Overlay Spreadsheets

Oliver Kennedy
okennedy@buffalo.edu
University at Buffalo
Buffalo, USA

Boris Glavic
bglavic@iit.edu
Illinois Institute of Technology
Illinois, USA

Michael Brachmann
mbrachmann@breadcrumb-analytics.com
Breadcrumb Analytics
Buffalo, USA

ABSTRACT

Efforts to scale spreadsheets either follow a 'virtual' strategy that imposes a spreadsheet interface over an existing database engine or a 'materialized' strategy based on re-engineering the spreadsheet engine. Because database engines are not optimized for spreadsheet access patterns, the materialized approach has better performance. However, the virtual approach offers several advantages that can not be easily replicated in the materialized approach, including the ability to re-apply user interactions to an updated dataset. We propose a hybrid approach, where patterns of user updates are indexed (as in the materialized approach) and overlaid on an existing dataset (as in the virtual approach). We introduce the overlay update model, and outline strategies for efficiently accessing an overlay spreadsheet. A key feature of our approach is storing updates generated by bulk operations (e.g., copy/paste) as "patterns" that can be leveraged to reduce execution costs. We implement an overlay spreadsheet over Apache Spark and demonstrate that, compared to DataSpread (a standard materialized-style spreadsheet), it can significantly reduce execution costs.

ACM Reference Format:
Oliver Kennedy, Boris Glavic, and Michael Brachmann. 2023. Overlay Spreadsheets. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Spreadsheets are a popular tools for data exploration, transformation, and visualization, but have historically had challenges managing "big data" — as few as 50k rows of data create problems for existing spreadsheet engines [16]. One approach to scalability, employed by *Wrangler* [12], *Vizier* [8, 10], and others [15] relies on translating spreadsheet interactions into declarative transformations (dataflows) that can be deployed to a database or dataflow system. In this model, the spreadsheet is a chain of versions, each linked by a lightweight transformation function [10]. A different approach employed by *DataSpread* [5, 6, 16], instead re-architects the spreadsheet runtime and specializes database primitives like indexes and incremental maintenance for spreadsheet access patterns. We refer to these as the virtual and materialized approaches, respectively, and illustrate them in Figure 1.

Conference'17, July 2017, Washington, DC, USA
2023. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

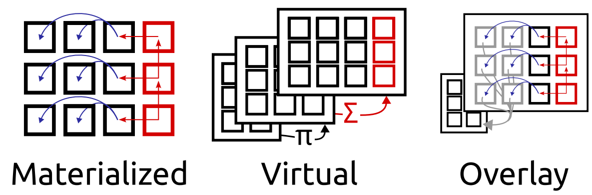


Figure 1: Approaches to scalable spreadsheet design

The materialized approach is optimized for multiple data access patterns common to spreadsheets [5, 6, 16], including (i) Data structures specialized for the positional referencing scheme commonly used in spreadsheet formulas [5], (ii) Execution strategies that prioritize completion of portions of the spreadsheet that the user is viewing [6], and (iii) Indexes storing compressed dependency graphs [6, 17]. Similar optimizations are considerably harder in the virtual approach, as the result of updates and their effects on cell position are only materialized when data is received.

Although the virtual approach is often less efficient, it does provide capabilities that the materialized approach does not: (i) It is a naturally efficient encoding of the spreadsheet's full version history. (ii) As in Wrangler, the user's actions can be re-applied to new data (e.g., an updated version of the source data); and (iii) As in Vizier, the spreadsheet can be re-encoded as a relational query allowing it to "plug into" existing scalable computation platforms (e.g., Spark [1]) and provenance analysis tools (e.g., [14]).

We propose an optimized hybrid of the virtual and materialized approaches: *Overlay Spreadsheets.* An Overlay Spreadsheet (Figure 1) presents an interface analogous to a normal spreadsheet. User edits are "overlaid" on top of a source dataset that can be easily be updated to a new version. As an added benefit, decoupling edits and source data makes it easier to leverage spreadsheet access patterns, reducing the time needed to respond to user actions.

We outline a preliminary implementation of Overlay Spreadsheets within Vizier [7, 8, 13], a multi-modal notebook-style workflow system built on Apache Spark. Existing versions of Vizier allow users to define workflow steps through a spreadsheet-style interface; each action adds a new workflow step. In spite of the performance limitations of this virtual approach, it remains preferable for Vizier, where (i) changes to an early step in the workflow may require automatically re-applying the user's edits, and (ii) fine-grained provenance features rely on encoding data transformations as Spark dataframes. Our objective is to demonstrate that a spreadsheet-style interface can provide **interactive latencies** (i.e., like the materialized approach), while still supporting **replay and provenance** (i.e., like the virtual approach).

