Scaling up to Billions of Cells with DATASPREAD: Supporting Large Spreadsheets with Databases

Mangesh Bendre, Vipul Venkataraman, Xinyan Zhou
Kevin Chen-Chuan Chang, Aditya Parameswaran
University of Illinois at Urbana-Champaign (UIUC)
{bendre1 | vvnktrm2 | xzhou14 | kcchang | adityagp}@illinois.edu

ABSTRACT

Spreadsheet software is the tool of choice for ad-hoc tabular data management, manipulation, querying, and visualization with adoption by billions of users. However, spreadsheets are not scalable, unlike database systems. We develop DATASPREAD, a system that holistically unifies databases and spreadsheets with a goal to work with massive spreadsheets: DATASPREAD retains all of the advantages of spreadsheets, including ease of use, ad-hoc analysis and visualization capabilities, and a schema-free nature, while also adding the scalability and collaboration abilities of traditional relational databases. We design DATASPREAD with a spreadsheet front-end and a regular relational database back-end. To integrate spreadsheets and databases, in this paper, we develop a storage and indexing engine for spreadsheet data. We first formalize and study the problem of representing and manipulating spreadsheet data within a relational database. We demonstrate that identifying the optimal representation is NP-HARD via a reduction from partitioning of rectangles; however, under certain reasonable assumptions, can be solved in PTIME. We develop a collection of mechanisms for representing spreadsheet data, and evaluate these representations on a workload of typical data manipulation operations. We augment our mechanisms with novel positionally-aware indexing structures that further improve performance. DATASPREAD can scale to billions of cells, returning results for common operations within seconds. Lastly, to motivate our research questions, we perform an extensive survey of spreadsheet use for ad-hoc tabular data management.

1. INTRODUCTION

Spreadsheet software, from the pioneering VisiCalc [12] to Microsoft Excel [2] and Google Sheets [1], have found ubiquitous use in ad-hoc manipulation, management, and analysis of tabular data. The billions who use spreadsheets take advantage of not only its ad-hoc nature and flexibility but also the in-built statistical and visualization capabilities. Spreadsheets cater to both novice and advanced users, spanning businesses, universities, organizations, government, and home.

Yet, this mass adoption of spreadsheets breeds new challenges. With the increasing sizes and complexities of data sets, as well as types of analyses, we see a frenzy to push the limits: users are struggling to work with spreadsheet software on large datasets; they are trying to import large data sets into Excel (*e.g.*, billions of gene-gene interactions) and are failing at doing so. In response, *spreadsheet softwares are stretching the size of data they can support*; *e.g.*, Excel has lifted its size limits from 65k to 1 million rows, and added Power Query and PowerPivot [46, 45] to support one-shot import of data from databases in 2010; Google Sheets has expanded its size limit to 2 million cells. Despite these developments, these moves are far from the kind of scale, *e.g.*, beyond memory limits, and functionality, *e.g.*, expressiveness, that databases natively provide.

This discussion raises the following question: ***can we leverage relational databases to support spreadsheets at scale?*** That is, can we retain the spreadsheet front-end that so many end-users are so comfortable with, while supporting that front-end with a standard relational database, seamlessly leveraging the benefits of scalability and expressiveness?

To address this question, our first challenge is to efficiently represent spreadsheet data within a database. First, notice that while databases natively use an unordered "set" semantics, spreadsheets utilize position as a first-class primitive, thus it is not natural to represent and store spreadsheet data in a database. Further, spreadsheets rarely obey a fixed schema — a user may paste several "tabular" or table-like regions within a spreadsheet, often interspersed with empty rows or columns. Users may also embed formulae into spreadsheets, along with data. This means that considering the entire sheet as a single relation, with rows corresponding to rows of the spreadsheet, and columns corresponding to columns of the spreadsheet, can be very wasteful due to sparsity. At the other extreme, we can consider only storing cells of the spreadsheet that are filled-in: we can simply store a table with schema (row number, column number, value): this can be effective for highly sparse spreadsheets, but is wasteful for dense spreadsheets with well-defined tabular regions. One can imagine hybrid representation schemes that use both "tabular" and "sparse" representation schemes as well or schemas that take access patterns, *e.g.*, via formulae, into account. In this paper, we show that it is NP-HARD to identify the optimal storage representation, given a spreadsheet. Despite this wrinkle, we characterize a certain natural subset of representations for which identifying the optimal one is in fact, PTIME; furthermore, we identify a collection of optimization techniques that further reduce the computation complexity to the point where the optimal representation can be identified in the time it takes to make a single pass over the data.

The next challenge is in supporting operations on the spreadsheet. Notice first that the most primitive operation on a spreadsheet is *scrolling* to an arbitrary position on a sheet. Unlike traditional databases, where order is not a first class citizen, spreadsheets require positionally aware access. This motivates the need for *positional indexes*; we develop and experiment with indexing mechanisms that adapt traditional indexing schemes, to take position into account. Furthermore, we need to support modifications to the spreadsheet. Note that even a small modification can be rather costly: inserting a single row can impact the row number of all subsequent rows. How do we support such modifications efficiently? We develop *positional mapping* schemes that allow us to avoid the expensive computation that results from small modifications.

By addressing the aforementioned challenges, we answer the question of whether *we can leverage relational databases to support spreadsheets at scale in the affirmative in this paper*. We build

a system, DATASPREAD, that can not only efficiently support operations on billions of records, but naturally incorporates relational database features such as expressiveness and collaboration support. DATASPREAD uses a standard relational database as a backend (currently PostgreSQL, but nothing ties us to that database), with a web-based spreadsheet system [3] as the frontend. By using a standard relational database, with no modifications to the underlying engine, we can just seamlessly leverage improvements to the database, while allowing the same data to be used by other applications. This allows a clean encapsulation and separation of front-end and back-end code, and also admits portability and a simpler design. DATASPREAD is fully functional — the DATASPREAD resources, along with video and code can be found at dataspread.github.io. We demonstrated a primitive version of DATASPREAD at the VLDB conference last year [11].

While there have been many attempts at combining spreadsheets and relational database functionality, ultimately, all of these attempts fall short because they do not let spreadsheet users perform ad-hoc data manipulation operations [47, 48, 28]. Other work supports expressive and intuitive querying modalities without addressing scalability issues [9, 5, 20], addressing an orthogonal problem. There have been efforts that enhance spreadsheets or databases without combining them [38]. Furthermore, while there has been work on array-based databases, most of these systems do not support edits: for instance, SciDB [13] supports an append-only, no-overwrite data model. We describe related work in more detail in Section 8.

