The Right Tool for the Job: Data-Centric Workflows in Vizier

Oliver Kennedy,⋆ Boris Glavic,⋄ Juliana Freire,† and Michael Brachmann ⋆
⋆ University at Buffalo, USA
⋄ Illinois Institute of Technology, USA
† New York University, USA

Abstract

Data scientists use a wide variety of systems with a wide variety of user interfaces such as spreadsheets and notebooks for their data exploration, discovery, preprocessing, and analysis tasks. While this wide selection of tools offers data scientists the freedom to pick the right tool for each task, each of these tools has limitations (e.g., the lack of reproducibility of notebooks), data needs to be translated between tool-specific formats, and common functionality such as versioning, provenance, and dealing with data errors often has to be implemented for each system. We argue that rather than alternating between task-specific tools, a superior approach is to build multiple user-interfaces on top of a single incremental workflow / dataflow platform with built-in support for versioning, provenance, error & tracking, and data cleaning. We discuss Vizier, a notebook system that implements this approach, introduce the challenges that arose in building such a system, and highlight how our work on Vizier lead to novel research in uncertain data management and incremental execution of workflows.

1 Introduction

Interactions with data, whether by experts or less technical users, are frequently iterative. As the user explores and transforms her data, she regularly backtracks to correct mistakes, extend her work, or to update source data as it evolves. Throughout this process, users employ a variety of tools, each providing a unique *data interaction modality*. For example, it is common for a data-centric workflow to involve some combination of a web browser (data discovery), a spreadsheet (quick statistics, minor data corrections, data entry), a database (batch manipulation and aggregation over large tabular data), a computational notebook (data visualization and model-building), a scripting IDE (modular component development), command-line tools (quick curation or summary tasks on simple data), and/or a business intelligence dashboard (data visualization).

Each tool provides a unique interaction modality designed for specific tasks and skill levels, but less effective at others. Thus, users working with data frequently string together workflows out of multiple tools, taking the most convenient for each task at hand. Such manual orchestration incentivizes bad habits that create challenges for reproducibility. It is also labor-intensive and error-prone because users have to translate data between the different formats expected by the tools they use, and must manually track provenance and manage documentation. Manual orchestration of data-centric workflows creates collections of workflow artifacts (e.g., datasets, code, and images) based on which it can be hard to (i) debug how a particular unexpected result was reached, or (ii)

Copyright 2022 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.
Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

reapply the workflow when source data changes or if an error has been discovered in the workflow and was fixed by modifying the workflow. Automating such features for workflows that span multiple modalities and systems is a challenging problem that typically entails building an overarching system for the orchestration of workflows and requires changes to the individual systems used in a workflow.

In this article, we outline the design of Vizier[1], an extensible, multi-modal platform for data-centric workflows. At the core of Vizier is a computational notebook, analogous to Jupyter or Zeppelin, that serves to orchestrate workflows. Vizier distinguishes itself from these tools in two ways. First, Vizier's modular architecture allows different interaction modalities to be built as views over the workflow encoded in the notebook. This includes simple views over individual cells or data artifacts that expose data interaction modalities like spreadsheets, as well as more complex modalities for data debugging, data documentation, and more. Second, Vizier's feature-rich workflow engine supports automatic refresh of stale outputs, fine-grained provenance features like propagation of information about uncertainty in data and data documentation through workflow steps, and supports workflows encoding notebooks where the dataflow between cells is only learned at runtime.

Vizier provides a simple workflow abstraction over which its extensible views are built: a workflow is a linear sequence of steps (called *cells* by analog to computational notebooks) and each step manipulates named, typed *artifacts*. Crucially, in Vizier, cells are isolated from each other and communicate only through a shared API. All inter-cell interaction takes place through cells reading and writing artifacts. This is in contrast to notebooks like Jupyter, where inter-cell interactions occur through the global state of an interpreter for a scripting language like Python. Vizier provides a streamlined mechanism for ensuring that workflow state (the versions of artifacts produced by the workflow) is re-computed when necessary. Each cell sees exactly those artifact versions that would be produced by executing all preceding cells in the order they appear in the notebook. The Vizier workflow engine also has built-in support for versioning workflows and artifacts. Every edit to a cell results in a new workflow version being created and associated with the corresponding versions of the artifacts it reads and creates. Both cells and artifacts are (conceptually) immutable and, as in functional data structures, an artifact or cell version can be shared by multiple versions of a workflow.

In what follows, we describe the Vizier workflow engine, as well as several views built on-top of this engine, including Vizier's notebook interface, a spreadsheet mode for interacting with datasets, a history viewer, and a summary view for data errors. We start with a motivating story of a user interacting with data.

1.1 User Story

Alice is organizing several hundred student volunteers for a multi-day online event. She asks volunteers to complete an online form with details including name, institutional email address, and time zone. At some point, many, although not all volunteers have completed the form and she is able to access the results as a downloadable CSV file. The event organizer has asked for a plot summarizing volunteer availability at different times of day, based on each participant's time zone. The availability window for students in adjacent time zones overlaps, making this a challenging problem for spreadsheets, but trivially solvable in SQL (given a table of time zone to availability mappings). Alice downloads the CSV file, ingests it into a relational database, runs the query, exports the result, and imports it into a spreadsheet to create a plot. It is convenient for Alice to use a mix of tools, because each tool is specialized for a particular subtask. However, she now has to manually manage multiple versions of her dataset across the tools.

> **Takeaway:** Data science benefits from chaining together multiple data interaction modalities, including but not limited to Exploration/Discovery, Direct Manipulation, and Querying/Scripting.

She then discovers that some volunteers have submitted multiple responses via the form. Automated curation libraries or stand-alone curation tools [22, 24] are usually sufficient to find and repair duplicate keys. However,

[1] `https://vizierdb.info`

the tool Alice selects is not able to handle several responses where both the original and revised form submissions contain distinct typos, a fact she does not realize until later.

> **Takeaway:** Many data curation tasks like key repair can be automated, but uncertainty arising from heuristics needs to be tracked and communicated to users [25, 37].

With each error corrected, Alice must return to the workflow and repeat all of the steps she previously performed so that she can generate a new plot that reflects these changes.

> **Takeaway:** Data workflows are developed iteratively; revisions to a prior step may require re-execution of subsequent steps to ensure that the results produced by these steps is up-to-date.