As a secondary goal, we explore potential performance improvements that the overlay approach enables. Specifically, we observe that bulk updates in a spreadsheet (e.g., pasting a formula across a range of cells) rely on expression "patterns," which admit more efficient dependency analysis and bulk computation, when intermediate values are not required. This hybrid strategy is akin to optimizations applied in DataSpread [6, 17], but operate over patterns of updates rather than patterns in the dependency graph, enabling additional optimizations.

2 SPREADSHEET DATA MODEL

2.1 Spreadsheets

Let $\mathcal{C}$ and $\mathcal{R}$ denote domains of column and row labels. Except where noted, $\mathcal{R} \subset \mathbb{Z}$. Let $\mathcal{V}$ and $\mathcal{E} \supset \mathcal{V}$ denote domains of values and expressions, respectively. A *spreadsheet* $\mathbb{S} : (\mathcal{C} \times \mathcal{R}) \rightarrow \mathcal{E}$ is a partial mapping from *cells* ($c[r] \in (\mathcal{C} \times \mathcal{R})$) to expressions. We use $\mathbb{S}[c, r]$ to denote $\mathbb{S}(c[r])$. Let $\perp \in \mathcal{V}$ indicate "undefined" and define the *domain* $\textsc{Dom}(\mathbb{S})$ to be the set of cells $c[r]$ where $\mathbb{S}[c, r] \neq \perp$.

An expression $e \in \mathcal{E}$ is a formula defined over literals from $\mathcal{V}$, the standard arithmetic operators, and references to other cells in the spreadsheet ($c[r]$). The expression e is evaluated in the context of a spreadsheet ($[\![\cdot]\!]_{\mathbb{S}} : \mathcal{E} \rightarrow \mathcal{V}$) as follows: (i) Literals and arithmetic are evaluated in the usual way, and (ii) References to the spreadsheet are evaluated recursively ($[\![c[r]]\!]_{\mathbb{S}} \equiv [\![\mathbb{S}(c, r)]\!]_{\mathbb{S}}$). By convention, cyclic references evaluate to $\perp$. An expression's dependencies (**deps** (e)) are the cells referenced by e. Dependencies induce a graph $G_{\mathbb{S}} \langle N, E \rangle$ over the spreadsheet, with cells as nodes (i.e., $N = \mathcal{C} \times \mathcal{R}$), and dependencies as directed edges:

$$E = \bigcup_{c[r] \in \mathcal{C} \times \mathcal{R}} \{ \, c[r] \rightarrow c'[r'] \mid c'[r'] \in \textbf{deps}\,(\mathbb{S}[c, r]) \, \}$$

Denote by $G^*_{\mathbb{S}}$ the graph $\langle V, E^* \rangle$ where E^* is the transitive closure of E (i.e., $G^*_{\mathbb{S}}$ captures both direct and indirect dependencies). Note that if all cell expressions are constants (i.e., a spreadsheet without formulas), then $[\![c[r]]\!]_{\mathbb{S}} = \mathbb{S}[c, r]$.

Example 2.1. Consider the spreadsheet at the top of Figure 2. Columns A and B hold constant expressions, while column C holds reference cells from columns A and B. Evaluating this spreadsheet assigns each cell a value, as in the top right. For example, *C[1]* evaluates to $[\![A[1] + B[1]]\!]_{\mathbb{S}} = [\![A[1]]\!]_{\mathbb{S}} + [\![B[1]]\!]_{\mathbb{S}} = 15 + 50 = 65$.

Spreadsheet $\mathbb{S}$

	A	B	C
1	15	50	A1 + B1
2	20	60	A2 + B2
3	25	100	A3 + B3
4	50	0	A4 + B4

Evaluated Spreadsheet $[\![\cdot]\!]_{\mathbb{S}}$

	A	B	C
1	15	50	65
2	20	60	80
3	25	100	125
4	50	0	50

Update $U = \{A[1] = 20, C[3] = 2 \cdot A3 + B3\}$

Updated Spreadsheet $U(\mathbb{S})$

	A	B	C
1	20	50	A1 + B1
2	20	60	A2 + B2
3	25	100	2 · A3 + B3
4	50	0	A4 + B4

Evaluated Update $[\![\cdot]\!]_{U(\mathbb{S})}$

	A	B	C
1	20	50	70
2	20	60	80
3	25	100	150
4	50	0	50

Figure 2: Example spreadsheet with expressions shown in dark green, and an update applied to the spreadsheet with updated expressions and values shown in red.

