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

ABSTRACT

Notebook and spreadsheet systems are currently the defacto standard for data collection, preparation, and analysis. However, these systems have been criticized for their lack of reproducibility, versioning, and support for sharing. We present Vizier, an open-source tool that helps analysts to iteratively build and refine data pipelines that combines the flexibility provided by notebooks with an easy-to-use direct-manipulation interface of spreadsheets. These, combined with advanced provenance tracking for both data and computational steps enables reproducibility and streamlines data exploration.

Keywords

Notebooks, Data Science, Provenance, Workflow System

1. INTRODUCTION

Notebook tools like Jupyter have emerged as a popular programming abstraction for data exploration, model-building, and rapid prototyping. Notebooks promise re-use and reproducibility. However, a recent study by Pimentel et. al. [10] 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 — but rather as library managers for code snippets (i.e., cells). Reproducibility and re-usability 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) [7].

In this paper, we present Vizier,[1] a notebook-style data exploration system designed from the ground up to encourage notebook reproducibility and re-use. Vizier eschews REPLs in favor of a more powerful state model: A versioned relational database and workflow. This change allows Vizier to act as a true workflow manager [3], precluding out-of-order execution, a common source of frustration[2] and non-reproducible workflows [10]. To aid reproducibility, Vizier maintains a full version history for each notebook. Vizier supplements its notebook with a tightly coupled "spreadsheet mode" that allows reproducible direct manipulation of data. Finally, Vizier facilitates debugging by tracking potential data errors through a principled form of incompleteness annotations that we call *caveats*.

Many aspects of Vizier, including parts of its user interface [6, 8], provenance models [1, 3], and caveats [11, 5], 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 build an incrementally assembled record of a user's workflow 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 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 dashboard or data curation pipeline. However, breadcrumbing in a REPL-based notebook also 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.

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. First,

[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

WOODSTOCK '97 El Paso, Texas USA
ACM ISBN 123-4567-24-567/08/06...$15.00
DOI: 10.475/123_4

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 the cell is idempotent. Second, treating the state as an opaque blob makes it difficult to automatically invalidate and recompute dependencies of a modified cell.

Vizier addresses both problems by isolating cells. In lieu of a REPL, cells execute in independent interpreter contexts. Cells can still communicate by consuming and producing *datasets*. This well-defined API enables efficient state snapshots, as well as dependency analysis across cell types.

Caveats. Vizier further leverages its structured data model to implement *caveats*, a technique for tracking possible errors based on incomplete databases [5, 11].

Concretely, we make the following contributions. 1) We outline the design of Vizier, a notebook-style workflow system; 2) We explore the challenges of integrating spreadsheet and notebook interfaces; and 3) We introduce caveats, a practical implementation of uncertainty annotations [5].

2. THE VIZIER NOTEBOOK

Vizier is an interactive code notebook similar to Jupyter or Apache Zeppelin. As in these systems, a Vizier notebook is a sequence of cells. Each cell describes one unit of computation, for example a Python or SQL script. Once a cell is executed, its results are displayed inline as part of the cell.

In contrast to REPL-based notebook systems (i.e., library managers), Vizier is a workflow and data manager: A notebook consists of a workflow specification (the cells and their configurations) as well as multiple datasets (relational tables). Both the workflow and the data are versioned as we will explain in more detail in Section 2.2. The cells of the notebook are steps in a workflow that each run in an isolated execution environment. Thus, no unforeseen interactions between cells are possible. The datasets of a notebook are either imported from the outside world using a load dataset cell or are produced as output by other cell types.

Cells can consume, produce, and update datasets through language-native APIs provided by Vizier for each cell type. For instance, in SQL cells, datasets are accessed as regular tables, while Python and Scala cells access datasets through a dataframe-style API. By isolating cells and by storing datasets in a relational database, Vizier can support many different cell types in the same notebook including Python, SQL, and Scala cells, as well as point-and-click cell types to streamline common notebook tasks like data ingest, missing value imputation, or visualization. In addition, using a relational database for state makes it possible to link other interaction modalities to the Vizier notebook. In particular, Vizier allows users to directly interact with notebook state through a spreadsheet-like view. User interactions in this view are translated back into (reproducible) notebook cells.

2.1 Data Caveats

Structured state also facilitates one of Vizier's core features: caveats. Similar to exception handling, caveats facilitate graceful, user-directed recovery from data errors.