Alice's next task is to generate a credits page for the event's website, but she realizes that she has not asked the students to provide a home institution. Fortunately, using a Python script she can heuristically fetch institution names based on the domain name of the students' email address, by scraping the institution's website and retrieving the organization name from the web page metadata. Unfortunately, the script fails on a small number of unusual email addresses — for example some institutions have a separate root domain name for student emails. Writing a script that fixes each outlier individually would be more effort than it is worth, so Alice opens the script output in a spreadsheet and manually enters institutions for the missing students.

> **Takeaway:** The 80/20 rule applies to data science: uncommon error are often easier to fix manually.

She then feeds the resulting spreadsheet into a new script that generates a credits page. As the credits page is posted, a new batch of form entries roll in and several students issue corrections. Now, not only does Alice have to manually re-evaluate her workflow, she also needs to manually merge her edits (made in the spreadsheet) with the updated outputs — a tedious and error-prone process.

> **Takeaway:** Data science is continuous; workflows, whether containing manual steps or not, will inevitably need to be re-applied in new settings, e.g., when some of the input data is updated.

1.2 Desired Interaction Modalities

Users interacting with data employ a mix of tools at each stage of a data-centric workflow's life cycle: (i) *Discovery*, where users identify relevant datasets; (ii) *Exploration*, where users become familiar with the data's structure and semantics, and identify potential problems; (iii) *Curation*, where users address data problems, integrate data, and transform data into a desired form; (iv) *Analysis*, where users answer questions and/or build models; and (v) *Deployment*, where users refactor the workflow into a form suitable for automation and/or production use. We emphasize that progress through these stages is not monotone: users frequently identify errors or shortcomings, and go through a (vi) *Debugging* process that usually results in a return to an earlier stage of development.

Artifact Manipulation. The first four stages primarily concern themselves with data *artifacts* like datasets, machine learning models, and data visualizations. Depending on the specific task, users might interact with these artifacts through: (i) *Scripts* in languages like Python, R, Scala, or SQL which are general-purpose tools that excel at bulk transformations that follow specific patterns; (ii) *Direct Manipulation Interfaces* like spreadsheets that enable the user to directly inspect and manipulate the data, making them well suited for small datasets, quick one-off repairs, and data entry tasks; or (iii) *Automated Tools* or libraries like command-line utilities, data curation and cleaning algorithms, and data visualizers that each perform one specific task (e.g., the creation or transformation of an artifact) sufficiently well to cover most usage. All of these modalities act on one or more source artifacts, and produce one or more output artifacts as a result.

Workflow Inspection. All stages, but specifically the Debugging and Deployment stages, require a holistic view of the workflow: (i) *Workflow Management* provides users with a way to trace back through their exploration process whether through a full workflow management system like Vistrails [4], a notebook system like Jupyter, or a provenance capture system like CamFlow [33] or ReproZip [11]; (ii) *Versioning* through revision control systems like GIT, or through 'Track Changes' features allows users to trace back through time to earlier revisions of the data and workflows; (iii) *Automatic Data Documentation* tools like profilers, inference of semantic types, and uncertainty trackers help users to understand and develop specific artifacts in the context of the broader workflow [10,15,16,26,37]. (iv) *Debuggers* and IDEs like PyCharm, or separate tools like PigPen [31] give users a view of internal system state, helping them to trace forward through their code, and streamline the process of refactoring. A common theme to all of these modalities is that they provide a way for users to trace (whether forward or backward) through the execution of their workflow and its state.

1.3 Requirements

We now outline how the modalities exemplified in the prior section translate into concrete requirements for Vizier. To make our modularity goals concrete, we focus on extensibility in terms of support for new modalities for Artifact Manipulation, as well as Workflow Inspection.

Workflow Inspection. The first requirement is forced by the objective itself. At a minimum we need a way to record the steps the user took to achieve her goal.

Requirement W1: *Vizier must be able to capture the steps (workflow) the user follows to produce an output artifact.*

State, i.e., artifacts created / used by a workflow, plays an important part as well, and we need to track the evolution of state as it flows through the workflow (step versioning) and over the evolution of the workflow itself (workflow versioning).

Requirement W2: *Vizier must support versioned workflows and both step and workflow versioning of artifacts.*

For each version of a workflow, there exist corresponding versions of artifacts produced by Vizier's workflow semantics (that we will introduce in Section 1.4). Vizier needs to ensure that artifacts are automatically refreshed to reflect the latest version of the workflow to avoid bugs and non-reproducible workflows [34].

Requirement W3: *Vizier needs to identify (and update) artifacts invalidated by a revision to the workflow.*

Artifact Manipulation. To provide a viable alternative to using a mix of tools in a data-oriented workflow that each implement one particular modality, Vizier has to be a platform that can host a multitude of modalities.

Requirement A1: *Vizier must support multiple modalities for interacting with data artifacts*

The remaining requirements arise from the diviersity of modalities. Crucially, we wish to decouple modalities from the semantics of the artifacts they create or manipulate.

Requirement A2: *Vizier needs standard data formats and APIs for exchanging artifacts between modalities.*

We want to allow users and automated workflow steps to attach annotations to artifacts or their component parts as a way to document uncertainty or ambiguity arising from the data or its preparation. However, general purpose annotation management systems like Mondrian [18] or DBNotes [12] do not take any annotation-specific semantics into account. For example, we should not propagate an annotation marking a value A as uncertainty that is used in a computation $A \vee B$ if B is guaranteed to be true.

Requirement A3: *Vizier should support the efficient propagation of annotations on the component parts of an artifact through annotation-specific semantics including semantics for uncertainty management.*

Some modalities such as spreadsheets allow (manual) updates to data artifacts. If the original data changes, then it should be possible to automatically re-apply these edits to the modified data.

Requirement A4: *Manual updates to an artifact must be re-applicable when the artifact is updated upstream.*

1.4 Solution Overview

An overview of Vizier's architecture is shown in Figure 1. Addressing requirement **W1**, the central abstraction in Vizier is a workflow: a linear sequence of steps. Unlike classical workflow systems, Vizier does not require users to explicitly declare information flow between steps. Rather Vizier borrows the model employed in popular computational notebooks like Jupyter, where inter-cell communication occurs through a global state (artifacts) passed sequentially through steps. Following notebook conventions, we refer to these steps as *cells*, and the global state as a *scope*, a map from artifact name to the version of the artifact valid at this point in the workflow. Vizier stores artifacts in common formats through a versioned **Artifact Store** (Section 2), addressing requirement **A2**. In Section 3, we formalize Vizier's workflow model, and show how we satisfy requirement **W3** by instrumenting how each cell interacts with the scope, allowing us to determine what artifact versions are valid.

