathcal{N}$`[`i`])` $\neq \emptyset$ **then**
          $\mathcal{N}$`[`j`].state` $\leftarrow$ `dirty`

2.4.2 The Dataflow Manager

The dataflow manager is responsible for storing and mediating access to datset versions, for propagating caveats, and for fine-grained provenance analysis. It exposes an API to the workflow manager and the cell execution engine that allows (i) new dataset versions to be uploaded or linked from URLs, (ii) new dataset versions to be defined declaratively as views, (iii) existing dataset versions to be queried, and (iv) the caveats of existing dataset versions to be analyzed.

Creating Datasets. A new *literal* dataset version may be created from scratch, either by uploading a data file or by providing a remote URL. Datasets from uploaded files and remote URLs (e.g., HTTP/S, google sheets, JDBC) are cached in, depending on configuration, the local filesystem, an S3 bucket, or HDFS. If the source is a flat file (e.g., an HTTPS link to a CSV file), the file is cached as-is. If not (e.g., a google sheet), the source is cached as an Apache parquet file. The dataflow manager saves the URL of the cached copy and returns a unique version identifier to the caller. If a URL already references a local resource, like a file in a configured S3 bucket, the URL is used directly. When a dataset is created from scratch, the caller may optionally create virtual provenance links to other existing datasets as described above.

Alternatively, a *declarative* dataset version may be created as a view in one of three ways. The caller provides: (i) a standard SQL view definition, (ii) a script in an imperative, DDL/DML-style language called Vizual [28] used to implement spreadsheet operations, or (iii) a lens definition [66] used to implement cleaning cells. We discuss Vizual further in Section 3 and lenses further in Section 4. Regardless of how the views are specified, the dataflow manager records the corresponding view definition in an intermediate representation based on Relational Algebra [51] and returns a unique identifier to the caller.

Dataset Access. The dataflow layer is built on Apache Spark [67] for scalability. Literal datasets are stored by the dataflow manager as URLs. The dataflow manager selects a Spark data loader (e.g., CSV, parquet) appropriate for the URL and creates a Spark dataframe. Queries and views defined over these dataframes are compiled from Vizier's intermediate representation to Spark's intermediate query representation, and optimized and executed directly in Spark. During this translation, queries are instrumented through a lightweight rewriting scheme [66, 27] that marks caveatted cells and rows as discussed in Section 4.

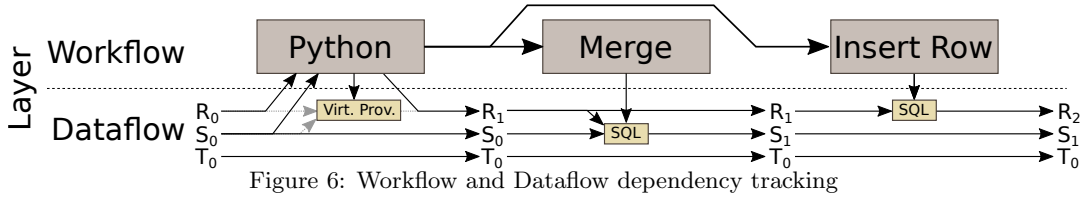


Figure 6: Workflow and Dataflow dependency tracking

Avoiding Spark's Data Store. The attentive reader may note that Spark already provides a datastore that can be used to save and query named datasets and views. The decision to re-implement this functionality in the dataflow layer is driven by one major factor: supporting Vizier's intermediate query representation. Keeping a separate intermediate representation permits the dataflow manager to implement caveats through lightweight query re-writing, rather than by modifying Spark directly. This reduces the effort required to maintain Vizier for new versions of Spark, and adds a layer of portability. For example, Spark's scalability comes with a high startup cost [63]. Portability allows the dataflow engine to execute queries, in whole or in part, internally through lighter-weight operator implementations when it is advantageous to do so.

Analyzing Caveats. When a dataset version is accessed, the dataflow manager marks data elements (i.e., cells and rows) that have a caveat applied. To avoid impacting data access latencies, these markings simply indicate the presence or absence of a caveat, but not the associated message or metadata. Thus, the dataflow manager exposes an interface that allows callers to retrieve the specific caveats affecting a given cell, row, column, or dataset on an as-needed basis. We discuss caveat handling in more detail in Section 4.

2.4.3 The User Interface

Most user interactions happen through a user interface (UI) written in React/Javascript. The Vizier UI consists of four main views: (i) The Notebook view links datasets and visualizations to the steps taken to generate them, (ii) The Spreadsheet view provides a direct manipulation interface for datasets, (iii) The Caveat view provides dataset-specific summaries of caveats, and (iv) The History view lays out how the current notebook was derived.

Notebook View. Vizier's notebook serves as a trail of breadcrumbs: It presents datasets and visualizations created by the current notebook, as well as the sequence of steps needed to create them. Figure 7a shows an example of two cells in a notebook: A SQL script cell followed by a Plot cell. When a notebook is first opened, the upper part of each cell shows the cell's script (in the case of Script cells), or a short textual summary of the cell (for all other cell types). Clicking on a summarized cell, or adding a new cell displays a form that configures the cell's behavior (e.g., Figure 7b).

The lower part of each cell displays status information. While the cell is running (i.e., in a `dirty` or `waiting` state), the status area shows information about execution progress (Figure 8). When the cell completes, the status area can be used to show: (i) Console output (i.e., from script cells), (ii) HTML data visualizations generated by the cell (e.g., from the Plot cell or Python plotting libraries like Bokeh), or (iii) A tabular view of any dataset version as it would appear immediately after the cell. When a dataset is displayed (Figure 7a), caveatted cells are highlighted in red. Users may manually switch the status view to display any available output or any dataset version at that point in the notebook. This allows users to, for example, debug a cell by simultaneously viewing before and after dataset versions.