Rest of the Paper. The outline of the rest of the paper is as follows.

- We begin with an empirical study of four real spreadsheet datasets, plus an on-line user survey, targeted at *understanding how spreadsheets are used for data analysis* in Section 2.
- Then, in Section 3, we introduce the notion of a *conceptual data model* for spreadsheet data, as well as the set of operations we wish to support on this data model.
- In Section 4, we propose three *primitive data models* for supporting the conceptual data model within a database, along with a *hybrid data model* that combines the benefits of these primitive data models. We demonstrate that identifying the optimal hybrid data model is NP-HARD, but we can develop a PTIME dynamic programming algorithm that allows us to find an approximately optimal solution.
- Then, in Section 5, we motivate the need for, and develop indexing solutions for *positional mapping*—a method for reducing the impact of cascading updates for inserts and deletes on all our data models.
- We give a brief overview of the system architecture from the perspective of our data models in Section 6. We also describe how we seamlessly support standard relational operations in DATASPREAD.
- We perform experiments to evaluate our data models and positional mapping schemes in Section 7, and discuss related work in Section 8.

2. SPREADSHEET USAGE IN PRACTICE

In this section, we empirically evaluate how spreadsheets are used for data management. We use the insights from this evaluation to both motivate the design decisions for DATASPREAD, and develop a realistic workload for spreadsheet usage. To the best of our knowledge, no such evaluation, focused on the usage of spreadsheets for data analytics, has been performed in the literature.

We focus on two aspects: (a) *structure:* identifying how users structure and manage data on a spreadsheet, and (b) *operations:* understanding the common spreadsheet operations that users perform.

To study these two aspects, we first retrieve a large collection of real spreadsheets from four disparate sources, and quantitatively analyze them on different metrics. We supplement this quantitative analysis with a small-scale user survey to understand the spectrum of operations frequently performed. The latter is necessary since we do not have a readily available trace of user operations from the real spreadsheets (*e.g.*, indicating how often users add rows or columns, or edit formulae.)

We first describe our methodology for both these evaluations, before diving into our findings for the two aspects.

2.1 Methodology

As described above, we have two forms of evaluation of spreadsheet use: the first, via an analysis of spreadsheets, and the second, via interviews of spreadsheet users. The datasets can be found at dataspread.github.io.

2.1.1 Real Spreadsheet Datasets

For our evaluation of real spreadsheets, we assemble four datasets from a wide variety of sources.

Internet. This dataset was generated by crawling the web for Excel (.xls) files, using a search engine, across a wide variety of domains. As a result, these 53k spreadsheets vary widely in content, ranging from tabular data to images.

ClueWeb09. This dataset of 26k spreadsheets was generated by extracting .xls file URLs from the ClueWeb09 [15] web crawl dataset.

Enron. This dataset was generated by extracting 18k spreadsheets from the Enron email dataset [26]. These spreadsheets were used to exchange data within the Enron corporation.

Academic. This dataset was collected from an academic institution; this academic institution used these spreadsheets to manage internal data about course workloads of instructors, salaries of staff, and student performance.

We list these four datasets along with some statistics in Table 1. Since the first two datasets are from the open web, they are primarily meant for *data publication*: as a result, only about 29% and 42% of these sheets (column 3) contain formulae, with the formulae occupying less than 3% of the total number of non-empty cells for both datasets (column 5). The third dataset is from a corporation, and is primarily meant for *data exchange*, with a similarly low fraction of 39% of these sheets containing formulae, and 3.35% of the non-empty cells containing formulae. The fourth dataset is from an academic institution, and is primarily meant for *data analysis*, with a high fraction of 91% of the sheets containing formulae, and 23.26% of the non-empty cells containing formulae.

2.1.2 User Survey

To evaluate the kinds of operations performed on spreadsheets, we solicited participants for a qualitative user survey: we recruited thirty participants from the industry who exclusively use spreadsheets for data management and analysis. This survey was conducted via an online form, with the participants answering a small number of multiple-choice and free-form questions, followed by the authors aggregating the responses.

2.2 Structure Evaluation

We now use our spreadsheet datasets to understand how data is laid out on spreadsheets.

Across Spreadsheets: Data Density. First, we study how similar real spreadsheets are to relational data conforming to a specific tabular structure. To study this, we estimate the *density* of each spreadsheet, defined as the ratio of the filled-in cells to the total number of cells—specified by the minimum bounding rectangular box enclosing the filled-in cells—within a spreadsheet. We depict the results

Dataset	Sheets	Sheets with formulae	Sheets with $> 20\%$ formulae	% of formulae	Sheets with $< 50\%$ density	Sheets with $< 20\%$ density
Internet	52311	29.15%	20.26%	1.30%	22.53%	6.21%
ClueWeb09	26148	42.21%	27.13%	2.89%	46.71%	23.8%
Enron	17765	39.72%	30.42%	3.35%	50.06%	24.76%
Academic	636	91.35%	71.26%	23.26%	90.72%	60.53%

Table 1: Spreadsheet Datasets: Preliminary Statistics

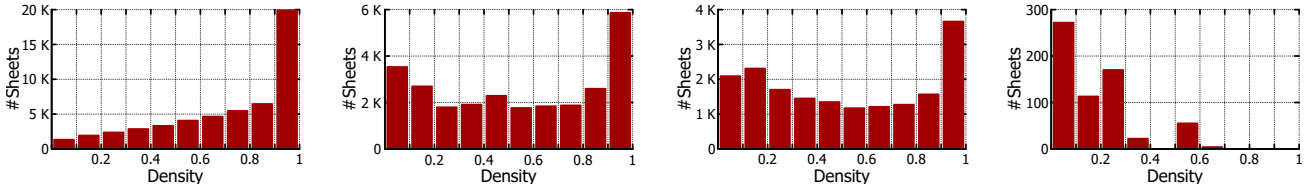


Figure 1: Data Density — (a) Internet (b) ClueWeb09 (c) Enron (d) Academic

in the last two columns of Table 1, and in Figure 1, which depicts the distribution of this ratio. We note that spreadsheets within Internet, Clueweb09, and Enron datasets are typically *dense*, *i.e.*, more than 50% of the spreadsheets have density greater than 0.5. On the other hand, for the Academic dataset, we note a high proportion (greater than 60%) of spreadsheets have density values less than 0.2. This low density is because the latter dataset embeds a number of formulae and use forms to report data in a user-accessible interface. Thus, we have:

Takeaway 1: Real spreadsheets vary widely in their density, ranging from highly sparse to highly dense, necessitating data models that can adapt to such variations.