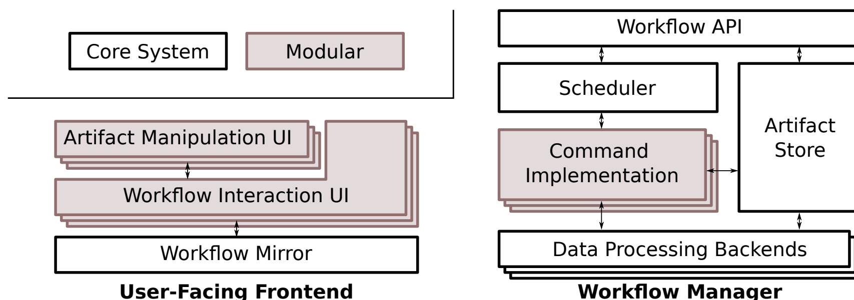

Figure 1: Vizier's architecture, comprised of a user-facing frontend component and a backend component.

Vizier's workflow semantics, paired with the versioned artifact store and workflow versioning (Section 3.2) addresses requirement **W2**. In contrast, classical notebooks like Jupyter or Zeppelin rely on the global state of an interpreter for inter-cell communication. Reverting this state to an earlier revision is challenging [39], limiting their ability to satisfy requirement **W3**. Vizier instead relies on its versioning system, allowing its **Scheduler** to automatically detect and re-evaluate stale cells (Section 3.3). To address requirement **A3**, we designed a light-weight uncertain data model that is implemented in Vizier in the form of *caveats*, annotations on data that indicate uncertain values and rows (Section 6).

Addressing requirement **A1** requires modularity in both Vizier's front- and back-end components. First, the user's interactions with a workflow and artifacts, whether through a scripting language, graphical interaction, or any other modality, need to be captured for replay (simultaneously addressing requirement **A4**). In Vizier this is achieved by requiring that every update to an artifact made through a particular modality has to be reflected as an operation in the workflow, i.e., a data update is translated into a workflow update. Vizier manages a collection of **Command Implementations** that implement the logic behind these artifact transformations (Section 4). To streamline the implementation of commands, Vizier's data formats and transformations are built over standard **Data Processing Backends** like Apache Spark.

The frontend is implemented over a **Workflow Mirror** that uses websockets to reflect a live view of the workflow the user is editing. Vizier automatically derives a default **Artifact Manipulation User Interface** for its notebook interface from each command's parameter schemas. This interface suffices for many templated commands, but the frontend can be further extended to provide a more customized experience, for example for Spreadsheet-style direct manipulation of data (Section 5). As illustrated in Figure 2, the frontend displays three **Workflow Interaction User Interfaces** by default: (i) A direct display of the workflow as a notebook, (ii) a table of contents summary of the notebook, including highlighting from documentation, and (iii) a list of artifacts derived by the notebook. Several of these components, including the notebook and the artifact list provide access to direct manipulation interfaces. Additional views currently implemented in Vizier include: (iv) A caveat view (Section 6) that shows and tracks potential errors in the workflow and data, (v) a history view that shows the evolution of the workflow over time, and (vi) a data provenance subway diagram view.

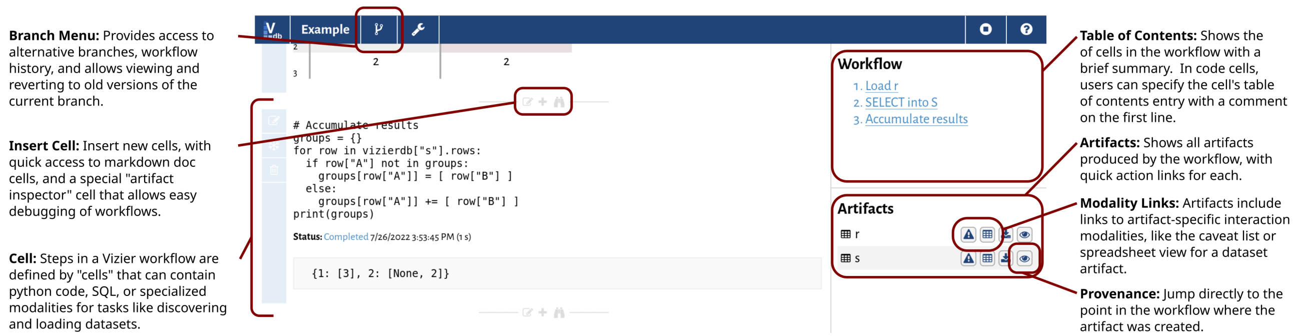

Figure 2: The Vizier User Interface

2 Versioned Data Artifact Store

Steps in Vizier workflows create, read, and update *"data artifacts"* or artifacts for short. To maximize interoperability between cells, Vizier establishes standards for how these artifacts are serialized, which we now discuss. Versions of artifacts are immutable. An update to an artifact creates a new object representing the updated artifact. Immutability greatly simplifies handling of state in Vizier and enables us to share artifact versions across multiple revisions of a workflow.

Dataset Artifacts. Tabular data is represented by Vizier as a Spark dataframe [3], a logical encoding of how a dataset is derived from source data. The choice to store datasets through a logical representation (specifying the computation instead of its result) is driven largely by the need for managing annotations (requirement **A3** which is implemented by rewriting the computation to handle annotations), but also results in lower space consumption.

Parameter Artifacts. A parameter artifact can be any primitive value of any data type supported by Apache Spark; We note that this includes simple nested collections like Arrays and Maps. This type of artifact provides a way to parameterize scripts and other system components, particularly for non-technical users. and parameter artifacts are used to pass simple data (e.g., simple constants in Python) between cells.

In addition to these two artifact types, Vizier also supports plots (stored in the vegalite [36] format), blobs and uninterpreted files, and language-specific constructs, e.g., Python function definitions. For lack of space, we do not further discuss these other artifact types.

3 Workflow Provenance Model and Runtime

In this section, we outline Vizier's workflow and versioning model, how we determine which cells of a workflow need to be re-executed if a cell is changed, and introduce the parallel scheduler of the system.