Example 2. *Alice's dataset includes a CSV file with an integer column. On most rows, the column consists only of digits "100200" and can be safely parsed. However, on some rows, the column contains formatting (e.g., "£100,200"), and on others it contains data errors (e.g., "!o0,200").*

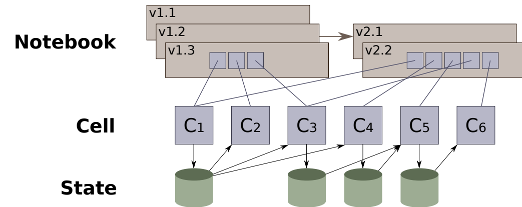

Figure 1: Vizier's layered data model consists of Notebook, Cell, and State versions.

Heuristics can often be used to recover from simple errors (e.g., dropping spurious characters or returning a NULL), allowing cell execution to proceed. Unfortunately applying heuristic repairs silently can create problems when the heuristic is wrong. The best alternative is laborious data unit testing, or to hard-stop the pipeline on every error. In Vizier, heuristic recovery code is instead instrumented to mark affected fields, rows, columns, or tables with *caveats* that propagate through notebook cells. Caveated values are then highlighted [8] and made available for review, allowing the user to decide whether the error merits her attention. We discuss caveats in greater depth in Section 4.

2.2 Version Model

Vizier maintains a branching version history for each notebook. A notebook version consists of cell configurations (specifying the workflow represented by the notebook) and a database (all dataset versions produced by executing the cells). Every edit a user makes to a notebook triggers Vizier to create a new notebook version, thus, preserving a full, reproducible history of the notebook. Users can browse previous versions and create a branch in the history of a notebook by editing a past version. Within a notebook version, there may be multiple versions of a dataset, as it is updated by different cells. For example, a load dataset cell imports a dataset into a notebook and the imported dataset may be updated by a cell to impute missing values. Internally, however, datasets, cells, and notebook versions are immutable to simplify versioning. To avoid unnecessary data replication, objects that are stable across notebook versions can be shared as illustrated in Figure 1.

Notebook Versions. A notebook version includes a reference to the prior version it was derived from and a compact representation of the changes between this and the previous version's cells. The notebook's cells are encoded as an ordered sequence of references to cell versions that may be shared across notebook versions. Notebook versions are identified by a randomly generated, unique branch identifier and a monotonically increasing version number.

Cell Versions. Vizier supports many cell types, including scripts (Python, SQL, Scala), Vizual [6] (DDL/DML operations), and point-and-click (e.g., Load Data, Plot, Repair Key, Impute). A cell version stores the cell's type along with user-provided input parameters (e.g., a python script).

The object storing a cell version is also used to cache results derived from executing the cell (unlike the version itself, the cache is mutable). Cached results are all non-stateful outputs from the cell: console output, or HTML-formatted data plots. Interactions between the cell and dataset versions are recorded in the form of read- and write-sets: the dataset versions that the cell read from and wrote

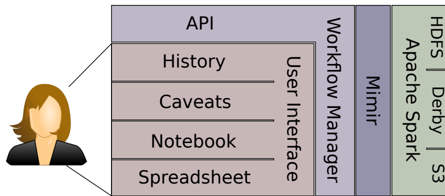

Figure 2: Layers of the Vizier System

to, respectively. Read and write sets are primarily used for inferring inter-cell dependencies, but may also be used as a back-up when fine-grained provenance is not available (Section 4.3). Cell versions are identified by an identifier derived from a hash of their parameters.

Dataset Versions. The datasets (relational tables) comprising a notebook's state are stored as Spark dataframes. As a cell executes, it may internally define tables *explicitly* by creating a materialized instance of the dataset. Alternatively, it may define a dataset *declaratively*, as a view over existing tables. Datasets versions are identified by a randomly generated, globally unique identifier. Users interact with datasets using human-friendly names.

2.3 The Vizier Stack

Figure 2 summarizes the four major layers of the Vizier stack: The User Interface, The Workflow Manager, The Mimir Incomplete Database, and Apache Spark.

The Workflow Manager. At the heart of Vizier is a simplified version of the VisTrails [3] workflow system, which is responsible for the top two layers of the versioned notebook model: notebook and cell versions. A REST API allows clients like Vizier's user interface to query state and manipulate notebooks. Clients can create new notebooks; add, update, or delete cells; or create new notebook branches.

The Workflow Manager also manages asynchronous cell execution, scheduling tasks for execution via a lightweight worker process (for easy setup) or a Celery distributed task queue (for better scaling). Cells are automatically executed when added or updated. When a cell is executed, its write set is compared against subsequent cell's read sets, and dependent cells are also scheduled for re-execution.