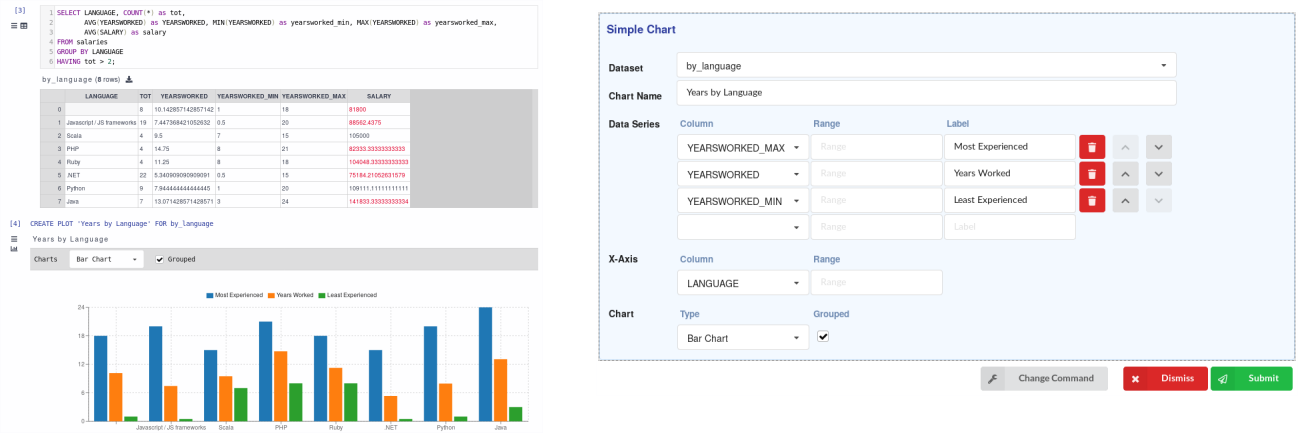
Spreadsheet View. Although data curation tasks can often be scripted, there are numerous situations where a manual override is more efficient. Practical examples include: (i) Cleaning tasks requiring manual data validation (e.g., personally contacting a cab driver to confirm the $1000 tip that the dataset claims that they received), (ii) Subjective data entry tasks (e.g., "tagging" user study transcripts), (iii) One-off repairs requiring human attention (e.g., standardizing notation in a free-entry text field from a survey), or (iv) Transient "what-if" exploration (e.g., how is an analysis affected when outliers are removed). Manual overrides are often performed in a text editor or through a spreadsheet.

In Vizier, manual data overrides are supported through a spreadsheet-style interface, illustrated in Figure 9. Opening a dataset version in this spreadsheet view displays the dataset as a relational table. Users may modify the contents of cells; insert, reorder, rename, or delete columns and rows; or apply simple data transformations like sorting. We note that in addition to allowing manual data overrides, the spreadsheet interface can be simpler and more accessible to novice users.

To facilitate Vizier's role in creating a trail of breadcrumbs for analysts, edits made in the spreadsheet view are reflected in the notebook. Vizier creates a cell for each spreadsheet interaction (Figure 2) using special cell types that each correspond to specific operations in Vizual, a DDL/DML language we developed for this purpose [28]. For instance, there are cell types for adding or deleting columns from a dataset. For each user interaction with the spreadsheet, a corresponding Vizual cell is created. As spreadsheet edits are typically lightweight and likely to occur in rapid succession, the notebook view collapses all such operations into a single Vizual cell that stores a script of Vizual operations.

After a Vizual command has been created in response to a spreadsheet action, the spreadsheet view must be refreshed. However, the normal cell execution workflow is too heavyweight to allow this refresh to happen at interactive speeds. To make value updates "feel" instantaneous, refreshes are asynchronous; While the spreadsheet view is being updated the client uses a placeholder value, typically the value the user typed in. Following the caveat metaphor, placeholder values are highlighted in red until the refresh completes.

Caveat View. As described above, caveats are a form of documentation that can be attached to individual data values or records. Both the Notebook and Spreadsheet views allow users to look up caveats for individual elements. However, on large datasets it is useful to see a summary of all caveats affecting the dataset. The Vizier UI provides one caveat tab for each dataset present at the end of the notebook. When a caveat tab is opened, the dataflow manager



(a) Notebook Cells

(b) Cell Configuration

Figure 7: The Vizier Notebook View

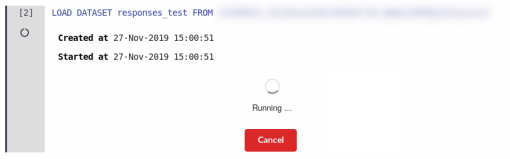


Figure 8: A Cell Being Executed

is queried (see Section 4) to generate a list of all caveats affecting the specified dataset as illustrated in Figure 10. Each caveat is displayed with a human-readable description of the error, for example as an input datum that could not be properly cast to the type of the column. As discussed in Section 4, the caveat list is a summary, with caveats organized into groups based on the type of error. The interface also allows caveats to be acknowledged by clicking on the caveat and then clicking "Acknowledge." An acknowledged caveat is still displayed in the caveat list, but otherwise ignored. For example, Vizier will not highlight cells that depend on it.

History View. As noted above, Vizier maintains a branching notebook history. The history view shown in Figure 11 displays the history of the current branch: the sequence of edits that led to the currently displayed version of the notebook. Any prior version of the notebook may be opened in a read-only form. If the user wishes to edit a prior version of the notebook, they can create a new branch from that version.

3. SPREADSHEET OPERATIONS

Spreadsheet edits like value updates or row/column insertions must be reflected in the notebook. Vizier supports a range of cell types based on a language for spreadsheet-style operations (e.g., delete a row or delete a column) called *Vizual* [28] that models user actions on a spreadsheet. In this section, we explore three challenges we had to overcome to implement DML/DDL operation cells in Vizier: (i) Data Types in Spreadsheets, (ii) Declarative Updates, and (iii) Row Identity.

3.1 Spreadsheet Data Types

The lightweight interface offered by typical spreadsheets has two impedance mismatches with the more strongly typed relational data model used by Vizier's datasets. First, types in a spreadsheet are assigned on a per-value basis, but on a per-column basis in a typical relational table. A spreadsheet allows users to enter arbitrary text into a column of integers. Because Vizier's history makes undoing a mistake trivial, Vizier assumes the user's action is intentional: column types are escalated (e.g., `int` to `float` to `string`) to allow the newly entered value to be represented as-is.

The second impedance mismatch pertains to `NULL` values. Spreadsheets do not distinguish between empty strings and missing values (i.e., a relational `NULL`). Thus, when a user replaces a value with the empty string, the user's intention is ambiguous. To resolve this ambiguity, Vizier relies on the type of the column: If the column is string-typed, an empty value is treated as the empty string. Otherwise, an empty value is treated as `NULL`.