3.1 Workflow Model

A workflow in Vizier is a sequence of workflow steps (cells) that can read and write artifacts. Artifacts are versioned at the granularity of cells. A cell writing an artifact a causes a new version of a to be created. We will discuss the APIs Vizier exposes to the data interaction modalities for accessing artifacts in Section 4. The semantics of a Vizier workflow is the serial execution of the cells of the workflow, where the latest version of each artifact is passed as input to the cell that will be executed next. Thus, as long as the cells themselves are deterministic, the artifact versions created by the execution of a workflow are uniquely determined by the workflow itself.

```
data = read_csv("social_data.csv")
show(data)
```

```
data["latlon"] = geocode(data["address"])
show(data)
```

```
censusblocks = read_geojson("blocks.json")
show(censusblocks)
```

```
data = spatial_join(data, censusblocks, ["latlon", "geometry"])
count_per_block = data.groupby("block_id").count()
show(count_per_block)
```

Figure 3: A simplified example notebook

Example 3.1: Figure 3 shows a notebook with a simplified data ingestion, cleaning, and analysis task. The notebook loads the dataset (cell 1), geocodes listed street addresses (cell 2), loads a collection of census blocks (cell 3), and computes summary statistics for each census block (cell 4). We use this notebook to illustrate a key challenge of traditional notebook architectures. The user modifies cell 2 to switch to a different geocoder, potentially requiring re-evaluation of the notebook. In this specific notebook, the user had the foresight to write in an idempotent style, making it unnecessary to re-run cell 1 to recover the state needed to run cell 2 correctly. It is also unnecessary to re-run cell 3, as it does not depend on the output of cell 2. However, in traditional notebooks like Jupyter the burden of deciding which cells to re-evaluate rests on the user, creating added overhead and increasing the chance of errors.

Formally, a Vizier workflow is a sequence $\mathcal{N} = \{c_1, \ldots, c_N\}$ of cells. The versions of artifacts valid after execution of cell c_i are modeled as a global scope $\mathcal{G}$ that maps artifact names to artifact versions or the distinguished symbol $\bot$, which indicates that no version of the artifact has been produced yet. A cell c is a function that takes a scope $\mathcal{G}$ as input and produces an updated scope $\mathcal{G}'$: $c(\mathcal{G}) = \mathcal{G}'$. We use $\vec{r}_{\mathcal{G},i}$ to denote the names of artifacts accessed by cell c_i applied to $\mathcal{G}_i$. The scope parameter is necessary, because a cell may dynamically decide which artifacts to read based on the content of other artifacts. The result $[\![\mathcal{N}]\!]$ of evaluating workflow $\mathcal{N} = \{c_1, \ldots, c_N\}$ is defined as the scope $\mathcal{G}_n$ produced by starting with an empty scope $\mathcal{G}_0$ (where all artifact names are mapped to $\bot$), and by computing $\mathcal{G}_i = c_i(\mathcal{G}_{i-1})$.

If workflow $\mathcal{N}$ is modified by replacing a cell c_i with a cell c'_i (denoted as $\mathcal{N}[c_i \backslash c'_i]$), we need to obtain the updated scope $[\![\mathcal{N}[c_i \backslash c'_i]]\!]$. Of course this can be achieved by evaluating $\mathcal{N}[c_i \backslash c'_i]$. However, to improve performance, Vizier attempts to update the output of $[\![\mathcal{N}]\!]$ by only re-evaluating a subset of the cells. First, observe that $\mathcal{G}_1, \ldots, \mathcal{G}_{i-1}$ are independent of c_i; and remain unchanged if c_i is modified. It is still necessary to have $\mathcal{G}_{i-1}$ to evaluate c'_i; so Vizier caches all intermediate global scopes after evaluating each workflow revision.

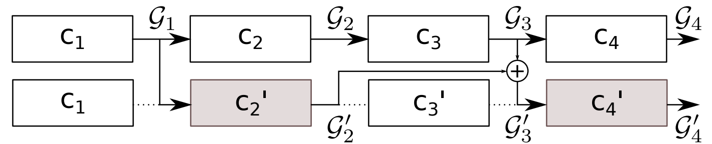

Figure 4: Evaluation logic for the example notebook in Figure 3. Edges are labeled with global scope versions. Cells that are (re-)evaluated are highlighted, dotted lines represent simulated state flow.

Naively, we have to still re-evaluate $c_{i+1}, \ldots, c_N$ using the new scope produced by c'_i. Let us denote the scopes produced by this evaluation as $\mathcal{G}'_{i+1}, \ldots, \mathcal{G}_n$. Vizier uses the readsets $\vec{r}_{\mathcal{G},i+1}, \ldots, \vec{r}_{\mathcal{G},N}$, to identify cells c_j for which the same output (artifact versions written by the cell) in $[\![\mathcal{N}[c_i \backslash c'_i]]\!]$ is guaranteed. Denote by $\Delta(\mathcal{G}, \mathcal{G}') = \{k | \mathcal{G}(k) \neq \mathcal{G}'(k)\}$ the names of artifacts that differ between $\mathcal{G}$ and $\mathcal{G}'$. We have to re-execute cell

c_{i+1} if an artifact read by c_{i+1} has changed, which is the case if $\Delta(\mathcal{G}_i, \mathcal{G}'_i) \cap \vec{r}_{\mathcal{G},i+1} \neq \emptyset$. If c_{i+1} needs to be re-executed, then we set $\mathcal{G}'_{i+1} = c_{i+1}(\mathcal{G}'_i)$. Otherwise, we generate $\mathcal{G}_{i+1}$ by merging the changes made by c_{i+1} in $\mathcal{N}$ to $\mathcal{G}_i$ into $\mathcal{G}'_i$. That is, for all artifacts k we set $\mathcal{G}'_{i+1}(k) = \mathcal{G}_{i+1}(k)$ if $k \in \Delta(\mathcal{G}_i, \mathcal{G}_{i+1})$ and $\mathcal{G}'_{i+1}(k) = \mathcal{G}'_i(k)$ otherwise. The same approach is applied to decide whether to evaluate or skip the remaining cells in $\mathcal{N}[c_i \backslash c_i{}']$.

Example 3.2: Figure 4 continues the running example from **??** 3.1. The user has replaced the initial version of cell c_2 with an updated version $c_2{}'$. The global scope produced by preceding cells ($\mathcal{G}_1$) may be re-used unchanged to evaluate $c_2{}'$, producing scope $\mathcal{G}'_2$. $\Delta(\mathcal{G}_2, \mathcal{G}'_2)$ is the `data` artifact, but the readset of c_3 is empty, and so this cell does not need to be re-evaluated. Instead $\mathcal{G}'_3$ is derived by merging variables the cell previously changed (i.e., $\Delta(\mathcal{G}_2, \mathcal{G}_3)$) into the current global scope. Finally, Vizier identifies that an element of $\vec{r}_{\mathcal{G},4}$ (the `data` variable) has changed between $\mathcal{G}_3$ and $\mathcal{G}'_3$, necessitating a re-evaluation of cell 4.