Within a Spreadsheet: Tabular regions. For the spreadsheets that are sparse, we further analyzed them to evaluate whether there are regions within these spreadsheets with high density—essentially indicating that these regions can be regarded as tables. To identify these tabular regions, we first constructed a graph consisting of filled-in cells within each spreadsheet, where two cells (*i.e.*, nodes) have an edge between them if they are adjacent either vertically or horizontally. We then computed the connected components on this graph. We declare a connected component to be a tabular region if it spans at least two columns and five rows, and has an overall density of at least 0.7, defined as before as the ratio of the filled-in cells to the total number of cells in the minimum bounding rectangle encompassing the connected component. In Table 2, for each dataset, we list the total number of tabular regions identified (column 2), the number of filled-in cells covered by these regions (column 3), and the fraction of the total filled-in cells that are captured within these tabular regions (column 4).

Dataset	Tables	Table cells	%Coverage
Internet Crawl	67,374	124,698,013	66.03
ClueWeb09	37,164	52,257,649	67.68
Enron	9,733	8,135,241	60.98
Academic	286	18,384	12.10

Table 2: Tabular Regions in Spreadsheets.

For the Internet Crawl, ClueWeb09, and Enron datasets, we observe that greater than 60% of the cells are part of tabular regions. We also note that for the Academic dataset, where the sheets are rather sparse, there still are a modest number of regions that are tabular (286 across 636 sheets).

Takeaway 2: Even within a single spreadsheet, there is often high skew, with areas of both high density and low density, indicating the need for fine-grained data models that can treat these regions differently.

2.3 Operation Evaluation

We now move onto evaluating the operations performed on spreadsheets, both the formulae embedded with spreadsheets, as well as other data manipulation, viewing and modification operations.

Popularity: Formulae Usage. We begin by studying how often formulae are used within spreadsheets. On examining Table 1, we find that there is a high variance in the fraction of cells that are formulae (column 5), ranging from 1.3% to 23.26%. We note that the academic institution dataset embeds a high fraction of formulae, indicating that the spreadsheets in that case are used primarily for data management and analysis as opposed to data sharing or publication. Despite that, all of the datasets have a substantial fraction of spreadsheets where the formulae occupy more than 20% of the cells (column 4)—20.26% and higher for all datasets.

Takeaway 3: Formulae are very common in spreadsheets, with over 20% of the spreadsheets containing a large fraction of over $\frac{1}{5}$ of formulae, across all datasets. The high prevalence of formulae indicates that optimizing for the access patterns of formulae when developing data models is crucial.

Access: Formulae Distribution and Access Patterns. Next, we study the distribution of formulae used within spreadsheets—see Figure 2. Not surprisingly, arithmetic operations are very common across all datasets. The first three datasets have an abundance of conditional formulae through IF statements (*e.g.*, second bar in Figure 2a)—these statements were typically used to fill in missing data or to change the data type, *e.g.*, IF(H67=true,1.0,0.0). In contrast, the Academic dataset is dominated by formulae on numeric data. Overall, there is a wide variety of formulae that span both a small number of cell accesses (*e.g.*, arithmetic), as well as a large number of them (*e.g.*, SUM, VL short for VLOOKUP). The last two correspond to standard database operations such as aggregation and joins.

Dataset	Total Cells Accessed	Cells accessed per Formula	Components Per Formula
Internet	2,460,371	334.26	2.5
ClueWeb09	2,227,682	147.99	1.92
Enron	446,667	143.05	1.75
Academic	35,335	3.03	1.54

Table 3: Cells accessed by formulae.

To gain a better understanding of how much effort is necessary to execute these formulae, we measure the number of cells accessed by each formula. Then, we tabulate the average number of cells accesses per formula in column 3 of Table 3 for each dataset. As we can see in the table, the average number of cells accesses per formula is not small—with up to 300+ cells per formula for the Internet dataset, and about 140+ cells per formula for the Enron and ClueWeb09 datasets. The Academic dataset has a smaller average number—many of these formulae correspond to derived columns that access a small number of cells at a time. Next, we wanted to check if the accesses made by these formulae were spread across the spreadsheet, or could exploit spatial locality. To measure this, we considered the set of cells accessed by each formula, and then generated the corresponding graph of these accessed cells as described in the previous subsection for computing the number of

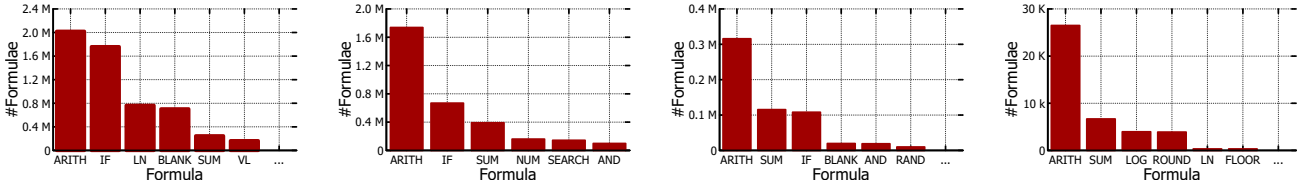


Figure 2: Formulae Distribution — (a) Internet (b) ClueWeb09 (c) Enron (d) Academic

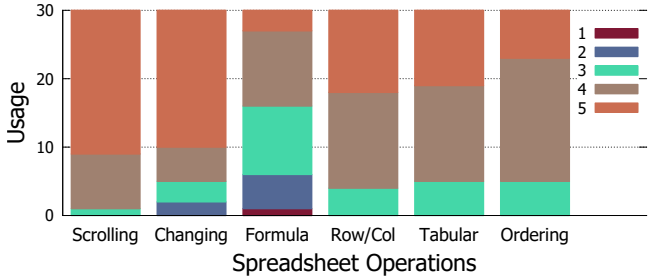


Figure 3: Operations performed on spreadsheets.

tabular regions. We then counted the number of connected components in this graph, and tabulated the results in column 4 in the same table. As can be seen, even though the number of cells accessed may be large, these cells stem from a small number of connected components; as a result, we can exploit spatial locality to execute them more efficiently.