3.2 Reenactment for Declarative Updates

Through the spreadsheet interface, users can create, rename, reorder, or delete rows and columns, or alter data — a standard set of DDL and DML operations for spreadsheets. These operations can not be applied in-place without sacrificing the immutability of versions. To preserve versioning and avoid unnecessary data copies, Vizier builds on a technique called reenactment [53, 3], which translates sequences of DML operations into equivalent queries. We emphasize that our use of the SQL code examples shown in this section are produced automatically as part of the translation of Vizual into SQL queries. Users will not need to write SQL queries to express spreadsheet operations. The user' actions in the spreadsheet are automatically added as Vizual cells to the notebook and these Vizual operations are automatically translated into equivalent SQL DDL/DML expressions [28].

Example 6. *Consider a table* `EMP` *and the SQL:*

```
UPDATE EMP SET pay=pay*1.1 WHERE type = 1;
INSERT INTO EMP(name, pay, type)
            VALUES ('Bob', 100000, 2);
```

Using reenactment, the version of the table after these operations are applied can be equivalently obtained by evaluating the following query over the initial `EMP` *table.*

```
SELECT name, type,
       (CASE type WHEN 1 THEN 1.1*pay
                         ELSE pay END) AS pay
FROM EMP UNION ALL
SELECT 'Bob' AS name, 100000 AS pay, 2 AS type;
```

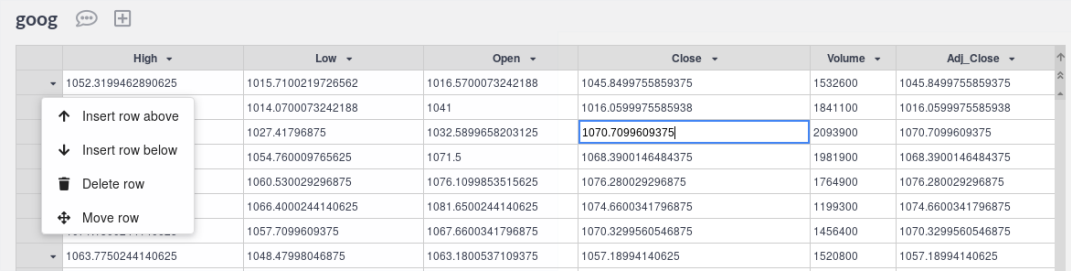


Figure 9: The Vizier Spreadsheet View

Figure 10: The Vizier Caveat View

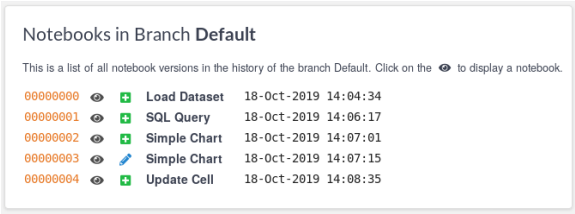


Figure 11: The Vizier History View

Vizier translates primitive cell operations into equivalent queries, resulting in each table version being defined (declaratively) as a view. We refer the interested reader to [53] for an introduction to reenactment, with a further discussion of its applicability to spreadsheets in [28].

Vizual [28], and by extension our primitive cell types, also cover DDL operations like column renaming, reordering, and creation. We implemented these through a similar view-based approach, adopting transformations similar to the ones from the Prism Workbench [23] and [37]. For example, column creation is analogous to projecting on the full schema of the table plus an additional column initialized with the new column's default value (or `NULL`).

3.3 Associating Updates with Data

Identifying update targets for Vizual cell operations presented a challenge. In SQL DML, update operations specify target rows by a predicate. By contrast, spreadsheet users specify the target of an update by explicitly modifying a cell at a certain position in the spreadsheet, e.g., overwriting the value of the cell in the second column of the 3rd row. To deal with such updates and to be able to represent unordered relational data as a spreadsheet we need to maintain a mapping between rows and their positions in the spreadsheet. Since we record both the position of a row and a unique stable identifier for it, we can ensure that a Vizual operation always applies to the same cell even when, e.g., rows/columns are deleted or added. However, when source data changes — for example when a new cell is added earlier in the notebook — determining how to reapply the user's update is more challenging. Ideally, we would like to use row identifiers that are stable through such changes.

For derived data, Vizier uses a row identity model based on GProM's [3] encoding of provenance. Derived rows, such as those produced by declaratively specified table updates, are identified as follows: (1) Rows in the output of a projection or selection use the identifier of the source row that produced them; (2) Rows in the output of a `UNION ALL` are identified by the identifier of the source row and an identifier marking which side of the union the row came from[9]; (3) Rows in the output of a cross product or join are identified by combining identifiers from the source rows that produced them into a single identifier; and (4) Rows in the output of an aggregate are identified by each row's group-by attribute values.

What remains is the base case: datasets loaded into Vizier or created through the workflow API. We considered three approaches for identifying rows in raw data: order-, hash-, and key-based. None of these approaches is ideal: If rows are identified by position, changes to the source data (e.g., uploading a new version) may change row identities. Worse, identifiers are re-used, potentially re-targeting spreadsheet operations in unintended ways. Using hashing preserves row identity through re-ordering, but risks collisions on duplicate data, and is sensitive to changes in column values. Using keys addresses both concerns, but requires users to manually specify a key column, assuming one exists in the first place. Our prototype implementation combines the first two ap-

[9]To preserve associativity and commutativity during optimization, union-handedness is recorded during parsing

proaches: deriving identifiers from both sequence and hash code. Such row-identifiers are stable under appends. While techniques for creating identifiers that are stable under updates has been studied extensively for XML databases (e.g., ORDPATH [55]) and recently also for spreadsheet views of relational databases [7], the main challenge we face in Vizier is how to retain row identity when a new version of a dataset is loaded into Vizier. In this scenario we only have access to two (identifier-free) snapshots of the dataset and no further information on how they relate to each other.

4. MANAGING CAVEATS

Vizier's central feature is support for annotations on groups of cells or rows called caveats. As introduced above, caveats consist of a human-readable description of a concern about the row or value, and a reference to the cell where the caveat was applied. More concretely, a caveat applied to a cell indicates that the cell value is potentially suspect or uncertain — typically because it is an outlier or its validity is tied to a heuristic assumption made during data preparation. Similarly, caveats applied to rows (resp., excluded rows) indicate the presence (resp., absence) of a row in a dataset is similarly suspect. In each case, if the heuristic is inappropriate (e.g., because a dataset is re-used for a new analysis) or the outlier indeed indicates an error, the analysis must be revisited.