3.2 Versioning Workflows

Vizier maintains a branching history of the evolution of the notebook. A cell is further subdivided into (i) **cell metadata** (c_i) that is unique to the workflow (e.g., timestamps and execution status), (ii) **results** (r_i) of executing the cell, and (iii) a **module** (m_i) describing the command executed in the cell. The latter two components (results and modules) may be shared across workflows. A module object describes the command to be evaluated in a cell. This includes its type (e.g., Python or Scala script; SQL query; or one of the graphical widgets), as well as any parameters to the command (e.g., the script itself, or the artifact name to materialize query results as). A results object stores references to versions of artifacts in the artifact store created by the execution of the cell. We do not materialize the full global scope after each cell, but rather only the changes made to the global scope by the cell (i.e., the cell's write set). Any global scope $\mathcal{G}_i$ can be reconstructed from the preceding write sets.

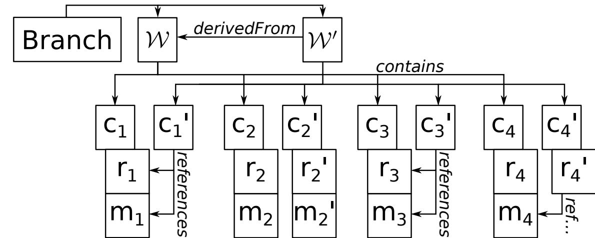

Figure 5: History for the running example, assuming cell 2 was changed as in Figure 4.

Example 3.3: Figure 5 continues our running example, which in our new terminology is a single branch comprised of two workflows $\mathcal{W}_1$ and $\mathcal{W}_2$. Both notebooks are comprised of four cell objects each. Cells 1, 3, and 4 were unchanged between workflows, and so the corresponding cell objects for $\mathcal{W}_2$ reference the same module description object as the corresponding cell for $\mathcal{W}_1$, while the module descriptions referenced by c_2 and $c_2{}'$ differ. Cells 1 and 3 do not need to be re-evaluated, and so can share their result objects, while Cells 2 and 4 both require re-evaluation and create new result objects.

Workflow and Branch History. Similar to version control systems like git, each workflow version in Vizier stores a reference to the workflow version it was derived from. A branch is an append-only sequence of workflow versions. The branch contains a reference to the most recent workflow version in the branch. A new branch may be created from any existing workflow version, including a historical one.

3.3 Parallel Scheduler

The evaluation model presented in Section 3.1 relies exclusively on *dynamic provenance*, where Vizier is informed about the readset and writeset of a cell at runtime when the cell accesses artifacts through Vizier's API. However, dynamic provenance is not always sufficient. For example, Vizier evaluates cells in parallel where possible, but dynamic provenance is not available until the cell has already been evaluated. Static provenance, which can derive a cell's read and write sets through static dataflow analysis of its source code, can be computed upfront.

However, static dataflow analysis is of necessity an approximation for some cell types; language features like control flow and dynamic code evaluation can lead to over- (or under-) estimates of the cell's read and write sets. Vizier uses a novel approach which refines static provenance at runtime using dynamic provenance [14].

Fundamentally, Vizier's scheduler needs to assign each cell in a running workflow to one of four lifecycle stages: (i) **DONE** when the cell has a valid result object, (ii) **PENDING** when the cell depends (directly or transitively) on a cell that does not have a result object, and (iii) **RUNNING** when the cell's dependencies have a valid result object but the cell itself does not. We further distinguish as (iv) **STALE** those cells in the **PENDING** stage for which we can conclusively determine that re-evaluation is required.

To manage lifecycle transitions, Vizier's scheduler relies on a combination of static and dynamic provenance [14]. It uses static provenance to generate an over-approximation on the read and write dependencies of a cell[2]. Accordingly, the scheduler tracks an over-approximation of the read and writesets at each step of the workflow, and refines them when the execution of cell finishes and we know its precise read and write set. This approximation is used by Vizier's scheduler to omit cells from re-execution or schedule them for parallel execution if we can determine that it is safe to do so based on these over-approximations.

4 Multimodality

As noted in Section 3.2, cells in Vizier implement a variety of different modalities, including scripting languages (e.g., Python, Scala), query languages (e.g., SQL), as well as graphical widgets for data ingestion, transformation, and curation (e.g., Load Dataset, Pivot Table, Repair Key). We refer to the evaluation logic for each modality as *cell command*. Each cell in a Vizier workflow identifies the command that should run to evaluate the cell, along with a set of arguments to that command. For example, the SQL cell (`sql.query`) takes two arguments: the text of a SQL query, and an optional name to materialize the result table as. In this section, we discuss how these modalities are implemented as cell types in Vizier.

A command is defined by the following methods: (i) **schema**: Returns a schema for the arguments the command accepts; (ii) **summary(arguments)**: Returns a textual description of the behavior of the command when parameterized by the provided arguments; (iii) **dependencies(arguments)**: Returns the over-approximation of the set of names of artifacts read (resp., written) by the cell as parameterized by the provided arguments, or indicates that either or both bounds can not be computed; and (iv) **evaluate(arguments, context)**: Evaluates the cell on the command, parameterized by the provided arguments and a context object.

The context object on which commands are evaluated provides a read/write interface to the global scope and a way to emit messages to be displayed alongside the cell in the notebook view. As we discuss in Section 3, the global scope maps artifact names to artifact versions. Artifacts are not persisted directly as part of the global scope, but rather are references to our write-only artifact store.[3] When a cell implementation writes an artifact through the context, the artifact version is serialized and written into the artifact store. The artifact store assigns the artifact (version) a unique identifier, which is then saved into the scope. Similarly, to read an artifact, a cell implementation first reads the artifact identifier out of the scope, and then accesses the corresponding artifact version from the store.

Cells implementing runtimes for general-purpose languages need to provide users of those languages with a way to interact with the global notebook state. This entails (i) providing a mechanism to reference the scope and import artifacts within a language-specific format, and (ii) predicting how the user-provided code will interact with the state to bound provenance.

[2] As we argue in [14], to deal with languages that support dynamic code evaluation such as Python, it would be necessary to allow under-approximations of read and write sets (missed data dependencies) and compensate for them at runtime. However, this not implemented in Vizier yet; attempts to dynamically read variables not in the maximal readset are flagged as errors.

[3] As mentioned before, artifact versions are immutable in Vizier which simplifies version management. We leave optimizations that selectively violate this policy and update artifacts or store deltas instead of creating full updated versions of artifacts to future work.