Takeaway 4: Formulae on spreadsheets access cells on the spreadsheet by position; some common formulae such as SUM or VLOOKUP access a rectangular range of cells at a time. The number of cells accessed by these formulae can be quite large, and most of these cells stem from contiguous areas of the spreadsheet.

User-Identified Operations. In addition to identifying how users structure and manage data on a spreadsheet, we now analyze the common spreadsheet operations that users perform. To this end, we conducted a small-scale online survey of 30 participants to study how users operate on spreadsheet data. This qualitative study is valuable since real spreadsheets do not reveal traces of user operations performed on them (*e.g.*, revealing how often users perform ad-hoc operations like scrolling, sorting, deleting rows or columns). Our questions in this study were targeted at understanding (a) how users perform operations on the spreadsheet and (b) how users organize data on the spreadsheet.

With the goal of understanding how users perform operations on the spreadsheet, we asked each participant to answer a series of questions where each question corresponded to whether they conducted the specific operation under consideration on a scale of 1–5, where 1 corresponds to "never" and 5 to "frequently". For each operation, we plotted the results in a stacked bar chart in Figure 3, with the higher numbers stacked on the smaller ones like the legend indicates.

We find that all the thirty participants perform *scrolling*, *i.e.*, moving up and down the spreadsheet to examine the data, with 22 of them marking 5 (column 1). All participants reported to have performed editing of individual cells (column 2), and many of them reported to have performed formula evaluation frequently (column 3). Only four of the participants marked < 4 for some form of *row/column-level operations*, i.e., deleting or adding one or more rows or columns at a time (column 4).

Takeaway 5: There are several common operations performed by spreadsheet users including scrolling, row and column modification, and editing individual cells.

Our second goal for performing the study was to understand how users organize their data on a spreadsheet. We asked each participant if their data is organized in well-structured tables, or if the data scattered throughout the spreadsheet, on a scale of 1 (not organized)–5 (highly organized)—see Figure 3. Only five participants marked < 4 which indicates that users do organize their data on a spreadsheet (column 5). We also asked the importance of ordering of records in the spreadsheet on a scale of 1 (not important)–5 (highly important). Unsurprisingly, only five participants marked < 4 for this question (column 6). We also provided a free-form textual input where multiple participants mentioned that ordering comes naturally to them and is often taken for granted while using spreadsheets.

Takeaway 6: Spreadsheet users typically try to organize their data as far as possible on the spreadsheet, and rely heavily on the ordering and presentation of the data on their spreadsheets.

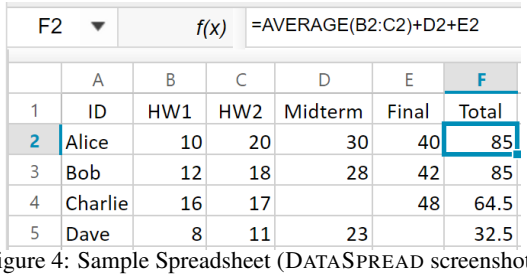
3. SPREADSHEET DESIDERATA

The goal for DATASPREAD is to combine the ease of use and interactivity of spreadsheets, while simultaneously providing the scalability, expressiveness, and collaboration capabilities of databases. Thus, as we develop DATASPREAD, having two aspects of interest: first, how do we support spreadsheet semantics over a database backend, and second, how do we support database operations within a spreadsheet. Our primary focus will be on the former, which will occupy the bulk of the paper. We return to the latter in Section 6. For now, we focus on describing the desiderata for supporting spreadsheet semantics over databases. We first describe our conceptual spreadsheet data model, and then describe the desired operations that need to be supported on this conceptual data model.

Conceptual Data Model. A spreadsheet consists of a collection of *cells*. A cell is referenced by two dimensions: row and column. Columns are referenced using letters A, . . ., Z, AA, . . .; while rows are referenced using numbers 1, 2, . . . Each cell contains either a *value*, or a *formula*. A value is a constant belonging to some fixed type. For example, in Figure 4 a screenshot from our working implementation of DATASPREAD, B2 (column B, row 2) contains the value 10. In contrast, a formula is a mathematical expression that contains values and/or cell references as arguments, to be manipulated by operators or functions. The expression corresponding to a formula eventually unrolls into a value. For example, in Figure 4, cell F2 contains the formula =AVERAGE(B2:C2)+D2+E2, which unrolls into the value 85. The value of F2 depends on the value of cells B2, C2, D2, and E2, which appear in the formula associated with F2.

In addition to a value or a formula, a cell could also additionally have formatting associated with it; *e.g.*, a cell could have a specific width, or the text within a cell can have bold font, and so on. For simplicity, we ignore formatting aspects, but these aspects can be easily captured within our representation schemes without significant changes.

Spreadsheet Operations. We now describe the operations that we aim to support on DATASPREAD, drawing from the operations we found in our user survey (takeaway 5). We consider the following read-only operations:



	A	B	C	D	E	F
1	ID	HW1	HW2	Midterm	Final	Total
2	Alice	10	20	30	40	85
3	Bob	12	18	28	42	85
4	Charlie	16	17		48	64.5
5	Dave	8	11	23		32.5

Figure 4: Sample Spreadsheet (DATASPREAD screenshot).

- **Scrolling:** This operation refers to the act of retrieving cells within a certain range of rows and columns. For instance, when a user scrolls to a specific position on the spreadsheet, we need to retrieve a rectangular range corresponding to the window that is visible to the user. Accessing an entire row or column, *e.g.*, A:A, is a special case of rectangular range where the column/row corresponding to the range is not bounded.
- **Formula evaluation:** Evaluating formulae can require accessing multiple individual cells (*e.g.*, A1) within the spreadsheet or ranges of cells (*e.g.*, A1:D100).

Note that in both cases, the accesses correspond to rectangular regions of the spreadsheet. We consider the following four update operations:

- **Updating an existing cell:** This operation corresponds to accessing a cell with a specific row and column number and changing its value. Along with cell updates, we are also required to reevaluate any formulae dependent on the cell.
- **Inserting/Deleting row/column(s):** This operation corresponds to inserting/deleting row/column(s) at a specific position on the spreadsheet, followed by shifting subsequent row/column(s) appropriately.