Vizier propagates caveats based on data-dependencies: *Is it possible to change the derived value by changing the caveatted values?* If so, the derived value is likewise tied to the heuristic choice or outlier and likewise suspect if the heuristic/outlier turns out to be wrong. Caveats were originally introduced as *uncertain values* in [66]. We later formalized propagation of row caveats in [27] and proposed a minimal-overhead rewrite-based implementation. For dataflow cells that utilize only query features that are supported by these approaches, we use these techniques to provide the following strong guarantee: If all invalid (uncertain) data values in the input are marked by a caveat, all outputs not marked by a caveat are guaranteed to be valid (certain). Here, we focus on the practical challenges of realizing caveats in Vizier[10]

Example 7. *Consider a simpler version of our running example where Alice uses Vizier's default (caveat-enabled) string parsing operation, as described below. This operation replaces unparseable numbers with* `NULL`*. Bob applies Alice's script to his data and begins exploring with a query:*

```
SELECT id, avg(cost) FROM parts GROUP BY id;
```

4.1 Applying Data Caveats

Vizier's dataflow layer exposes a new scalar function to annotate data with caveats: `caveat(id, value, message)`. This function is meant to be used inline in SQL queries written by the user (a SQL cell) or produced by dataflow cells. `caveat` takes as input a value to annotate and a message describing the caveat. A unique identifier (e.g., derived from the row id) is used for book-keeping purposes, and omitted from examples for conciseness. Rows (as well as excluded rows) are annotated when the caveatted value is accessed in a `WHERE` clause that evaluates to `true`. Vizier also provides a manual annotation cell. The workflow dataset API also supports passing a list of caveats to apply when a dataset is uploaded. Additionally, several existing Vizier cells make use of caveats to encode heuristic choices about a dataset or to communicate heuristic recovery from errors. We now use two of Vizier's cell types to illustrate how caveats are used to annotate data.

Instrumenting String Parsing. Many file formats lack type information (e.g., CSV), or have minimal type systems (e.g., JSON). Thus extracting native representations from strings is a common task. For example consider the query:

```
SELECT CAST(cost as int) AS cost, ... FROM parts;
```

Vizier rewrites `CAST` to emit a `NULL` annotated with a caveat when a string can not be safely parsed:

```
CASE WHEN CAST(cost as int) IS NULL
     THEN caveat(NULL, cost || ' is not an int')
     ELSE CAST(cost as int) END
```

If the cast fails, it is replaced by a null value caveatted with a message indicating that the invalid string could not be cast.

Instrumenting CSV Parsing. CSV files are subject to data errors like un-escaped commas or newlines, blank lines, or comment lines. Purely rewrite-based annotations are not possible, as no dataframe exists during CSV parsing. Instead, Vizier adopts an instrumented version of Spark's CSV parser that emits an additional field that is `NULL` on lines that successfully parse and contains error-related metadata otherwise. Caveats are applied in a post-processing step.

```
SELECT * FROM parts_raw WHERE
  CASE WHEN _error_msg IS NOT NULL
  THEN caveat(true, _error_msg) ELSE true END;
```

This use of the `WHERE` clause seems un-intuitive at first, but is a deliberate decision rooted in caveats' origin in incomplete databases. Due to the parse error, we are not certain that the row is valid. Here `caveat(true, ...)` captures that the choice to include the row (i.e., `WHERE true`) is in question.

4.2 Propagating Caveats

Unlike general annotation management systems like Mondrian [31] or DBNotes [12] which let the user decide how annotations propagate, we propagate annotations based on the uncertainty semantics of caveats for operations supported by our approach [27, 66]. However, Vizier applies a further refinement: a caveat is only propagated if changing the caveatted value could affect the output. For operations not supported by these techniques we use provenance information at the finest granularity available to determine how to propagate caveats.

Example 8. *Consider the expression* `A = 1 OR B = 2`*. It's results depends on two input values* `A` *and* `B`*. If we know certainly that* `A = 1` *(*there are no caveats associated with the value*), then the result will be* `true`*, regardless of the value of* `B`*. Even if the current value of* `B` *is changed or incorrect, it can not affect the result of the expression and the caveat is not propagated. Conversely, if* `A = 1` *had a caveat, then an error in* `B` *could affect the expression if there was also an error in* `A` *and the caveat is propagated.*

4.3 Implementing Data Caveats

To minimize overhead during data access and query processing, Vizier splits propagation of caveats into two parts.

[10] As before, the SQL code shown here is for strictly pedagogical purposes or the result of automated processed in Vizier; Managing caveats does not require the user to write SQL.

A light-weight instrumentation method is used to rewrite queries to track the presence (resp., absence) of caveats on individual data values or records. As noted above, such elements are highlighted when they are displayed to the user. When needed, Vizier can undertake the more expensive task of deriving the full set of caveats associated with a given data value, row, or dataset.

4.3.1 Instrumenting Queries

Even just determining whether a data value or row is affected by a caveat is analogous to determining certain answers for a query applied to an incomplete database [39] (i.e., CoNP-complete for relatively simple types of queries [27]). Thus, Vizier adopts a conservative approximation: All rows or cells that depend on a caveatted value are guaranteed to be marked. It is theoretically possible, although rare in practice [27] for the algorithm to unnecessarily mark cells or rows. Specifically, queries are rewritten recursively using an extension of the scheme detailed in [27] to add Boolean-valued attributes that indicate whether a column, or the entire row depends on a caveatted value.

EXAMPLE 9. *Consider Bob's* `parts` *query, instrumented as above to cast* `cost` *to an integer and assume no other operations that create caveats have been applied. Let* `parts_uncast` *be the dataset before the cast operation and* `parts` *be the dataset after the operation. Recall the Bob's query is:*

```
WITH parts AS (/* as string parsing, above */)
SELECT id, avg(cost) FROM parts GROUP BY id;
```

This query would be instrumented and optimized into:

```
WITH parts AS (
  SELECT CAST(cost as int) AS cost, id, ...,
         (CAST(cost as int) IS NULL
         OR _caveat_cost) AS _caveat_cost
  FROM parts_uncast)
SELECT id, avg(cost),
       exists(_caveat_cost) AS _caveat_avg,
FROM parts GROUP BY id;
```

A new `_caveat_cost` *column is introduced, marking rows of the input affected by caveats. Caveat columns guaranteed to be false are optimized out, as is the* `caveat` *function itself.*

4.3.2 Computing Caveat Details

When the full set of caveats for a cell, row, or entire dataset is required, Vizier's dataflow layer employs a two-phase evaluation strategy: (i) A quick static analysis is enough to present a summary to users, and (ii) a dynamic analysis that refines the static analysis' summary.