Python. Python cells are evaluated in an independent interpreter to avoid concurrency bottlenecks from the global interpreter lock (GIL). Thus, a key challenge is minimizing the volume of state transferred into and out of each interpreter. Global scope is lazily loaded by populating the global state with a set of proxy objects, one for each artifact in the scope. Access to the Vizier context for messaging and artifact access occurs through a control bus that, by default, operates over the python process' standard input and output streams. Vizier defines a special 'show' command within the module to produce structured messages (analogous to placing an item on the last line of a Jupyter cell). The show command includes support for most Vizier-defined types, matplotlib and bokeh plots, and provides fallbacks using the Jupyter-standard `_repr_html_` method, or direct stringification as a final resort. For predictive dependency tracking, and to identify variables that are modified, Vizier relies on Python's `ast` module, which provides an introspective compilation and code analysis framework. Vizier performs lightweight dependency analysis [14] to bound the cell's read and write dependencies.

SQL. SQL cells are parsed and evaluated by Apache Spark. Vizier intercepts the parsed SQL query AST and manually injects references to the corresponding artifacts — either datasets or python functions — where needed. Spark-provided view name decorators make it possible for the injected views to retain their names from the query, avoiding incomprehensible error messages. The same injection logic is also used to statically identify exact read and write dependencies.

5 Interactive Spreadsheets

A key feature of Vizier is support for direct interaction with artifacts [10], most notably a spreadsheet-like interface for interacting with Spark Data Frames. Spreadsheets provide a data exploration experience that is distinct from notebooks. Users are limited to a single dataset, but have significantly more flexibility when exploring the data. For example, it is common on small datasets (e.g., under 1000 rows) for users to complete preliminary data cleaning tasks like outlier detection, data integration, repair of typos and outliers, and even some limited computation (e.g., deriving new fields) in a spreadsheet prior to working with the data further (e.g., in a notebook). Another common use case is manual data entry; The user may enter the entire dataset, or may generate a data entry template (e.g., with a script) and import the resulting file into another tool.

Vizier's spreadsheet interface is intended to provide a view over a subset of the notebook, allowing users to interact with the dataset in the appropriate modality, while simultaneously preserving workflow provenance through interactions. As the user interacts with the spreadsheet, their edits are reflected in the notebook as new cells. As the user edits the notebook, their edits are likewise reflected in the spreadsheet — If the source data frame is updated in the notebook, Vizier attempts to re-apply the user's edits to the updated data.

Track Changes as a View. To support bi-directional interaction between spreadsheet and notebook views, Vizier implements a series of specialized cell types that mimic SQL DDL and DML operations, allowing for inserting, deleting, or reordering columns and rows, or for updating individual cell values. We refer to the operations described by these cell types collectively as the Vizual language [10, 17].

Vizual is based on the principle that the user's interactions with a database can be modeled as views over an original version of the data. As prior work has shown, a view defined over a table can mimic the effects of any DDL [13] or DML [30] operation applied to the table. Analogously, each Vizual cell in the notebook uses Spark's standard data frame manipulation language to apply a successive transformation to the dataset that mimics the user's interaction. To ensure that the spreadsheet remains responsive, a shim layer tentatively injects predicted updates to the user's interactions until the effects of the user's edit are fully applied (e.g., as [20]).

In SQL DML, update operations specify target rows by a predicate. By contrast, operations in a spreadsheet explicitly target specific rows of data, requiring Vizier to assign unique identifiers to each record to encode their order in the spreadsheet[4]. To allow Vizual operations to be replayed as source data changes, these identifiers

[4] We assume that the number of columns will remain manageable and reference them purely by name

should remain stable through data transformations. For derived data, Vizier uses a row identity model similar to GProM's [2] encoding of provenance. Derived rows, such as those produced by declaratively specified table updates, are identified by an appropriate combination of input tuple identifiers. For example, rows in the output of a join are identified by combining identifiers from the source rows that produced them into a single identifier, and rows in the output of a projection or selection use the identifier of the source row that produced them.

The remaining challenge is assigning row identifiers to source data, which we want to remain stable through changes to the source data so that spreadsheet operations can be replayed. Ideally, the data would include a unique identifier that we can leverage; but this is not always the case. Storing the data in a revision control system [9, 21], is not always a viable option [1]. A more heavyweight approach is to link records across revisions of a dataset [38], but this adds non-negligible overhead to common-case data revisions. Vizier presently supports persistent identifiers through append- or edit-only revisions by assigning each record a unique identifier based on its position, and a hash of its contents. This approach has the benefit of being lightweight (it can be applied in a single pass), and resilient. In contrast to simply using a hash-based identifier, the approach supports duplicate records. Conversely, solely using a position-based identifier could lead to spreadsheet operations being applied to the wrong row in case of insertions. While techniques for creating identifiers that are stable under updates has been studied extensively for XML databases (e.g., ORDPATH [32]) and recently also for spreadsheet views of relational databases [5], the main challenge we face in Vizier is how to retain row identity when a new version of a dataset is loaded into Vizier, as opposed to keeping identity consistent once the data is already in the system.

6 Data Documentation, Error, and Uncertainty Management with Caveats

Like notebook systems, Vizier enables users to document their workflow through markdown cells which do not manipulate artifacts, but simply serve as documentation. However, some documentation is specific to individual artifacts, or their component parts (e.g., rows, or columns); We would like such documentation to accompany the data as it is transformed [26]. Like other annotation management systems including Mondrian [18] and DBNotes [8], Vizier empowers users to annotate data with textual comments. However, in contrast to these systems, in Vizier these annotations also have a precise semantics: they encode uncertainty about an attribute value of a row or the existence of a row. This is important, because uncertainty arises naturally in most data science pipelines (e.g., because of errors in the data or because of heuristic choices during data cleaning) and if data analysis ignores the uncertainty in the data, it can lead to analysis results that cannot be trusted. To address this issue, caveats are propagated through operations on dataset artifacts in Vizier using an efficient uncertain query semantics we have developed [15, 16]. Thus, caveats on values and rows in the result of an analysis conducted using Vizier encode information about how data cleaning and curation operations on the data used in the analysis affect the analysis result. Furthermore, Vizier's implementation of data cleaning operations introduce caveats to encode information about other possible repairs.