Note that, similar to read-only operations, the update operations require updating cells corresponding to rectangular regions.

In the next section, we develop data models for representing the conceptual data model as described in this section, with an eye towards supporting the operations described above.

4. REPRESENTING SPREADSHEETS

We now address the problem of representing a spreadsheet within a relational database. For the purpose of this section and the next, we focus on representing one spreadsheet, but our techniques seamlessly carry over to the multiple spreadsheet case; like we described earlier, we focus on the content of the spreadsheet as opposed to the formatting, as well as other spreadsheet metadata, like spreadsheet name(s), spreadsheet dimensions, and so on.

We describe the high-level problem of representation of spreadsheet data here; we will concretize this problem subsequently.

4.1 High-level Problem Description

The conceptual data model corresponds to a collection of cells, represented as $C = \{C_1, C_2, \ldots, C_m\}$; as described in the previous section, each cell C_i corresponds to a location (i.e., a specific row and column), and has some contents—either a value or a formula. Our goal is to represent and store the cells C comprising the conceptual data model, via one of the *physical data models*, $\mathbb{P}$. Each $T \in \mathbb{P}$ corresponds to a collection of relational tables $\{T_1, \ldots, T_p\}$. Each table T_i records the data in a certain portion of the spreadsheet, as we will see subsequently. Given a collection C, a physical data model T is said to be *recoverable* with respect to C if for each $C_i \in C, \exists T_j \in T$ such that T_j records the data in C_i, and $\forall k \neq j, T_k$ does not record the data in C_i. Thus, our goal is to identify physical data models that are recoverable.

At the same time, we want to minimize the amount of storage required to record T within the database, i.e., we would like to minimize $\text{size}(T) = \sum_{i=1}^{p} \text{size}(T_i)$. Moreover, we would like to minimize the time taken for accessing data using T, *i.e.*, the *access cost*, which is the cost of accessing a rectangular range of cells for formulae (takeaway 4) or scrolling to specific locations (takeaway 5), which are both common operations. And we would like to minimize the time taken to perform updates, *i.e.*, the *update cost*, which is the cost of updating individual cells or a range of cells, and the insertion and deletion of rows and columns.

Overall, starting from a collection of cells C, our goal is to identify a physical data model T such that: (a) T is recoverable with respect to C, and (b) T minimizes a combination of storage, access and update costs, among all $T \in \mathbb{P}$.

We begin by considering the setting where the physical data model T has a single relational table, *i.e.*, $T = \{T_1\}$. We develop three ways of representing this table: we call them *primitive data models*, and are all drawn from prior work, each of which work well for a specific structure of spreadsheet—this is the focus of Section 4.2. Then, we extend this to the setting where $|T| > 1$ by defining the notion of a *hybrid data model* with multiple tables each of which uses one of the three primitive data models to represent a certain portion of the spreadsheet—this is the focus of Section 4.3. Given the high diversity of structure within spreadsheets and high skew (takeaway 2), having multiple primitive data models, and the ability to use multiple tables, gives us substantial power in representing spreadsheet data.

4.2 Primitive Data Models

Our primitive data models represent trivial solutions for spreadsheet representation with a single table. Before we describe these data models, we discuss a small wrinkle that affects all of these models. To capture a cell's identity, *i.e.*, its row and column number, we need to implicitly or explicitly record a row and column number with each cell. Say we use an attribute to capture the row number for a cell. Then, the insertion or deletion of rows requires cascading updates to the row number attribute for all subsequent rows. As it turns out, all of the data models we describe in this section suffer from performance issues arising from cascading updates, but the solution to deal with these issues is similar for of these all of them, and will be described in Section 5.

Also, note that the access and update cost of various data models depends on whether the underlying database is a row store or a columnar store. For the rest of this section and the paper, we focus on a row store, such as PostgreSQL, which is what we use in practice, and is also more tailored for hybrid read-write settings. We now describe the three primitive data models:

Row-Oriented Model (ROM). The row-oriented data model (ROM) is straightforward, and is akin to data models used in traditional relational databases. Let r_{max} and c_{max} represent the maximum row number and column number across all of the cells in C. Then, in the ROM model, we represent each row from row 1 to r_{max} as a separate tuple, with an attribute for each column $Col1 \ldots, Colc_{max}$, and an additional attribute for explicitly capturing the row identity, *i.e.*, $RowID$. The schema for ROM is: ROM($\underline{RowID}$, $Col1$, $\ldots$, $Colc_{max}$)—we illustrate the ROM representation of Figure 4 in Figure 5: each entry is a pair corresponding to a value and a formula, if any. For dense spreadsheets that are tabular (takeaways 1 and 2), this data model can be quite efficient in storage and access, since it minimizes redundant information: each row number is recorded only once, independent of the number of columns. Overall, the ROM representation shines when entire rows are accessed at a time, as opposed to entire columns. It is also efficient for accessing a large range of cells at a time.

Column-Oriented Model (COM). The second representation is also straightforward, and is simply the transpose of the ROM representation. Often, we find that certain spreadsheets have many

RowID	Col_1	...	Col_6
1	ID, NULL	...	Total, NULL
2	Alice, NULL	...	85, AVERAGE(B2:C2)+D2+E2
...	...	...	...

Figure 5: ROM Data Model for Figure 4.

columns and relatively few rows, necessitating such a representation. The schema for COM is: COM($\underline{ColID}$, $Row1$, . . ., $Rowr_{max}$). The COM representation of Figure 4 is provided in Figure 6. Like ROM, COM shines for dense data; while ROM shines for row-oriented operations, COM shines for column-oriented operations.

ColID	Row_1	...	Row_5
1	ID,NULL	...	Dave,NULL
2	HW1,NULL	...	8,NULL
...	...	...	...

Figure 6: COM Data Model for Figure 4.

Row-Column-Value Model (RCV). The Row-Column-Value Model (RCV) is inspired by key-value stores, where the Row-Column number pair is treated as the key, *i.e.*, the row and column identifiers are explicitly captured as two attributes. The schema for RCV is RCV($\underline{RowID}$, $\underline{ColID}$, $Value$). The RCV representation for Figure 4 is provided in Figure 7. For sparse spreadsheets that are often found in practice (takeaway 1 and 2), this model is quite efficient in storage and access since it records only the cells that are filled in, but for dense spreadsheets, it incurs the additional cost of recording and retrieving both the row and column number for each cell as compared to ROM and COM, and has a much larger number of tuples. RCV is also efficient when it comes to retrieving specific cells at a time.