2.2 Cell Updates

A cell update set $U \subseteq \mathcal{C} \times \mathcal{R} \times \mathcal{E}$ is a set of cell updates of the form $c[r] = e$ that assign to cell $c[r]$ the expression e. Denote by $\textsc{Dom}(U)$ the domain of update U, containing all cells $c[r]$ defined in U (i.e., $\exists e : ([c[r] = e] \in U)$). Applying an update U to a spreadsheet $\mathbb{S}$ returns an updated spreadsheet:

$$U(\mathbb{S})[c, r] = \begin{cases} U(c[r]) & \textbf{if } c[r] \in \textsc{Dom}(U) \\ \mathbb{S}[c, r] & \textbf{otherwise} \end{cases}$$

An update may affect cells beyond its domain. For example, the update shown in Figure 2 changes two cells *A[1]* and *C[3]*, but evaluating the updated spreadsheet $U(\mathbb{S})$ results in *three* cell changes (in red).

2.3 Spreadsheet Access to Datasets

To uniformly model source datasets, whether from relational databases or other spreadsheets, we assume an input dataset D with designated row and column labels $\mathcal{C}_D$ and $\mathcal{R}_D$ as appropriate to the source data. In a relational table, these are the table's columns and the values of a key or rowid attribute, respectively. For csv data, $\mathcal{R}_D \subset \mathbb{Z}$ is the position of the row in the file. We write $D[r, c]$ to denote the value at column $c \in \mathcal{C}_D$ of row $r \in \mathcal{R}_D$ in D.

Denote by $\mathcal{F} : \mathcal{R}_D \rightarrow \mathbb{Z}$ a reference frame, an injective map from rows in D to rows of the spreadsheet. A *spreadsheet overlay* for a dataset D is then a pair $(D, \mathcal{F})$ that defines a spreadsheet $\mathbb{S}_{D,\mathcal{F}}$ with domains $\mathcal{C} = \mathcal{C}_D$, $\mathcal{R} = \textsc{Dom}(\mathcal{F})$ as $\mathbb{S}_{D,\mathcal{F}}[c, r] = D[c, \mathcal{F}^{-1}(r)]$

2.4 Overlay Updates

An Overlay Update describes a set of changes to a spreadsheet (or dataset). As we discuss in Section 3.1, column operations are purely cosmetic in our model, and we focus on cell and row updates exclusively. Concretely, a spreadsheet overlay $O = \langle \mathcal{T}, \mathcal{U} \rangle$ is a reference frame transformation $\mathcal{T}$ and a set of pattern updates $\mathcal{U}$, terms we now define.

Reference Frame Transformations. Recall that a reference frame maps the spreadsheet's positional row references to native record identifiers. Thus, to insert, delete, or move rows in the spreadsheet, it is sufficient to modify the reference frame. Formally, a reference frame transformation $\mathcal{T}$ is an injective mapping $\mathbb{Z} \to \mathbb{Z} \cup \perp$ from initial row positions to new row positions, or the value $\perp$ for a deleted row. The new reference frame, after applying O is $\mathcal{F}' = \mathcal{T} \circ \mathcal{F}$, where $\circ$ denotes function composition. As an example, consider deleting the 2nd row of the spreadsheet from Figure 2. The positions of rows 3 and 4 are decreased by one, while row 1 retains its position

$$\mathcal{T}(x) = \begin{cases} x & \textbf{if } x < 2 \\ \perp & \textbf{if } x = 2 \\ x - 1 & \textbf{otherwise} \end{cases}$$

Row insertions and movement are handled analogously. Note that row insertions, deletions, and movement are expressible in constant size, independent of the size of the data.

Pattern Updates. Spreadsheets allow a formula from one cell to be pasted across a range of cells. In a classical spreadsheet, bulk interactions like this modify each cell's expression individually. Overlay spreadsheets avoid the high cost that individual modifications can entail by grouping together the set of pasted cells into a single *pattern*.

A *range* $C[R]$ is the Cartesian product $C \times [l, h]$ of a set of columns ($C \subseteq \mathcal{C}$) and row positions ($R = [l, h] \subset \mathbb{Z}$). A pattern update $\mathcal{U}$ is a set of pairs $\{(C_i[R_i], P_i)\}$ where $C_i[R_i]$ is a range and P_i is a *pattern expression*, i.e., an expression that may also contain cell references where rows are relative offsets (written as $+i$ or $-i$). Ranges in an update $C_i[R_i]$ must be pairwise disjoint. A pattern update $(C_i[R_i], P_i)$ assigns an expression to every cell $c[r]$ in $C_i[R_i]$ by replacing any relative references of the form $c[+\delta]$ in P_i with $c[r + \delta]$. We use $P_i(c[r])$ to denote instantiation of pattern P_i for cell $c[r]$.