6.1 Incomplete Databases

Formally, caveats in Vizier are based on an approximation of incomplete databases. An incomplete database $\mathcal{D} = \{D_1, \ldots, D_n\}$ is a set of deterministic databases called possible worlds that encode alternative possibilities for the state of the real world: one possible world corresponds to the actual state of the real world, but we do not know which. As an example, consider an analyst that has to find the names of important customers (e.g., who ordered products totaling more than $500). An example instance is shown in Figure 6. A common method for primary key repair is to group rows by their PK values and select one row from each group to be retained. However, typically we have insufficient information to know which row is the correct choice and will have to rely on heuristics (e.g., selecting the most recently updated row if this information is available). Incomplete databases can be used to model this uncertainty: we create an incomplete database whose worlds are all the repairs of the database violating the constraint [7]. Figure 6 shows two (out of 4) possible repairs for this dataset.

Customer Relation

cid	name	total
1	Peter Petersen	1000
1	Peter Petersen	950
2	Bob Smith	300
3	Alice Smith	400
3	Alice Smith	600

Possible World (Repair) D_1

cid	name	total
1	Peter Petersen	1000
2	Bob Smith	300
3	Alice Smith	400

Possible World (Repqir) D_2

cid	name	total
1	Peter Petersen	1000
2	Bob Smith	300
3	Alice Smith	600

Certain Answers

name
Peter Petersen

Answers in D_1

name
Peter Petersen

Answers in D_2

name
Peter Petersen
Alice Smith

Figure 6: Example customer database violating the primary key constraint that cid is unique and two possible worlds corresponding to some of the possible repairs of the database achieved by selecting one row among each group of rows with the same primary key value.

Typical constraint-repair algorithms will select one repair (one possible world) based on a heuristic like selecting the row whose values are most common in the dataset [35]. For instance, the cleaning algorithm may choose D_1 and the user would then evaluate their query (shown below) over D_1.

```
SELECT name FROM Customer WHERE total > 500;
```

While such heuristics may be quite effective on average and are certainly superior to just randomly selecting a world, it is unavoidable that they fail for some cleaning scenarios. An alternative approach called consistent query answering [6] takes a conservative stance, instead of selecting one repair, we reason about all possible repairs and only return query answers (the so-called *certain answers*) that are in the query's results for every repair (i.e., are guaranteed to be in the result independent of which repair is correct). This approach has the advantage that only correct query answers are returned, but is computationally expensive, may exclude many very likely answers (if they are not 100% certain), and is not closed (it is not possible to evaluate queries with certain answer semantics over the certain answers of a query).

REPAIR KEY COLUMN 0

Status: Completed 7/27/2022 10:30:38 AM (6 s)

```
Building Fix Key Column lens on customer...
Saving results...
```

customer

	cid (int)	name (string)	total (int)
1	1	Peter Petersen	1000 (?)
2	3	Alice Smith	400 (?)
3	2	Bob Smith	300

Figure 7: A UB-DB in Vizier encoding D_1 with possibly uncertain cells marked with caveats.

- 1000 could be one of 1 other distinct values for "customer".total when "customer".cid = 1, including 950
- 400 could be one of 1 other distinct values for "customer".total when "customer".cid = 3, including 600

OK

Figure 8: Caveats annotating the relation D_1

6.2 Attribute- and Row-level Caveats and Uncertainty-Annotated Databases

For Vizier, we developed an uncertain data model called uncertainty-annotated databases [15] that annotates one possible world (the so-called *selected-guess world*) with an under-approximation of certain answers (if we claim that a row is certain, then it is certain) that can be computed efficiently. This is encoded by annotating a subset of a dataset's rows with row-level caveats to mark them as not being certain. The reason that we use an under-

approximation is to be able to evaluate complex queries (full relational algebra including aggregation) efficiently (with PTIME data complexity and small overhead over deterministic query processing). Furthermore, attribute-level caveats are used to mark attribute values as uncertain (they may not be the same in every possible world) which is similar in nature to certain answers with nulls [27].

7 Conclusions and Future Work

In this paper, we have made the case for a multi-modal data science platform built on top of an incremental, data-centric workflow engine and have introduced the reader to Vizier, our system implementing this vision. Because each modality (notebooks, spreadsheets, the caveat view, etc...) is a "view" interacting with the same underlying workflow and datasets, it is easy to extend the system to support new interaction paradigms in the future. Building our system so that cells execute in isolation and only interact through dataflow makes it easy to add new cell types to the system, because we only have to worry about the interaction of the new cell type with data artifacts. Thus, adding support for other interaction modalities (e.g., new programming languages) into the system is straight-forward: we implement a new cell type and an API to access Vizier artifacts from within the modality.

As demonstrated in recent preliminary work, having a scheduler that revises its schedule based on dependencies discovered at runtime and using static dataflow analysis, we can execute notebook cells in parallel and avoid re-executing cells that are guaranteed not to change during automatic refresh. Significantly more opportunities for avoiding work exist, in particular by leveraging Vizier's standardized representation of artifacts. For example, by representing updates to dataset artifacts as change sets, existing approaches for incremental view maintenance can reduce the runtime overhead of recomputing workflow steps, as well as the space costs of preserving multiple artifact versions. Another interesting direction for future work is to study incremental maintenance of workflow results when the definition of a workflow step changes. This is an novel variation of the traditional incremental view maintenance problem where we have to update the result based on a change to the query rather than based on a change to the data.

Similarly, standard representations of artifacts can be used to help users better understand the outputs of their workflows. For example, numerous efforts have explored causal explanations in database query results ([19, 23, 28, 29], and explainability in machine learning has recently become an area of active research. For such techniques to be truly valuable, they need to operate across artifact types. For example, what would it take to link a sudden change in predictions made by a model to the addition of a new category in source data five transformations removed from the model training step. With Vizier's uncertainty model and provenance tracking we lay the ground work to develop methods for generating explanations that span multiple steps in a workflow.

Finally, we note that Vizier provides a "ground-up" approach to stitching different interaction modalities into a single workflow. Tools implemented as views over Vizier's workflow model seamlessly interact with each other, with coarse-grained provenance, reactive cell execution, and repeatability/reproducibility. However, these capabilities are limited to a single Vizier instance, and are of limited value to already existing tools. An important open challenge is the design of a federated infrastructure for data analytics workflows, allowing multiple Vizier instances or unrelated tools to interoperate.

Acknowledgements. This work was supported by NSF Awards ACI-1640864, IIS-1750460, IIS-1956149, and IIS-2125516.

References

[1] I. Alagiannis, R. Borovica, M. Branco, S. Idreos, and A. Ailamaki. Nodb: efficient query execution on raw data files. In *SIGMOD Conference*, pages 241–252. ACM, 2012.

[2] B. S. 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.