Static Analysis. The initial phase as explained above simply determines which fields and rows are affected by caveats. At this point, the user can request more detailed information through the Vizier user interface. In the spreadsheet view, clicking on a field or row-header opens up a pop-up listing caveats on the field or row. Vizier also provides a dedicated view to list caveats on any dataset and its rows, fields, or columns. As before, we adopt a conservative approximation — it is possible, though rare, for a caveat to be displayed in this list unnecessarily.

The first step of generating the caveat details views is to statically analyze the query. This analysis produces a *CaveatSet* query, which computes the `id` and `message` parameters for every relevant call to the `caveat` function.

EXAMPLE 10. *Bob now asks for caveats affecting the average price of part* `12345`. *The* `caveat` *function is used exactly once, so we obtain a single CaveatSet:*

```
SELECT cost || ' is not an int' AS message,
       ROWID AS id
FROM parts_uncast WHERE id = '12345'
```

To generate the *CaveatSets* of a query, the query is first refined through selection and projection to produce the specific field(s) or row(s) being analyzed. Selection pushdown and dead-column elimination are used to push filters as close to the query leaves as possible. For each selection, projection, or aggregation in which a caveat appears, we construct a query to compute parameters for every call to `caveat`.

Dynamic Analysis. Unioned together, the CaveatSets for a query compute the full list of caveats affecting the target. Even in simple data pipelines with small datasets, there may be thousands of potential caveats and dealing with these caveats can easily overwhelm the user. Thus, exposing caveats at the right level of abstraction in the right context is paramount. When a CaveatSet contains more than three caveats, we select one representative example by `LIMIT`ing the query and present it alongside a `count(DISTINCT id)` of the results.

4.4 Cells with Coarse-Grained Provenance

Propagating caveats through queries is sufficient for the declaratively specified datasets created by SQL cells, most point-and-click cells, and Vizual cells (spreadsheet operations). However, certain cell types like Python cells produce datasets for which fine-grained provenance is not available. In such cases, we take a conservative approach and rely on the cached read-sets for the cell. When the cell writes to a table, we register a coarse-grained dependency from each of the datasets the cell has read from to the table being updated.

5. EXPERIMENTS

In this section, we evaluate Vizier's incremental caveat construction process. Concretely, we evaluate whether: (1) instrumenting workloads to detect the presence of caveats on result fields or rows adds minimal overhead, and (2) incrementally constructed caveats can be sufficiently responsive.

All experiments were performed on a 12-core 2.50GHz Intel(R) Xeon(R) E5-2640 with 198GB RAM, running Ubuntu 16, OpenJDK 1.8.0_22, Scala 2.11.11, and Spark 2.4. To minimize noise from the HTTP stack, we report timing numbers as seen by Mimir with a warm cache. All datasets were loaded through Vizier and cached locally in Parquet format.

Dataset	Rows	Cols	Caveats	CaveatSets
`Shootings`	2.9K	43	121	51
`Graffiti`	985K	15	47	28

Figure 12: Datasets Evaluated

We base our experiments on the experiments of [27], using the two real world datasets summarized in Figure 12. For each dataset, we enable four caveat-generating operators: header detection, type inference, error-aware CSV parser, and missing value repair. We then pose a single 2-column group-by aggregate query mirroring the non-aggregate query used by [27]. We measure the time taken to run the query both with and without instrumentation, the time taken for static analysis, and for CaveatSet expansion.

Figures 13 and 14 show a timeline of the query and caveat generation process.[11] Raw (uninstrumented) query execution time is shown for comparison. Instrumentation for caveat tracking is minimal, adding at worst 30% to an already fast-running query. Through parallel execution, the majority of caveats are rendered in seconds. We note the high overhead of queries in Spark, and are exploring a hybrid distributed+local engine to accelerate small interactive queries.

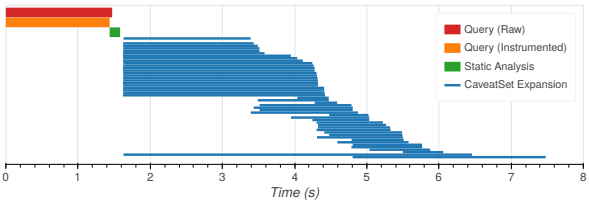


Figure 13: Materializing caveats for `Shootings`

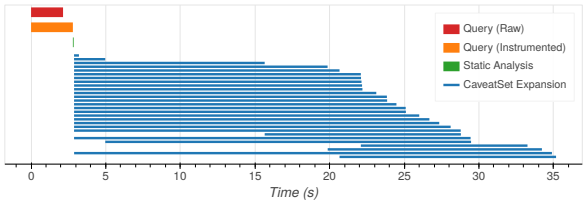


Figure 14: Materializing caveats for `Graffiti`

6. RELATED WORK

Our work has connections to error detection and data cleaning, notebook systems, dataset versioning, provenance in workflows and databases, uncertain data management, and spreadsheet interfaces for databases. We discuss each of these areas in the following.

Error Detection, Data Curation and Cleaning. Automated data curation and cleaning tools help users to prepare their data for analysis by detecting and potentially repairing errors [1, 22]. These tools employ techniques such as constraint-based data cleaning [26], transformation scripts aka wrangling [41], entity resolution [33, 21] and data fusion [15], and many others. While great progress has been made, error detection and repair are typically heuristic in nature, since there is insufficient information to determine which data values are erroneous let alone what repair is correct. Vizier enhances existing data cleaning and curation techniques by exposing the uncertainty in their decisions as data caveats and tracks the effect of caveats on further curation and analysis steps using a principled, yet efficient solution for incomplete data management [66, 27]. Thus, our solution enhances existing techniques with new functionality instead of replacing them. Based on our experience, wrapping existing techniques to expose uncertainty is often surprisingly straight-forward. We expect to extend Vizier with many additional error detection and cleaning techniques in the future.