RowID	*ColID*	Value
1	1	ID, NULL
...	...	..., ...
2	2	10, NULL
...	...	..., ...
2	6	85, AVERAGE(B2:C2)+D2+E2
...	...	..., ...

Figure 7: RCV Data Model for Figure 4.

4.3 Hybrid Data Model: Intractability

So far, we developed three primitive data models, that represent reasonable extremes if we are to represent and store a spreadsheet within a single table in a database system. If, however, we do not limit data models to have a single table, we may be able to develop even better solutions by combining the benefits of the three primitive data models, and decomposing the spreadsheet into multiple tables each of which is represented by one of the primitive data models. We call these data models as *hybrid data models*.

DEFINITION 1 (HYBRID DATA MODELS). *Given a collection of cells C, we define* hybrid data models *to the space of physical data models that are formed using a collection of tables T such that the T is recoverable with respect to C, and further, each $T_i \in T$ is either a ROM, COM, or an RCV table.*

As an example, for the spreadsheet in Figure 8, we might want the dense areas, *i.e.*, B1:D4 and D5:G7, represented via a ROM or COM table each and the remaining area, specifically, H1 and I2 to be represented by an RCV table.

Cost Model. Next, the question is how do we model the cost for a specific hybrid data model. As discussed earlier, the storage, the access cost, and the update cost all impact our choice of hybrid data model. For the purpose of this section, we will focus on exclusively on the storage. We will generalize to the access cost in Appendix B. The update cost will be the focus of the next section. Furthermore, our focus will now be on ROM tables; we will generalize to RCV and COM tables in Section 4.6.

Given a hybrid data model $T = \{T_1, \ldots, T_p\}$, where each ROM table T_i has r_i rows and c_i columns, the cost of T is defined as

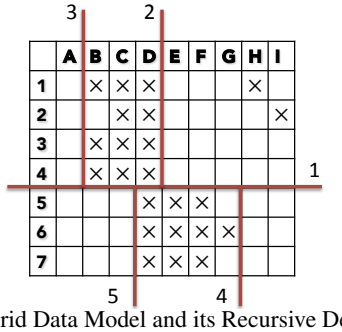


Figure 8: Hybrid Data Model and its Recursive Decomposition

$$\text{cost}(T) = \sum_{i=1}^{p} s_1 + s_2 \cdot (r_i \times c_i) + s_3 \cdot c_i + s_4 \cdot r_i. \quad (1)$$

Here, the constant s_1 is the cost of initializing a new table, as well as storing table-related metadata, while the constant s_2 is the cost of storing each individual cell (empty or not) in the ROM table. Note that the non-empty cells that have content may require even more space than s_2; however this is a constant cost that does not depend on the specific hybrid data model instance, and hence is excluded from the cost above. The constant s_3 is the cost corresponding to each column, while s_4 is the cost corresponding to each row. The former is necessary to record schema information per column, while the latter is necessary to record the row information in the RowID attribute. Overall, while the specific costs s_i may differ quite a bit across different database systems, what is clear is that all of these different costs matter.

Formal Problem. We are now ready to state our formal problem below.

PROBLEM 1 (HYBRID-ROM). *Given a spreadsheet with a collection of cells C, identify the hybrid data model T with only ROM tables that minimizes* $\text{cost}(T)$.

Unfortunately, Problem 1 is NP-HARD, via a reduction from the minimum edge length partitioning problem [27] of rectilinear polygons—the problem of finding a partitioning of a polygon whose edges are aligned to the X and Y axes, into rectangles, while minimizing the total sum of the perimeter of the resulting rectangles.

THEOREM 1 (INTRACTABILITY). *Problem 1 is* NP-HARD.

We formally show the hardness of the problem in Appendix A.

4.4 Optimal Recursive Decomposition

Instead of directly solving Problem 1, which is intractable, we instead aim to make it tractable, by reducing the search space of solutions. In particular, we focus on hybrid data models that can be obtained by *recursive decomposition*. Recursive decomposition is a process where we repeatedly subdivide the spreadsheet area from $[1 \ldots r_{max}, 1 \ldots c_{max}]$ by using a vertical cut between two columns or a horizontal cut between two rows, and then recurse on the two areas that are formed. As an example, in Figure 8, we can make a cut along line 1 horizontally, giving us two regions from rows 1 to 4 and rows 5 to 6. We can then cut the top portion along line 2 vertically, followed by line 3, separating out one table B1:D4. By cutting the bottom portion along line 4 and line 5, we can separate out the table D5:G7. Further cuts can help us carve out tables out of H1 or I2, not depicted here.

As the example illustrates, recursive decomposition is very powerful, since it captures a broad space of hybrid models; basically anything that can be obtained via recursive cuts along the x and y axis. Now, a natural question is: what sorts of hybrid data models cannot be composed via recursive decomposition? We present an example in Figure 9(a).

OBSERVATION 1 (COUNTEREXAMPLE). *In Figure 9(a), the tables: A1:B4, D1:I2, A6:F7, H4:I7 can never be obtained via recursive decomposition.*

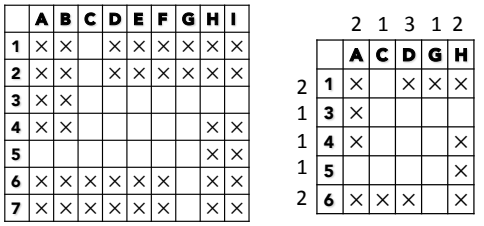


Figure 9: (a) Counterexample (b) Weighted Representation

To see this, note that any vertical or horizontal cut that one would make at the start would cut through one of the four tables, making the decomposition impossible. Nevertheless, the hybrid data models obtained via recursive decomposition form a natural class of data models.

As it turns out, identifying the solution to Problem 1 is PTIME for the space of hybrid data models obtained via recursive decomposition. The algorithm involves dynamic programming. Informally, our algorithm makes the most optimal "cut" horizontally or vertically at every step, and proceeds recursively. We now describe the dynamic programming equations.