[3] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia. Spark sql: Relational data processing in spark. In *Proceedings of the ACM International Conference on Management of Data (SIGMO)*, pages 1383–1394. ACM, 2015.

[4] 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. IEEE Computer Society, 2005.

[5] M. Bendre, B. Sun, D. Zhang, X. Zhou, K. C. Chang, and A. G. Parameswaran. DATASPREAD: unifying databases and spreadsheets. *Proc. VLDB Endow.*, 8(12):2000–2003, 2015.

[6] L. Bertossi. Database repairing and consistent query answering. *Synthesis Lectures on Data Management*, 3(5):1–121, 2011.

[7] G. Beskales, I. F. Ilyas, L. Golab, and A. Galiullin. Sampling from repairs of conditional functional dependency violations. *VLDB J.*, 23(1):103–128, 2014.

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

[9] A. P. Bhardwaj, S. Bhattacherjee, A. Chavan, A. Deshpande, A. J. Elmore, S. Madden, and A. G. Parameswaran. Datahub: Collaborative data science & dataset version management at scale. In *CIDR*. www.cidrdb.org, 2015.

[10] M. Brachmann, W. Spoth, O. Kennedy, B. Glavic, H. Müller, S. Castel, C. Bautista, and J. Freire. Your notebook is not crumby enough, replace it. In *Proceedings of the 10th Conference on Innovative Data Systems*, 2020.

[11] F. S. Chirigati, D. E. Shasha, and J. Freire. Reprozip: Using provenance to support computational reproducibility. In *TaPP*. USENIX Association, 2013.

[12] L. Chiticariu, W. C. Tan, and G. Vijayvargiya. Dbnotes: a post-it system for relational databases based on provenance. In *SIGMOD Conference*, pages 942–944. ACM, 2005.

[13] C. Curino, H. J. Moon, and C. Zaniolo. Graceful database schema evolution: the PRISM workbench. *Proc. VLDB Endow.*, 1(1):761–772, 2008.

[14] N. Deo, B. Glavic, and O. Kennedy. Runtime provenance refinement for notebooks. In *Proceedings of the 14th International Workshop on the Theory and Practice of Provenance*, 2022.

[15] S. Feng, A. Huber, B. Glavic, and O. Kennedy. Uncertainty annotated databases - a lightweight approach for approximating certain answers. In *Proceedings of the 44th International Conference on Management of Data*, 2019.

[16] S. Feng, A. Huber, B. Glavic, and O. Kennedy. Efficient uncertainty tracking for complex queries with attribute-level bounds. In *Proceedings of the 46th International Conference on Management of Data*, page 528 – 540, 2021.

[17] J. Freire, B. Glavic, O. Kennedy, and H. Müller. The Exception that Improves the Rule. In *SIGMOD Workshop on Human-In-the-Loop Data Analytics*, 2016.

[18] F. Geerts, A. Kementsietsidis, and D. Milano. MONDRIAN: Annotating and Querying Databases through Colors and Blocks. Technical report, University of Edinburgh, 2005.

[19] B. Glavic, A. Meliou, and S. Roy. Trends in explanations: Understanding and debugging data-driven systems. *Found. Trends Databases*, 11(3):226–318, 2021.

[20] N. Gupta, A. J. Demers, J. Gehrke, P. Unterbrunner, and W. M. White. Scalability for virtual worlds. In *ICDE*, pages 1311–1314. IEEE Computer Society, 2009.

[21] S. Huang, L. Xu, J. Liu, A. J. Elmore, and A. G. Parameswaran. Orpheusdb: Bolt-on versioning for relational databases. *Proc. VLDB Endow.*, 10(10):1130–1141, 2017.

[22] I. F. Ilyas and X. Chu. Trends in cleaning relational data: Consistency and deduplication. *Found. Trends Databases*, 5(4):281–393, 2015.

[23] B. Kanagal, J. Li, and A. Deshpande. Sensitivity analysis and explanations for robust query evaluation in probabilistic databases. In *SIGMOD Conference*, pages 841–852. ACM, 2011.

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

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

[26] P. Kumari, M. Brachmann, O. Kennedy, S. Feng, and B. Glavic. Datasense: Display agnostic data documentation. In *CIDR*, 2021.

[27] L. Libkin. Sql's three-valued logic and certain answers. *ACM Transactions on Database Systems (TODS)*, 41(1):1, 2016.

[28] A. Meliou, W. Gatterbauer, K. F. Moore, and D. Suciu. The complexity of causality and responsibility for query answers and non-answers. *Proc. VLDB Endow.*, 4(1):34–45, 2010.

[29] A. Meliou, S. Roy, and D. Suciu. Causality and explanations in databases. *Proc. VLDB Endow.*, 7(13):1715–1716, 2014.

[30] 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. *Proc. VLDB Endow.*, 10(12):1857–1860, 2017.

[31] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD Conference*, pages 1099–1110. ACM, 2008.

[32] 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 Conference*, pages 903–908. ACM, 2004.

[33] T. F. J. Pasquier, X. Han, M. Goldstein, T. Moyer, D. M. Eyers, M. I. Seltzer, and J. Bacon. Practical whole-system provenance capture. In *SoCC*, pages 405–418. ACM, 2017.

[34] J. F. Pimentel, L. Murta, V. Braganholo, and J. Freire. A large-scale study about quality and reproducibility of jupyter notebooks. In M. D. Storey, B. Adams, and S. Haiduc, editors, *Proceedings of the 16th International Conference on Mining Software Repositories, MSR 2019, 26-27 May 2019, Montreal, Canada.*, pages 507–517. IEEE / ACM, 2019.

[35] T. Rekatsinas, X. Chu, I. F. Ilyas, and C. Ré. Holoclean: Holistic data repairs with probabilistic inference. *PVLDB*, 10(11):1190–1201, 2017.

[36] A. Satyanarayan, D. Moritz, K. Wongsuphasawat, and J. Heer. Vega-lite: A grammar of interactive graphics. *IEEE Trans. Vis. Comput. Graph.*, 23(1):341–350, 2017.

[37] Y. Yang, N. Meneghetti, R. Fehling, Z. H. Liu, and O. Kennedy. Lenses: an on-demand approach to etl. *Proceedings of the VLDB Endowment*, 8(12):1578–1589, 2015.

[38] G. S. Yilmaz, T. Wattanawaroon, L. Xu, A. Nigam, A. J. Elmore, and A. G. Parameswaran. Datadiff: User-interpretable data transformation summaries for collaborative data analysis. In *SIGMOD Conference*, pages 1769–1772. ACM, 2018.

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