Notebook Systems. Notebook systems like Jupyter and Zeppelin have received praise for their interactivity, for providing immediate feedback through interleaving code and results, and for integrating documentation with source code. However, these systems have also been widely criticized for their lack of collaboration features, versioning, and reproducibility [56]. We are not the first to observe that treating a notebook as a dataflow graph can lead to improved reproducibility and a more consistent user experience because it allows dependent cells to be updated automatically and resolves the non-determinism introduced by executing cells in a certain order in REPL-based notebooks. Koop et al. [43] discusses how the lack of data-dependency tracking among cells in a notebook affects the correctness and reproducibility of Jupyter notebooks. The proposed solution to these problems is a Jupyter kernel that attaches unique identifiers to cells in a notebook. Cells can reference the output of other cells using these identifiers. Based on these inter-cell data-dependencies, the system automatically refreshes dependent cells when a cell re-executed. NiW [19] converts notebooks into workflows to enable provenance tracking and reproducibility. Nodebook [68] automatically tracks dependencies across cells in a Jupyter notebook and caches outputs of cells to ensure reproducibility and enable automated refresh of dependent cells for notebooks that use Python exclusively. Like these approaches Vizier notebooks are also data-centric, i.e., they represent a dataflow graph. However, in contrast to NiW and Nodebook, Vizier natively supports many different programming languages. Since cells communicate by consuming and producing datasets using Vizier's dataset API, adding support for new languages and cell types is straightforward.

Versioning and Provenance. Another problem with notebook systems is their lack of versioning capabilities. For reproducibility and collaboration, it is essential to keep track of both versions of the datasets produced and consumed by notebooks as well as versions of the notebook itself. Versioning is closely related to data provenance which tracks the creation process of data keeping track of both dependencies among data items and the processes and actors involved in the creation process. The W3C PROV standard [49] has been proposed as an application-independent way of representing provenance information. Provenance in workflow systems has been studied intensively in the past [19, 16, 25, 59, 24, 29, 59, 17, 6]. So-called retrospective provenance, the data and control-dependencies of a workflow execution, can be used to reproduce a result and understand how it was derived. Koop [42] and [17] propose to track the provenance of how a workflow evolves over time in addition to tracking the provenance of its executions. Niu et al. [53] use a similar model to enable "provenance-aware data workspaces" which allow analysts to non-destructively change their workflows and update their data. In the context of dataset versioning, prior work has investigated optimized storage for versioned datasets [65, 13, 48]. Bhattacherjee et al. [14] study the trade-off between storage versus recreation cost for versioned datasets. The version graphs used in this work essentially track coarse-grained provenance. The Nectar system [36] automatically caches intermediate results of distributed dataflow computations also trading storage versus computational cost. Similarly, metadata management systems like Ground and Apache Atlas (https://atlas.apache.org/) manage coarse-grained provenance for datasets in a data lake. In contrast to workflow provenance which is often coarse-grained, i.e., at the level of datasets, database provenance is typically more fine-grained, e.g., at the level of rows [20, 38, 3, 4, 34, 60, 50]. Many systems

[11] Plots generated with Vizier using Bokeh.

capture database provenance by annotating data and propagating these annotations during query processing. Vizier's version and provenance management techniques integrate several lines of prior work by the authors including tracking the provenance of workflow versions [58, 52], provenance tracking for updates and reenactment [4, 53], and using provenance-based techniques for tracking uncertainty annotations [66, 27]. The result is a system that is more than the sum of it components and to the best of our knowledge is the first system to support all of these features.

Uncertain Data. Vizier's cavets are a practical application of uncertain data management. Incomplete [35, 18, 62, 40], inconsistent [30, 9, 18], and probabilistic databases [61, 54, 64, 2, 57] have been studied for several decades. However, even simple types of queries become intractable when evaluated over uncertain data. While approximation techniques have been proposed (e.g., [35, 54, 32]), these techniques are often still not efficient enough, ignore useful, albeit uncertain, data, or do not support complex queries. In [27] we formalized *uncertainty-annotated databases* (*UA-DBs*), a light-weight model for uncertain data where rows are annotated as either certain or uncertain. In [66] we introduced Lenses which are uncertain versions of data curation and cleaning operators that represent the uncertainty inherent in a curation step using an attribute-level version of the UA-DB model. Data cavets in Vizier generalize this idea to support non-relational operations and to enrich such annotations with additional information to record more details about data errors.

Data Spreadsheets. Approaches like DataSpread and others [8, 46, 5] utilize spreadsheet interfaces as front-ends for databases. Vizier stands out through its seamless integration of spreadsheets and notebooks [28]. Like other approaches that improve the usability of databases [45], Vizier provides a simple user interface that can be used effectively by both experts and non-experts and does not require any background in relational data processing to be understood. Furthermore, we argue in [28] that the spreadsheets and notebook interfaces complement each other well for data curation and exploration tasks. For example, spreadsheets are suited well for handling rare exceptions by manually updating cells and are convenient for certain schema-level operations (e.g., creating or deleting columns) while notebooks are more suited for complex workflows and bulk operations (e.g., automated data repair). Integrating the spreadsheet paradigm which heavily emphasizes updates, e.g., a user overwrites the value of a cell, with Viziers functional, dataflow model of notebook workflows would have been challenging if not for our prior work on *reenactment* [4, 3, 4, 53]. Reenactment enables us to translates updates into queries (side-effect free functions).

7. CONCLUSIONS AND FUTURE WORK

In this paper, we discuss the design and implementation of Vizier, a novel system for data curation and exploration. Vizier's UI is a combination of a spreadsheet and a notebook interface. In contrast to other library-manager style notebook systems (i.e., wrappers around REPLs), Vizier is a manager for versioned workflows. Vizier supports iterative notebook construction through automated data-dependency tracking and debugging through the automated detection and propagation of *caveats*. In future work, we will investigate (fine-grained) propagation of caveats for new cell types, e.g., displaying caveats in plots, and explore trade-offs between performance overhead and accuracy when propagating caveats through cells with turing-complete languages. Furthermore, we plan to develop caching and incremental maintenance techniques for datasets in Vizier workflows to speed-up reexecution of cells in response to an update to a notebook. Finally, to support very large datasets, we will investigate how to incorporate sampling into Vizier.