Consider a rectangular area formed from x_1 to x_2 as the top and bottom row numbers respectively, both inclusive, and from y_1 to y_2 as the left and right column numbers respectively, both inclusive, for some x_1, x_2, y_1, y_2. We represent the optimal cost by the function $\mathrm{Opt}()$. Now, the optimal cost of representing this rectangular area, *i.e.*, $\mathrm{Opt}((x_1, y_1), (x_2, y_2))$, is the minimum of the following possibilities:

- If there is no filled cell in the rectangular area $(x_1, y_1), (x_2, y_2)$, then we do not use any data model. Hence, we have

$$\mathrm{Opt}((x_1, y_1), (x_2, y_2)) = 0 \tag{2}$$

- Do not split, *i.e.*, store as a ROM model ($\mathrm{romCost}()$):

$$\mathrm{romCost}((x_1, y_1), (x_2, y_2)) = s_1 + s_2 \cdot (r_{12} \times c_{12}) + s_3 \cdot c_{12} + s_4 \cdot r_{12}, \tag{3}$$

  where number of rows $r_{12} = (x_2 - x_1 + 1)$, and the number of columns $c_{12} = (y_2 - y_1 + 1)$.
- Perform a horizontal cut (C_H):

$$C_H = \min_{i \in \{x_1, \ldots, x_2\}} \mathrm{Opt}((x_1, y_1), (i, y_2)) + \mathrm{Opt}((i+1, y_1), (x_2, y_2)). \tag{4}$$

- Perform a vertical cut (C_V):

$$C_V = \min_{j \in \{y_1, \ldots, y_2\}} \mathrm{Opt}((x_1, y_1), (x_2, j)) + \mathrm{Opt}((x_1, j+1), (x_2, y_2)). \tag{5}$$

Therefore, when there are filled cells in the rectangle,

$$\mathrm{Opt}((x_1, y_1), (x_2, y_2)) = \min\left(\mathrm{romCost}((x_1, y_1), (x_2, y_2)), C_H, C_V\right). \tag{6}$$

else $\mathrm{Opt}((x_1, y_1), (x_2, y_2)) = 0$.

The base case is when the rectangular area is of dimension 1×1. Here, we store the area as a ROM table if it is a filled cell. Hence, we have, $\mathrm{Opt}((x_1, y_1), (x_1, y_1)) = c_1 + c_2 + c_3 + c_4$, if filled, and 0 if not.

We have the following theorem:

THEOREM 2 (DYNAMIC PROGRAMMING OPTIMALITY). *The optimal ROM-based hybrid data model based on recursive decomposition can be determined via dynamic programming.*

Time Complexity. Our dynamic programming algorithm runs in polynomial time with respect to the size of the spreadsheet. Let the length of the larger side of the minimum enclosing rectangle of the spreadsheet is of size n. Then, the number of candidate rectangles is $\mathcal{O}(n^4)$. For each rectangle, we have $\mathcal{O}(n)$ ways to perform the cut. Therefore, the running time of our algorithm is $\mathcal{O}(n^5)$. However, this number could be very large if the spreadsheet is massive—which typical of the use-cases we aim to tackle.

Weighted Representation. We now describe a simple optimization that helps us reduce the time complexity substantially, while preserving optimality for the cost model that we have been using so far. Notice that in many real spreadsheets, there are many rows and columns that are very similar to each other in structure, *i.e.*, they have the same set of filled cells. We exploit this property to reduce the effective size n of the spreadsheet. Essentially, we collapse rows that have identical structure down to a single weighted row, and similarly collapse columns that have identical structure down to a single weighted column.

Consider Figure 9(b) which shows the weighted version of Figure 9(a). Here, we can collapse column B down into column A, which is now associated with weight 2; similarly, we can collapse row 2 into row 1, which is now associated with weight 2. In this manner, the effective area of the spreadsheet now becomes 5×5 as opposed to 7×9.

Now, we can apply the same dynamic programming algorithm to the weighted representation of the spreadsheet: in essence, we are avoiding making cuts "in-between" the weighted edges, thereby reducing the search space of hybrid data models. As it turns out, this does not sacrifice optimality, as the following theorem shows:

THEOREM 3 (WEIGHTED OPTIMALITY). *The optimal hybrid data model obtained by recursive decomposition on the weighted spreadsheet is no worse than the optimal hybrid data model obtained by recursive decomposition on the original spreadsheet.*

4.5 Greedy Decomposition Algorithms

Greedy Decomposition. To improve the running time even further, we propose a greedy heuristic that avoids the high complexity of the dynamic programming algorithm, but sacrifices somewhat on optimality. The greedy algorithm essentially repeatedly splits the spreadsheet area in a top-down manner, making a greedy locally optimal decision, instead of systematically considering all alternatives, like in the dynamic programming algorithm. Thus, at each step, when operating on a rectangular spreadsheet area $(x_1, y_1), (x_2, y_2)$, it identifies the operation that results in the lowest local cost. We have three alternatives: Either we do not split, in which case the cost is from Equation 3, *i.e.*, $\mathrm{romCost}((x_1, y_1), (x_2, y_2))$. Or we split horizontally (vertically), in which case the cost is the same as C_H (C_V) from Equation 4 (Equation 5), but with $\mathrm{Opt}()$ replaced with $\mathrm{romCost}()$, since we are making a locally optimal decision. The smallest cost decision is followed, and then we continue recursively decomposing using the same rule on the new areas, if any.

Complexity. This algorithm has a complexity of $\mathcal{O}(n^2)$, since each step takes $\mathcal{O}(n)$ and there are $\mathcal{O}(n)$ steps. While the greedy algorithm is sub-optimal, the local decision that it makes is *optimal in the worst case*, *i.e.*, with no further information about the structure of the areas that arise as a result of the decomposition, this is the best decision to make at each step.

Aggressive Greedy Decomposition. The greedy algorithm described above stops exploration as soon as it is unable to find a cut that reduces the cost locally, based on a worst case assumption. This may cause the algorithm to halt prematurely, even though exploring further decompositions may have helped reduce the cost. An alternative to the greedy algorithm described above is one where we don't stop subdividing, *i.e.*, we always choose to use the best

horizontal or vertical cut, and then subdivide the area based on that cut in a *depth-first* manner. We keep doing this until we end up with rectangular areas where all of the cells are filled in with values. (At this point, it provably doesn't benefit us to subdivide further.) After this point, we backtrack up the tree of decompositions, bottom-up, assembling the best solution that was discovered, similar to the dynamic programming approach, considering whether to not split, or perform a horizontal or vertical split.