For instance, to store a running sum of the values in column C into column D (for the spreadsheet from Figure 2):

$$\mathcal{U}_{running} = (D[1], (C, +0)), (D[2-4], (C, +0) + (D, -1))$$

Semantics for Overlay Updates. An overlay update O applied to a spreadsheet $\mathbb{S}$ defines the spreadsheet $O(\mathbb{S})$ computed by applying the reference frame update and then applying all pattern updates (with $O = \langle \mathcal{T}, \{(C_i, R_i, P_i)\} \rangle$):

Spreadsheet $\mathbb{S}$

	A	B	C	D
1	15	50	A1 + B1	C1
2	20	60	A2 + B2	C2 + D1
3	25	100	A3 + B3	C3 + D2
4	50	0	A4 + B4	C4 + D3

Evaluated Spreadsheet $[\![\cdot]\!]_{\mathbb{S}}$

	A	B	C	D
1	15	50	65	65
2	20	60	80	145
3	25	100	125	270
4	50	0	50	320

Figure 3: Example overlay update and result (updated expressions and values are shown in red).

$$O(\mathbb{S})[c, r] = \begin{cases} P_i(c[r]) & \textbf{if} \exists i : c[r] \in C_i[R_i] \\ \mathbb{S}[c, \mathcal{T}^{-1}(r)] & \textbf{if} \exists r' : \mathcal{T}(r') = r \\ \perp & \textbf{otherwise} \end{cases}$$

Example 2.2. Consider our example update ($O_{running} = (\mathcal{T}_{id}, \mathcal{U}_{running})$ where $\mathcal{T}_{id}(x) = x$). Figure 3 shows the result of applying $O_{running}$ to our running example spreadsheet.

Several remarks are in order. First, overlays can be used to encode common spreadsheet update operations in constant space (per update), including bulk updates via copy/paste. Second, [17] uses similar ideas to compress the dependencies in a spreadsheet using ranges and patterns, but focuses exclusively on the dependency graph rather than expressions.

2.5 Replacing Source Data

An overlay designed for source data $(D, \mathcal{F})$ may be applied to a dataset $(D', \mathcal{F}')$ as long as each $r \in \mathcal{R}_D$ that corresponds to some $r' \in \mathcal{R}_{D'}$, $\mathcal{F}'(\mathcal{F}^{-1}(r)) = r'$. This is possible if, for example, $\mathcal{R}_D = \mathcal{R}_{D'}$ is a semantic key for the dataset.

3 SYSTEM DESIGN

Our prototype overlay spreadsheet is implemented within the Vizier reproducible notebook platform [7, 8, 13]. Vizier leverages Apache Spark [1] for data provenance, processing, and data import/export. Our prototype is designed to accept any Spark dataframe as a data source.

Client applications connect through a thin **Presentation** layer that mediates concurrent access to the spreadsheet and translates to our simplified model of a spreadsheet to a more natural interface. An **Execution** layer is responsible for evaluating spreadsheet cells and materializing values for the viewable set of cells. An **Indexing** layer provides efficient access to the updates themselves, and a simple LRU cache provides efficient random access to the source dataframe.

3.1 Presentation Layer

User-facing client applications connect to the overlay spreadsheet through a presentation layer. This layer mediates concurrent updates of the spreadsheet and provides clients with the illusion of a fixed grid of cells by defining and maintaining an explicit order over columns. Column operations (insertion, deletion, reordering) are handled at this layer, so lower levels can reference the (comparatively small) set of columns solely by column identity. Other updates are put into a serial order and relayed to lower levels.

The presentation layer expects the Executor to provide efficient random access to cell values and support updating ranges of cells with pattern expressions.

3.2 Executor

The executor provides efficient access to cell values and generates notifications about cell state changes. Cell values are derived from two sources: (i) A data source $(D, \mathcal{F})$ defines a base spreadsheet $\mathbb{S}_D[c, r] = D[c, \mathcal{F}^{-1}(r)]$, and (ii) A sequence of overlay updates ($O_1 \ldots O_k$; where $O_i = \langle \mathcal{T}_i, \mathcal{U}_i \rangle$) that extend the spreadsheet $\mathbb{S} = (O_k \circ \ldots \circ O_1)(\mathbb{S}_D)$. These sources are implemented by a cache around $\mathbb{S}_D$ and the update index, as discussed below.