8. REFERENCES

[1] Z. Abedjan, X. Chu, D. Deng, R. C. Fernandez, I. F. Ilyas, M. Ouzzani, P. Papotti, M. Stonebraker, and N. Tang. Detecting data errors: Where are we and what needs to be done? *PVLDB*, 9(12):993–1004, 2016.

[2] L. Antova, C. Koch, and D. Olteanu. Maybms: Managing incomplete information with probabilistic world-set decompositions. In *ICDE*, pages 1479–1480, 2007.

[3] B. Arab, S. Feng, B. Glavic, S. Lee, X. Niu, and Q. Zeng. GProM - a swiss army knife for your provenance needs. *IEEE Data Eng. Bull.*, 41(1):51–62, 2018.

[4] B. Arab, D. Gawlick, V. Krishnaswamy, V. Radhakrishnan, and B. Glavic. Using reenactment to retroactively capture provenance for transactions. *TKDE*, 30(3):599–612, 2018.

[5] E. Bakke and D. R. Karger. Expressive query construction through direct manipulation of nested relational results. In *SIGMOD*, pages 1377–1392, 2016.

[6] L. Bavoil, S. P. Callahan, C. E. Scheidegger, H. T. Vo, P. Crossno, C. T. Silva, and J. Freire. Vistrails: Enabling interactive multiple-view visualizations. In *IEEE Visualization*, pages 135–142, 2005.

[7] M. Bendre, B. Sun, D. Zhang, X. Zhou, K. C.-C. Chang, and A. G. Parameswaran. Dataspread: Unifying databases and spreadsheets. *PVLDB*, 8(12):2000–2003, 2015.

[8] M. Bendre, V. Venkataraman, X. Zhou, K. C.-C. Chang, and A. G. Parameswaran. Towards a holistic integration of spreadsheets with databases: A scalable storage engine for presentational data management. In *ICDE*, pages 113–124, 2018.

[9] L. Bertossi and J. Chomicki. Query answering in inconsistent databases. In *Logics for emerging applications of databases*, pages 43–83. 2004.

[10] G. Beskales, I. F. Ilyas, and L. Golab. Sampling the repairs of functional dependency violations under hard constraints. *PVLDB*, 3(1):197–207, 2010.

[11] G. Beskales, M. A. Soliman, I. F. Ilyas, S. Ben-David, and Y. Kim. Probclean: A probabilistic duplicate detection system. In *ICDE*, pages 1193–1196, 2010.

[12] D. Bhagwat, L. Chiticariu, W. C. Tan, and G. Vijayvargiya. An annotation management system for relational databases. *VLDBJ*, 14(4):373–396, 2005.

[13] A. Bhardwaj, A. Deshpande, A. J. Elmore, D. Karger, S. Madden, A. Parameswaran, H. Subramanyam, E. Wu, and R. Zhang. Collaborative data analytics with datahub. *PVLDB*, 8(12):1916–1919, 2015.

[14] S. Bhattacherjee, A. Chavan, S. Huang, A. Deshpande, and A. Parameswaran. Principles of dataset versioning: Exploring the recreation/storage tradeoff. *PVLDB*, 8(12):1346–1357, 2015.

[15] J. Bleiholder and F. Naumann. Data fusion. *ACM Comput. Surv.*, 41(1):1:1–1:41, 2009.

[16] S. Bowers, T. McPhillips, and B. Ludäscher. Provenance in collection-oriented scientific workflows. *Concurrency and Computation: Practice and Experience*, 20(5):519–529, 2008.

[17] S. P. Callahan, J. Freire, E. Santos, C. E. Scheidegger, C. T. Silva, and H. T. Vo. Managing the evolution of dataflows with vistrails. In *ICDE Workshops*, pages 71–71, 2006.

[18] A. Calì, D. Lembo, and R. Rosati. On the decidability and complexity of query answering over inconsistent and incomplete databases. In F. Neven, C. Beeri, and T. Milo, editors, *PODS*, pages 260–271, 2003.

[19] L. A. Carvalho, R. Wang, Y. Gil, and D. Garijo. Niw: Converting notebooks into workflows to capture dataflow and provenance. In *K-CAP Workshops*, pages 12–16, 2017.

[20] J. Cheney, L. Chiticariu, and W.-C. Tan. Provenance in databases: Why, how, and where. *Foundations and Trends in Databases*, 1(4):379–474, 2009.

[21] P. Christen. *Data Matching - Concepts and Techniques for Record Linkage, Entity Resolution, and Duplicate Detection*. Data-Centric Systems and Applications. Springer, 2012.

[22] X. Chu, I. F. Ilyas, S. Krishnan, and J. Wang. Data cleaning: Overview and emerging challenges. In *SIGMOD*, pages 2201–2206, 2016.

[23] C. A. Curino, H. J. Moon, and C. Zaniolo. Graceful database schema evolution: The prism workbench. *PVLDB*, 1(1):761–772, 2008.

[24] S. B. Davidson, S. Cohen-Boulakia, A. Eyal, B. Ludäscher, T. McPhillips, S. Bowers, and J. Freire. Provenance in scientific workflow systems. *IEEE Data Eng. Bull.*, 32(4):44–50, 2007.

[25] S. B. Davidson and J. Freire. Provenance and scientific workflows: challenges and opportunities. In *SIGMOD*, pages 1345–1350, 2008.

[26] W. Fan, F. Geerts, and X. Jia. A revival of integrity constraints for data cleaning. *PVLDB*, 1(2):1522–1523, 2008.

[27] S. Feng, A. Huber, B. Glavic, and O. Kennedy. Uncertainty annotated databases - a lightweight approach for approximating certain answers. In *SIGMOD*, 2019.

[28] J. Freire, B. Glavic, O. Kennedy, and H. Mueller. The exception that improves the rule. In *HILDA*, 2016.

[29] J. Freire and C. T. Silva. Making computations and publications reproducible with vistrails. *Computing in Science and Engineering*, 14(4):18–25, 2012.

[30] A. D. Fuxman and R. J. Miller. First-order query rewriting for inconsistent databases. In *ICDT*, pages 337–351, 2005.

[31] F. Geerts, A. Kementsietsidis, and D. Milano. Mondrian: Annotating and querying databases through colors and blocks. In *ICDE*, page 82, 2006.

[32] F. Geerts, F. Pijcke, and J. Wijsen. First-order under-approximations of consistent query answers. *International Journal of Approximate Reasoning*, 83:337–355, 2017.