The Mimir Incomplete Database. A lightweight incomplete database [11, 5, 8] built over Apache Spark provides a REST API for managing state tables. Tables can be created *explicitly* by uploading data or by providing a URL. Tables may also be created declaratively: (1) by defining a view through Relational Algebra or SQL, (2) Through a spreadsheet-themed DDL/DML language called Vizual [6], or (3) through data-cleaning operations [11] for common tasks like imputation, schema matching, or key-repair. Notebook workloads involve many small changes, so declarative state tables are materialized lazily only when queried.

Mimir's main role is to implement caveats. When processing queries, Mimir applies a lightweight rewriting scheme [11, 5] that annotates query results to identify rows or fields that depend on a caveat-marked data value. We discuss this scheme and caveats in depth in Section 4.

The User Interface. Although it is possible to access the workflow manager's API directly, most user interactions are mediated through a React/Javascript frontend. As shown in Figure 3, Vizier's notebook view displays the current version of the notebook: a sequence of cells. The upper half of each cell shows a script or point-and-click configuration.

On branch **Default**

[1] LOAD DATASET readings FROM

readings (8 rows)

	TRAN_TS	JOIN_DT
0	Mon, 07 Oct 2013 08:23:19 GMT	Mon, 07 Oct 2013 00:00:00 G
1	Sun, 15 Jun 2014 08:23:19 GMT	Sun, 15 Jun 2014 00:00:00 G
2	Tue, 28 Oct 2014 08:23:19 GMT	Tue, 28 Oct 2014 00:00:00 GI
3	Mon, 17 Aug 2015 08:23:19 GMT	Mon, 17 Aug 2015 00:00:00 G
4	Tue, 27 Oct 2015 08:23:19 GMT	Tue, 27 Oct 2015 00:00:00 GI
5	Fri, 06 Nov 2015 08:23:19 GMT	Fri, 06 Nov 2015 00:00:00 GN
6	Mon, 21 Mar 2016 08:23:19 GMT	Mon, 21 Mar 2016 00:00:00 G
7	Mon, 10 Apr 2017 08:23:19 GMT	Wed, 21 Jun 2017 00:00:00 G

[2]

```
1 # Get object for dataset with given name.
2 ds = vizierdb.get_dataset('readings')
3
4 v = ds.rows[0].get_value("JOIN_DT")
5
6 print("XXX: {}<br/>YYY: {}".format(v, v.year
```

XXX: 2013-10-07
YYY: 2013

Figure 3: The Vizier Notebook View

The bottom half of each cell is a tabbed result area. By default this shows console output, errors, and/or HTML generated by a data visualization library like Bokeh or by certain point-and-click cells. Additional tabs allow users to view data tables as they evolve through the notebook. These tabs show the version of each dataset as it was immediately after the cell was last executed. Figure 3 illustrates this: The result area of the Load Dataset Cell displays the recently loaded table. Results are shown in a simple grid, with fields and rows affected by caveats highlighted in red.

In addition to its notebook, Vizier provides a spreadsheet-like interface for each state table, allowing direct manipulation of data and schemas. Finally, the user interface also provides detail views for caveats and the notebook history.

3. THE SPREADSHEET INTERFACE

The spreadsheet displays a state table, similar to the result area table display in Figure 3. However, this grid is interactive: The user may insert, reorder, rename, or delete columns and rows; modify the contents of cells; or apply simple data transformations like sorting.

3.1 Displaying Data

A classical spreadsheet offers a large grid of unstructured cells. By contrast, Vizier's spreadsheet view show a single structured, relational table defined by the notebook. The spreadsheet view is populated by querying Mimir and retrieving rows incrementally as they are needed for display.

Caveats. As described above, Mimir uses lightweight instrumentation to identify query result cells and rows affected by caveats. Elements affected by caveats are highlighted in the spreadsheet through red text. When a highlighted cell is selected, an icon appears allowing users to query Mimir to retrieve descriptive text for all caveats affecting the cell.

3.2 Spreadsheet Updates

In [6], we developed a simple language called *Vizual* that models user actions on a spreadsheet. Each operation in Vizual is implemented as a cell type in Vizier. Thus, each modification of the spreadsheet adds the corresponding cell

to the notebook. There may be many such interactions, so the notebook view defaults to a "collapsed" representation that groups adjacent Vizual cells into a single script cell.