The naive approach to materializing $\mathbb{S}$ (e.g., as in [6]) computes a topological sort over cell dependencies and evaluates cells in this order. The Executor side-steps the linear (in the data size) cost of the naive approach through two insights: (i) Updates applied over multiple cells are already provided by higher layers as patterns, and (ii) Only a small fraction of cells will be visible at any one time. Assuming the dependencies of a range of cells can be computed efficiently (we return to this assumption in Section 3.3), only the visible cells and any hidden dependencies need to be evaluated. The Executor only evaluates cell expressions on rows that are (close to being) visible to the user, and the transitive closure of their dependencies.

Some dependency chains (e.g., running sums) still require computation for each row of data. Although we leave a detailed exploration of this challenge to future work, we observe that the fixed point of such pattern expressions can often be rewritten into a closed form. For example, any cell in a running sum column is equivalent to a sum over the preceding cells. Our preliminary experiments (Section 4) suggest promise in a hybrid evaluation strategy that evaluates visible cells individually and computes cells defined by patterns through closed form aggregate queries.

Updates. When the executor receives an update to a cell, it uses the index to compute the set of invalidated cells, marks them as "pending," and begins re-evaluating them in topological order. An update to the reference frame is applied to both the index and the data source. Following typical spreadsheet semantics, an insertion or row move updates references in dependent formulas, so no re-evaluation is typically required. If a row with dependent cells is deleted, the dependent cells need to be updated to indicate the error.

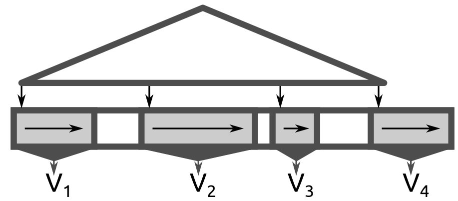


Figure 4: A range map maps disjoint ranges to values.

3.3 Update Index

The update index stores sequence of updates ($O = O_k \circ \ldots \circ O_1$) and provide efficient access to the cells of an overlay spreadsheet (denoted $\mathbb{S}_O$) where undefined cells have the value $\perp$. This entails: (i) cell expressions $\mathbb{S}_O[c, r]$ (for cell evaluation); (ii) upstream dependencies of a range (for topological sort and computing the active set), and (iii) downstream dependents of a range (for cell invalidation after an update). The key insight behind the index is that updates are stored as pattern-range tuples instead of as individual cells.

Range Maps. The update index is built over a one-dimensional range map, an ordered map with integer keys. In addition to the usual operations of an ordered map (e.g., `put`, `get`, `successorOf`), we define the operation `bulkPut(low, high, value)` which is equivalent to a `put` on every element in the range from `low` to `high`. Implemented naively (e.g. a size N binary tree), this operation is $O((\texttt{high} - \texttt{low}) \cdot \log(N))$.

A range map avoids the $(\texttt{high} - \texttt{low})$ factor (and correspondingly reduces N) by storing an ordered sequence of disjoint ranges, each mapping one specific value as illustrated in Figure 4. A binary tree provides efficient membership lookups over the ranges. With a range map, the set of distinct values appearing in a range can be accessed in $O(\log(N) + M)$ time (where M is the number of distinct values), and has similar deletion and insertion costs.

Cell Access. The index layer maintains a "forward" index: An unordered map $\mathcal{I}$ that stores a range map $\mathcal{I}[c]$ for each column. The expression for a cell $c[r]$ is stored at $\mathcal{I}[c][r]$.

Upstream Reachability. The execution layer needs to be able to derive the set of cells on which a specific target cell (or range) depends. We refer to this set as the target's *upstream*. Algorithm 1 illustrates a naive breadth-first search to obtain the full upstream set for a given target range. Each item in the BFS's work queue consists of a column, a row set, and a lineage; We will return to the lineage shortly. For each work item enqueued, we query the forward index to obtain the set of patterns in the range (line 4), and iterate over the set of

Algorithm 1 upstream(C, R)

Require: $C, R[]$: A range to compute the upstream of.
Ensure: `upstream`: Cells on which $c[R]$ is a dependency.

1: `upstream` $\leftarrow \{\}$
2: `work` $\leftarrow \{\ (c, R, \{\}) \mid c \in C\ \}$
3: **while** $(c', R', \texttt{lineage}) \leftarrow$ `work`.**dequeue** **do**
4:     **for** $(R'', \texttt{pattern}) \leftarrow \texttt{forwardIndex}(c', R')$ **do**
5:         **for** $(c_d, R_d, \texttt{offset}) \leftarrow$ **getDeps**$(\texttt{pattern}, c', R'')$ **do**
6:             $(c_d, R_d) \leftarrow (c_d, R_d) - \texttt{upstream}$
7:             **if** (c_d, R_d) is non-empty **then**
8:                 $\texttt{upstream} \leftarrow \texttt{upstream} + (c_d, R_d)$
9:                 `queue`.**enqueue**$(c_d, R_d,$
10:                     $\{\ \texttt{p}' \rightarrow (\texttt{o}' + \texttt{offset})$
11:                       $\mid (\texttt{p}' \rightarrow \texttt{o}') \in \texttt{lineage}\}$
12:                         $\cup\{\texttt{pattern} \rightarrow \texttt{offset}\})$

their dependencies (line 5). If we discover a new dependency (lines 6-7), the newly discovered range is added to the return set and the work queue. We will explain lines 10-12 shortly.