[33] L. Getoor and A. Machanavajjhala. Entity resolution: theory, practice & open challenges. *PVLDB*, 5(12):2018–2019, 2012.

[34] B. Glavic, R. J. Miller, and G. Alonso. Using sql for efficient generation and querying of provenance information. *In search of elegance in the theory and practice of computation: a Festschrift in honour of Peter Buneman*, pages 291–320, 2013.

[35] S. Greco, C. Molinaro, and I. Trubitsyna. Algorithms for computing approximate certain answers over incomplete databases. In *IDEAS*, pages 1–4, 2018.

[36] P. K. Gunda, L. Ravindranath, C. A. Thekkath, Y. Yu, and L. Zhuang. Nectar: Automatic management of data and computation in datacenters. In *OSDI*, pages 75–88, 2010.

[37] K. Herrmann, H. Voigt, T. B. Pedersen, and W. Lehner. Multi-schema-version data management: data independence in the twenty-first century. *VLDBJ*, 27(4):547–571, 2018.

[38] M. Herschel, R. Diestelkämper, and H. B. Lahmar. A survey on provenance: What for? what form? what from? *VLDB*, pages 1–26, 2017.

[39] T. Imieliński and W. Lipski, Jr. Incomplete information in relational databases. *J. ACM*, 31(4):761–791, 1984.

[40] T. Imieliński and W. Lipski Jr. Incomplete information in relational databases. *JACM*, 31(4):761–791, 1984.

[41] S. Kandel, A. Paepcke, J. M. Hellerstein, and J. Heer. Wrangler: interactive visual specification of data transformation scripts. In *CHI*, pages 3363–3372, 2011.

[42] D. Koop. Versioning version trees: The provenance of actions that affect multiple versions. In *IPAW*, pages 109–121, 2016.

[43] D. Koop and J. Patel. Dataflow notebooks: Encoding and tracking dependencies of cells. In *TaPP*, 2017.

[44] P. Kumari, S. Achmiz, and O. Kennedy. Communicating data quality in on-demand curation. In *QDB*, 2016.

[45] F. Li and H. V. Jagadish. Usability, databases, and hci. *IEEE Data Eng. Bull.*, 35(3):37–45, 2012.

[46] B. Liu and H. V. Jagadish. A spreadsheet algebra for a direct data manipulation query interface. In *ICDE*, pages 417–428, 2009.

[47] S. Lohr. For big-data scientists, 'janitor work' is key hurdle to insights. http://nyti.ms/1Aqif2X, 2014.

[48] M. Maddox, D. Goehring, A. J. Elmore, S. Madden, A. Parameswaran, and A. Deshpande. Decibel: The relational dataset branching system. *PVLDB*, 9(9):624–635, 2016.

[49] P. Missier, K. Belhajjame, and J. Cheney. The w3c prov family of specifications for modelling provenance metadata. In *EDBT*, pages 773–776, 2013.

[50] T. Müller, B. Dietrich, and T. Grust. You say 'what', i hear 'where'and 'why'—(mis-) interpreting sql to derive fine-grained provenance. *PVLDB*, 11(11), 2018.

[51] A. Nandi, Y. Yang, O. Kennedy, B. Glavic, R. Fehling, Z. H. Liu, and D. Gawlick. Mimir: Bringing ctables into practice. Technical report, 2016.

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

[53] X. Niu, B. S. Arab, S. Lee, S. Feng, X. Zou, D. Gawlick, V. Krishnaswamy, Z. H. Liu, and B. Glavic. Debugging transactions and tracking their provenance with reenactment. *PVLDB*, 10(12):1857–1860, 2017.

[54] D. Olteanu, J. Huang, and C. Koch. Approximate confidence computation in probabilistic databases. In *ICDE*, pages 145–156, 2010.

[55] P. E. O'Neil, E. J. O'Neil, S. Pal, I. Cseri, G. Schaller, and N. Westbury. Ordpaths: Insert-friendly xml node labels. In *SIGMOD*, pages 903–908, 2004.

[56] J. F. Pimentel, L. Murta, V. Braganholo, and J. Freire. A large-scale study about quality and reproducibility of jupyter notebooks. In *MSR*, 2019.

[57] C. Re, N. Dalvi, and D. Suciu. Query evaluation on probabilistic databases. *IEEE Data Eng. Bull.*, 29(1):25–31, 2006.

[58] C. E. Scheidegger, D. Koop, E. Santos, H. T. Vo, S. P. Callahan, J. Freire, and C. T. Silva. Tackling the provenance challenge one layer at a time. *Concurrency and Computation: Practice and Experience*, 20(5):473–483, 2008.

[59] C. E. Scheidegger, H. Vo, D. Koop, J. Freire, and C. T. Silva. Querying and re-using workflows with vistrails. In *SIGMOD*, pages 1251–1254, 2008.

[60] P. Senellart, L. Jachiet, S. Maniu, and Y. Ramusat. Provsql: provenance and probability management in postgresql. *PVLDB*, 11(12):2034–2037, 2018.

[61] D. Suciu, D. Olteanu, C. Ré, and C. Koch. Probabilistic databases. *Synthesis Lectures on Data Management*, 3(2):1–180, 2011.

[62] R. van der Meyden. Logical approaches to incomplete information: A survey. In *Logics for databases and information systems*, pages 307–356. 1998.

[63] S. Venkataraman, A. Panda, K. Ousterhout, M. Armbrust, A. Ghodsi, M. J. Franklin, B. Recht, and I. Stoica. Drizzle: Fast and adaptable stream processing at scale. In *SOSP*, pages 374–389, 2017.

[64] J. Widom, M. Theobald, and A. D. Sarma. Exploiting lineage for confidence computation in uncertain and probabilistic databases. In *ICDE*, pages 1023–1032, 2008.

[65] L. Xu, S. Huang, S. Hui, A. Elmore, and A. Parameswaran. OrpheusDB: A lightweight approach to relational dataset versioning. In *SIGMOD*, pages 1655–1658, 2017.

[66] Y. Yang, N. Meneghetti, R. Fehling, Z. H. Liu, and O. Kennedy. Lenses: An on-demand approach to etl. *PVLDB*, 8(12), 2015.

[67] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, pages 15–28, 2012.

[68] K. Zielnicki. Nodebook. https://multithreaded.stitchfix.com/blog/2017/07/26/nodebook/.