We needed to address two mismatches between spreadsheets and Vizier's relational state model. First, columns in a database table are strongly typed, while spreadsheets are typed on a per-cell basis. To match the typical spreadsheet experience, column types are escalated to match newly entered values — **string** if nothing else matches. Second, empty cells in a spreadsheet can indicate either an empty cell or the empty string. Vizier leaves empty strings unaltered in **string** columns, and treats them as `NULL` otherwise.

3.3 Reenactment

Through the spreadsheet interface, users can create, rename, reorder, or delete rows and columns, or alter data — the standard array of DDL and DML operations. These operations can not be applied in-place without sacrificing versioning. In lieu of copying data, Vizier builds on a technique called reenactment [9, 2], which translates sequences of DDL operations into equivalent queries.

EXAMPLE 3. *Consider a table* `EMP` *and the statements:*

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

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;
```

Vizier translates Vizual operations into an equivalent query, resulting in each table version being defined (declaratively) as a view. Due to limited space, we refer the interested reader to [9] for an introduction to reenactment, with a further discussion of its applicability to spreadsheets in [6].

Vizual [6] also includes DDL operations like column renaming, reordering, and creation. We implemented these through a similar view-based approach, adopting transformations from the Prism Workbench [4]. For example, column creation is analogous to projecting the full schema of the table plus an additional column initialized with `NULL`.

3.4 Update Targets

Identifying update targets for Vizual operations presented a challenge. In SQL DDL, update operations specify target rows by a predicate, making the user's intent explicit. By contrast, spreadsheet users specify target rows by pointing at them. When the source data changes — for example when a new cell is added earlier in the notebook — the user's update must be re-applied to the new data. Thus, we need row identifiers that are stable through such changes.

Vizier builds on the row identity model of GProM [1]. 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.[3] (3) Rows in the output of a cross product or join are identified by combining identifiers from the source rows that produced them, and (4) Rows in the output of an aggregate are identified by the group-by attribute.

We considered three approaches to identifying rows in explicitly declared state tables: 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 approaches: deriving identifiers from both sequence and hash code. Such row-identifiers are stable under appends, but not re-used.

3.5 Interactive Update Performance

After every user interaction, the spreadsheet view must be refreshed. With reenactment, this requires adding a new cell, executing it (running a query), and retrieving the results. For updates to cell values in particular, we have found significant pushback from users to lag while the system refreshes the view. To make value updates "feel" instantaneous, refreshes are handled asynchronously; While refreshing, placeholders selected client-side stand in for updated values. Leveraging the caveat metaphor, such values are highlighted in red until the refresh completes.

4. MANAGING CAVEATS

Analytics tools often encounter data errors like un-parseable integers or CSV files with uneven column counts. Existing systems take one of three approaches to handling such errors: (1) The error stops the workflow, discarding work done or leaving the system in an ambiguous state; (2) The error is silently dropped as the system heuristically recovers, impeding debugging efforts; or (3) The system heuristically recovers and logs the event to a log file that no one looks at.

As described in Section 2.1, Vizier takes a fourth approach: annotating affected fields, rows, columns, and tables with a caveat. Caveats include a human-readable description of the problem, and are propagated through queries to any dependent elements derived from them. Propagation follows a simple rule: *Is it possible to change the derived value by changing the caveatted value?* If so, the caveat propagates to the derived value. Caveats were originally introduced as *uncertain values* in [11]. We later formalized propagation of row caveats in [5] and proposed a minimal-overhead rewrite-based implementation. Here, we focus on the practical challenges of realizing caveats in Vizier.

4.1 Applying Data Caveats

Caveats in Vizier are implemented by Mimir, a simple query rewriting frontend over Spark's dataframes. Mimir provides a new function: `caveat(id, value, message)` to annotate data. The function takes 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,

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

and omitted from examples for conciseness. Rows are annotated when the caveat appears in a `WHERE` clause[4]. We now highlight several examples of how Vizier instruments several operators to capture heuristically recoverable errors.

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 common, as in the following:

```
SELECT CAST(pay as int) AS pay, ... FROM EMP;
```

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

```
CASE WHEN CAST(pay as int) IS NULL
     THEN caveat(NULL, pay & ' is not an int')
     ELSE CAST(pay 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 EMP_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 Implementing Data Caveats

The power of caveats arises from being propagated through queries, allowing users to avoid costly data repairs to irrelevant data. In this section, we overview Vizier's lightweight strategy for propagating caveats through a Vizier notebook.