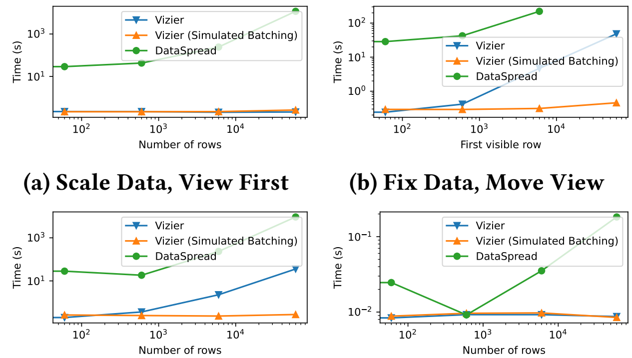
The **getDeps** operation (Line 5; Algorithm 2) computes the immediate dependencies of a range of cells $c[R]$ that share a pattern. Concretely, it returns a set of cells deps such that for each cell $c[r] \in$ deps, there exists at least one cell $c[r]' \in c[R]$ such that $c[r]$ is in the transitive closure of **deps** $(c[r]')$. The algorithm uses a recursive traversal (lines 6-7) to visit every cell reference (offset or explicit): For offset references (lines 2-3), the provided range of rows is offset by the appropriate amount. For explicit cell references (lines 4-5), the explicit reference is used.

Algorithm 2 getDeps$(\texttt{pattern}, c, R)$

Require: `pattern`: An expression pattern
Require: $c[R]$: A range of cells
Ensure: `deps`: The dependencies of `pattern` applied to $c[R]$

1: `deps` $\leftarrow \{\}$
2: **if** `pattern` **is an offset reference** $c', \delta'[]$ **then**
3:     $\texttt{deps} \leftarrow \texttt{deps} \cup \{(c', R + \delta', \delta')\}$
4: **else if** `pattern` **is a direct reference** $c', r'[]$ **then**
5:     $\texttt{deps} \leftarrow \texttt{deps} \cup \{(c', r', \emptyset)\}$
6: **else**
7:     $\texttt{deps} \leftarrow \texttt{deps} \bigcup_{\texttt{child} \in \texttt{pattern}}$ **getDeps**$(\texttt{child}, c, R)$

Optimizing Recursive Reachability. Consider a running sum, such as the one in Example 2.2. The kth element will have $O(k)$ upstream dependencies, and so naively following Algorithm 1 requires $O(k)$ compute. However, observe that a single pattern is responsible for all of these dependencies, suggesting that a more efficient option may be available.



(a) Scale Data, View First (b) Fix Data, Move View

(c) Scale Data, View Last (d) Scale Data, View First

Figure 5: Time to initialize the spreadsheet (a-b) and cost to update one cell (c-d)

This dependency chain arises from recursion over single pattern; most cells depend on other cells defined by the same pattern. We refer to such a pattern as *recursive*, even if it does not create dependency cycle over individual cells.

As with cell execution, the transitive closure of the dependencies of a recursive pattern has a closed-form representation. In our running example, the upstream of any $D[k]$ is exactly $D[1-(k-1)]$ and $C[1-k]$. The `lineage` field of Algorithm 1 is used to track the set of patterns visited, and the offset(s) at which they were visited. If the pattern being visited already appears in the lineage, then we know it is recursive and that we can extend out the sequence of upstream cells across the remaining cells of the pattern. When the offset is ± 1, the elements of this sequence are efficiently representable as a range of cells, computable in $O(1)$ time.

Downstream Reachability. When a cell's expression is updated, cells that depend on it (even transitively) must be recomputed, so the index must support downstream reachability queries. For efficient downstream lookups, the index maintains a "backward" index relating ranges to the set of patterns that depend on all cells in the range. The resulting algorithm over the backward index is analogous to **getDeps**.

4 EXPERIMENTS

In this section we explore the performance of the overlay approach. Concretely, we are interested in two questions: (i) How does data size affect the performance of each system? (ii) How does dependency chain length affect the performance of each system? Experiments were run on an 8-core 2.3 GHz Intel i7-11800H running Linux (Kernel 5.19), with 32G of DDR4-3200 RAM, and a 2TB 970 EVO NVME solid state drive. We compare three systems: (i) **DataSpread**: Dataspread version 0.5 [4]; (ii) **Vizier**: Our prototype implementation of overlay spreadsheets; and (iii) **Vizier (Simulated Batching)**: Simulated hybrid batch processing (see Setup, below). All experiments were performed with a warm cache.

Setup. We address our questions through a microbenchmark modeled after TPC-H query 1 [9]: The spreadsheet is defined by the TPC-H `lineitem` dataset with `N` rows and four additional columns defined by the patterns:

```
base_price[1-N] = ext_price[+0]
disc_price[1-N] = base_price[+0] * (1 - discount[+0])
charge[1-N]     = disc_price[+0] * (1 + tax[+0])
sum_charge[1]   = charge[1]
sum_charge[2-N] = charge[+0] + sum_charge[-1]
```

The `sum_charge` column is a running total, creating a dependency chain that grows linearly with row index. As the user scrolls down the page (under normal usage), the runtime to compute visible cells grows linearly. Each system loads the spreadsheet with a viewable area of 50 rows and updates a single cell. We measure (i) the cost of initialization and (ii) the cost of a single update. Time is measured until quiescence. To emulate batch processing, we replace the formula for the `sum_change`$[i-1]$ (where i is the first visible row) with a formula that computes the analogous aggregate query.

Moving View. Figure 5(a,c) shows costs for a fixed dataset size of approximately 600,000 rows, varying the viewable rows. Due to the running sum, later rows require more computation. Costs for Vizier and Dataspread grow significantly with the length of the dependency chain, while batch processing can compute the updated sum significantly faster.

Scaling Data. Figure 5(b,d) shows costs when varying data size, with the view fixed on the first cell. Because dependencies in the visible area are of constant size, Vizier is faster.

5 RELATED WORK

Although spreadsheets present a convenient interface to data, they lack the scalability to manage large data. A common approach to scaling spreadsheets (the "virtual" approach) reformulates the interface to an existing database or workflow system using spreadsheet-style direct manipulation metaphors [2, 10–12, 15]. The resulting systems bear varying levels of resemblance to existing spreadsheets, usually introducing concepts from relational databases like explicit tables, attributes, and records. Wrangler [12] is an ETL workflow development tool with an interface inspired by spreadsheets. Users open a small sample of a dataset in Wrangler and use spreadsheet-style direct manipulations to indicate desired changes to the dataset. Vizier [7, 8, 13, 14] is a computational notebook system that allows users to define workflow stages through a spreadsheet-style interface. Other approaches more directly mimic relational databases: The Spreadsheet Algebra [11, 15] allows users to specify any SPJGA-query purely through spreadsheet-style user interactions. Related Worksheets [2, 3] re-imagines the classical spreadsheet-style interface with record structure and inlined display of foreign-key references.

A second approach (the "materialized" approach) instead redesigns the spreadsheet engine itself through database concepts; An example is DataSpread [5, 6, 16]. A key challenge is that classical database techniques, which exploit common structures in a dataset, are not directly applicable. [5] explores data structures that can leverage partial structure; for example, when a range of cells are structured as a relational table. [6] explores strategies for quickly invalidating cells and computing dependencies, by leveraging a (lossy) compressed dependency graph that can efficiently bound a cell's downstream. [17] introduces a different type of compressed dependency graph which is lossless, instead exploiting repeating patterns in formulas. This is analogous to our own approach, but focuses on the dependency graph rather than expressions, limiting opportunities for optimization.

In summary, DataSpread introduced multiple efficient algorithms for storing, accessing, and updating spreadsheets. The virtual approach is often less efficient, but has the advantage of supporting light-weight versioning, tracking the provenance. Crucially, this approach also enables replaying a user's updates, originally applied to one dataset, on a new dataset (e.g., to re-apply curation work on an updated version of the data). The overlay approach we present in this work has the potential to retain these benefits while enabling performance competitive with DataSpread.

6 CONCLUSIONS AND FUTURE WORK

In this work, we introduced overlay spreadsheets as a potential direction for reproducible spreadsheets in workflow and provenance analysis systems like Vizier. This novel capability is powered by overlays that decouple the user's edits from the source data they are applied to. We also demonstrated how updates to ranges of cells can be represented declaratively, improving performance and enabling optimized evaluation of recursive patterns.