Example 4. *Pay for a small number of employees in Alice's dataset includes foreign currency markers (e.g., £). Alice's analytics tool silently converts these numbers to* `NULL`*. Alice begins exploring with a query:*

```
SELECT dept, avg(pay) FROM EMP GROUP BY dept;
```

The result that Alice gets is incorrect. Vizier however, caveats the average pay of departments with remote employees, helping Alice to discover and repair the error.

To avoid the overheads of propagating annotated data directly, Vizier simulates propagation in three stages: Presence, Static, and Detail. Each stage provides a progressively more detailed picture of the caveats affecting a query result.

Presence. The first stage identifies affected fields or rows. Queries are rewritten recursively through a scheme detailed in [5] to add boolean-valued attributes that indicate that a column, or the entire row depends on a caveatted value.

[4]We leave a discussion of how Vizier implements column and table caveats to future work

Example 5. *Consider Alice's query over* `EMP`*, instrumented as above to cast* `pay` *to an integer and assuming no other caveat-instrumented operations.*

```
WITH EMP AS ( /* cast pay as above */ )
SELECT dept, avg(pay) FROM EMP GROUP BY dept;
```

This query would be instrumented and optimized into:

```
WITH EMP AS (
  SELECT CAST(pay as int) AS pay, dept, ...,
         ( CAST(pay as int) IS NULL
           AND _caveat_pay) AS _caveat_pay
  FROM EMP_uncast)
SELECT dept, avg(pay),
       exists(_caveat_pay) AS _caveat_avg
  FROM EMP GROUP BY dept;
```

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

Propagating caveats is analogous to a probabilistic database query (i.e., $\#P$ [5]). 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 [5] for the algorithm to unnecessarily mark cells or rows. As we show in our experimental results, caveat propagation also has minimal computational overhead relative to native query processing.

Static. The initial phase 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 state table 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.

For both caveat detail views, the first step 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 6. *Alice asks for caveats affecting the Billing department's average pay. The* `caveat` *function is used exactly once, so we obtain a single CaveatSet:*

```
SELECT pay&' is not an int' AS message,
       ROWID AS id
FROM EMP_uncast WHERE dept = 'billing'
```

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`s appears, we construct a query to compute parameters for every call to `caveat`.

Detail. 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 many 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.

4.3 Coarse-Grained Cells

Propagating caveats through queries is sufficient for the declaratively specified tables created by SQL cells, most point-and-click cells, and the Spreadsheet-generated Vizual cells. However, certain cell types like Python cells rely on explicit tables 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 tables 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 4: Datasets Evaluated

We base our experiments on the experiments of [5], using the two real world datasets summarized in Figure 4. 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 [5]. 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 5 and 6 [5] show a timeline of the query and caveat generation process. 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.

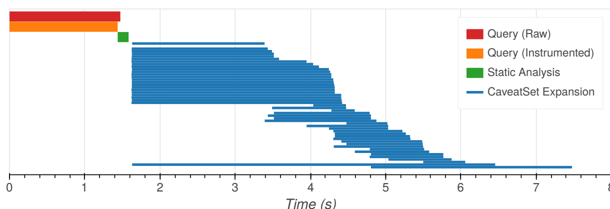

Figure 5: Materializing caveats for `Shootings`

6. CONCLUSIONS AND FUTURE WORK

In this paper, we discuss the design and implementation of Vizier, a novel system for data curation and exploration with 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 versioned workflow manager notebook system. 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 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 Vizier can natively incorporate sampling.

[5] Plots generated with Vizier using Bokeh.

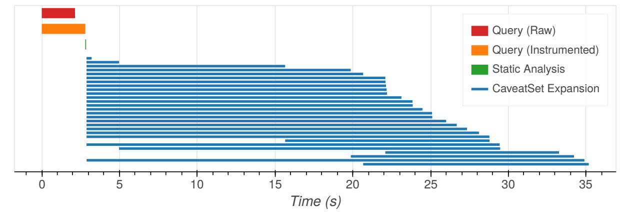

Figure 6: Materializing caveats for `Graffiti`

7. REFERENCES

[1] 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.

[2] B. S. Arab, D. Gawlick, V. Krishnaswamy, V. Radhakrishnan, and B. Glavic. Using reenactment to retroactively capture provenance for transactions. *IEEE Trans. Knowl. Data Eng.*, 30(3):599–612, 2018.

[3] 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.

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

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

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

[7] D. Koop and J. Patel. Dataflow notebooks: Encoding and tracking dependencies of cells. In *Proceedings of the 9th USENIX Conference on Theory and Practice of Provenance*, TaPP'17, pages 17–17, Berkeley, CA, USA, 2017. USENIX Association.

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

[9] 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.

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

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