Recursive patterns remain the source of several open challenges for us. Most notably, in the absence of recursive patterns, the depth of a dependency chains is bounded by the number of user interactions. We suggested two strategies for improving performance in the presence of recursive patterns: (i) Closed-form computation of dependencies, and (ii) using bulk processing to avoid individual evaluation of cells that are not being shown to the user.

We also observe two additional challenges of adapting a dataset to new source data. As we noted, row identity is a critical challenge for updating source data, as each row in the updated dataset needs to be mapped to its corresponding row in the original. Additionally, the spreadsheet itself may need to change, for example extending patterns to incorporate newly introduced rows in the dataset.

REFERENCES

[1] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. 2015. Spark SQL: Relational Data Processing in Spark. In *SIGMOD Conference*. ACM, 1383–1394.

[2] Eirik Bakke and Edward Benson. 2011. The Schema-Independent Database UI: A Proposed Holy Grail and Some Suggestions. In *CIDR*. www.cidrdb.org, 219–222.

[3] Eirik Bakke, David R. Karger, and Rob Miller. 2011. A spreadsheet-based user interface for managing plural relationships in structured data. In *CHI*. ACM, 2541–2550.

[4] Mangesh Bendre, Bofan Sun, Ding Zhang, Xinyan Zhou, Kevin Chen-Chuan Chang, and Aditya G. Parameswaran. 2015. DATASPREAD: Unifying Databases and Spreadsheets. *PVLDB* 8, 12 (2015), 2000–2003.

[5] Mangesh Bendre, Vipul Venkataraman, Xinyan Zhou, Kevin Chen-Chuan Chang, and Aditya G. Parameswaran. 2018. Towards a Holistic Integration of Spreadsheets with Databases: A Scalable Storage Engine for Presentational Data Management. In *ICDE*. 113–124.

[6] Mangesh Bendre, Tana Wattanawaroon, Kelly Mack, Kevin Chang, and Aditya G. Parameswaran. 2019. Anti-Freeze for Large and Complex Spreadsheets: Asynchronous Formula Computation. In *SIGMOD*. 1277–1294.

[7] Mike Brachmann, Carlos Bautista, Sonia Castelo, Su Feng, Juliana Freire, Boris Glavic, Oliver Kennedy, Heiko Mueller, Remi Rampin, William Spoth, and Ying Yang. 2019. Data Debugging and Exploration with Vizier. In *SIGMOD*.

[8] Michael Brachmann, William Spoth, Oliver Kennedy, Boris Glavic, Heiko Mueller, Sonia Castelo, Carlos Bautista, and Juliana Freire. 2020. Your notebook is not crumby enough, REPLace it. In *CIDR*.

[9] The Transaction Processing Performance Council. 2018. TPC Benchmark H (Decision Support), Revision 2.18.0. https://www.tpc.org/tpch/default5.asp.

[10] Juliana Freire, Boris Glavic, Oliver Kennedy, and Heiko Mueller. 2016. The Exception That Improves The Rule. In *HILDA*.

[11] H. V. Jagadish, Adriane Chapman, Aaron Elkiss, Magesh Jayapandian, Yunyao Li, Arnab Nandi, and Cong Yu. 2007. Making database systems usable. In *SIGMOD Conference*. ACM, 13–24.

[12] Sean Kandel, Andreas Paepcke, Joseph M. Hellerstein, and Jeffrey Heer. 2011. Wrangler: interactive visual specification of data transformation scripts. In *CHI*. ACM, 3363–3372.

[13] Oliver Kennedy, Boris Glavic, Juliana Freire, and Mike Brachmann. 2022. The Right Tool for the Job: Data-Centric Workflows in Vizier. *IEEE-DEB* (2022).

[14] Poonam Kumari, Michael Brachmann, Oliver Kennedy, Su Feng, and Boris Glavic. 2021. DataSense: Display Agnostic Data Documentation. In *CIDR*.

[15] Bin Liu and H. V. Jagadish. 2009. A Spreadsheet Algebra for a Direct Data Manipulation Query Interface. In *ICDE*. IEEE Computer Society, 417–428.

[16] Sajjadur Rahman, Kelly Mack, Mangesh Bendre, Ruilin Zhang, Karrie Karahalios, and Aditya G. Parameswaran. 2020. Benchmarking Spreadsheet Systems. In *SIGMOD*. 1589–1599.

[17] Dixin Tang, Fanchao Chen, Christopher De Leon, Tana Wattanawaroon, Jeaseok Yun, Srinivasan Seshadri, and Aditya G. Parameswaran. 2023. Efficient and Compact Spreadsheet Formula Graphs. *CoRR* abs/2302.05482 (2023). arXiv:2302.05482