Complexity. Like the greedy approach, the aggressive greedy approach has complexity $\mathcal{O}(n^2)$, but takes longer since it considers a larger space of data models than the greedy approach.

4.6 Extensions

In this section, we describe extensions to the cost model and algorithms to handle COM and RCV tables in addition to ROM. Other extensions can be found in Appendix B, including incorporating access cost along with storage, including the costs of indexes, and dealing with situations when database systems impose limitations on the number of columns in a relation. We will describe these extensions to the cost model, and then describe the changes to the basic dynamic programming algorithm; modifications to the greedy and aggressive greedy decomposition algorithms are straightforward.

RCV and COM. The cost model can be extended in a straightforward manner to allow each rectangular area to be a ROM, COM, or an RCV table. First, note that it doesn't benefit us to have multiple RCV tables—we can simply combine all of these tables into one, and assume that we're paying a fixed up-front cost to have one RCV table. Then, the cost for a table T_i, if it is stored as a COM table is:

$$\text{comCost}(T_i) = s_1 + s_2 \cdot (r_i \times c_i) + s_4 \cdot c_i + s_3 \cdot r_i.$$

This equation is the same as Equation 1, but with the last two constants transposed. And the cost for a table T_i, if it is stored as an RCV table is simply:

$$\text{rcvCost}(T_i) = s_5 \times \#\text{cells}.$$

where s_5 is the cost incurred per tuple. Once we have this cost model set up, it is straightforward to apply dynamic programming once again to identify the optimal hybrid data model encompassing ROM, COM, and RCV. The only step that changes in the dynamic programming equations is Equation 3, where we have to consider the COM and RCV alternatives in addition to ROM. We have the following theorem.

THEOREM 4 (OPTIMALITY WITH ROM, COM, AND RCV). *The optimal ROM, COM, and RCV-based hybrid data model based on recursive decomposition can be determined via dynamic programming.*

5. POSITIONAL MAPPING

As discussed in Section 4, for all of the data models, storing the row and/or column numbers may result in substantial overheads during insert and delete operations due to cascading updates to all subsequent rows or columns—this could make working with large spreadsheets infeasible. In this section, we develop solutions for this problem by introducing the notion of *positional mapping* to eliminate the overhead of cascading updates. For our discussion we focus on row numbers; the techniques can be analogously applied to columns. To keep our discussion general, we use the term *position* to represent the ordinal number, *i.e.*, either row or column number, that captures the location of the cell along a specific dimension. In addition, row and column numbers can be dealt with independently.

Problem. We require a data structure to efficiently support positional operations without the overhead of cascading updates. In

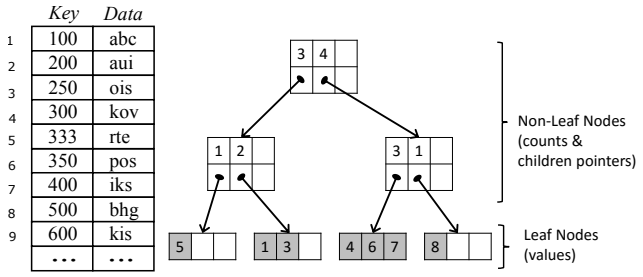


Figure 10: (a) Monotonic Positional Mapping (b) Index for Hierarchical Positional Mapping

particular, we want a data structure on items (here tuples) that can capture a specific ordering among the items and efficiently support the following operations: *(a) fetch* items based on a position, *(b) insert* items at a position, and *(c) delete* items from a position. The insert and delete operations require updating the positions of the subsequent items, *e.g.*, inserting an item at the n^{th} position requires us to first increment by one the positions of all the items that have a position greater than or equal to n, and then add the new item at the n^{th} position. Due to the interactive nature of DATASPREAD, our goal is to perform these operations within a few hundred milliseconds.

Row Number as-is. We motivate the problem by demonstrating the impact of cascading updates in terms of time complexity. Storing the row numbers as-is with every tuple makes the fetch operation efficient at the expense of making the insert and delete operations inefficient. With a traditional index, *e.g.*, a B-Tree index, the complexity to access an arbitrary row identified by a row number is $\mathcal{O}(\log N)$. On the other hand, insert and delete operations require updating the row numbers of the subsequent tuples. These updates also need to be propagated in the index, and therefore it results in a worst case complexity of $\mathcal{O}(N \log N)$. To illustrate the impact of these complexities on practice, in Table 4(a), we display the performance of storing the row numbers as-is for two operations—fetch and insert—on a spreadsheet containing 10^6 cells. We note that irrespective of the data model used, the performance of inserts is beyond our acceptable threshold whereas that of the fetch operation is acceptable.

Row Number as-is

Operation	RCV	ROM
Insert	87,821	1,531
Fetch	312	244

Positional Mapping

Operation	RCV	ROM
Insert	9.6	1.2
Fetch	30,621	273

Table 4: The performance of (in ms) (a) storing Row Number as-is (b) Monotonic Positional Mapping.

Intuition. To improve the performance of inserts and deletes for ordered items, we introduce the idea of *positional mapping*. At its core, the idea is remarkably simple: we do not store positions but instead store what we call *positional mapping keys*. These positional mapping keys p are proxies that have a one-to-one mapping with the positions r, *i.e.*, $p \rightleftarrows r$. Formally, positional mapping $\mathbb{M}$ is a bijective function that maintains the relationship between the row numbers and positional mapping keys, *i.e.*, $\mathbb{M}(r) \rightarrow p$.

Monotonic Positional Mapping. One approach towards positional mapping is to have positional mapping keys monotonically increase with position, *i.e.*, for two arbitrary positions r_i and r_j, if $r_i > r_j$ then $\mathbb{M}(r_j) > \mathbb{M}(r_i)$. For example, consider the ordered list of items shown in Figure 10(a). Here, even though the positional mapping keys do not correspond to the row number, and even though there can be arbitrary differences between consecutive positional mapping keys, we can fetch the n^{th} record by scanning the positional mapping keys in an increasing order while maintaining a running counter to skip n-1 records. The gaps between the consecutive

positional mapping keys reduce or even eliminate the renumbering during insert and delete operations.

Thus, monotonic positional mapping trades-off the performance of the fetch operation for making insert and delete operations efficient. To fetch the n^{th} item, in the absence of the stored position we need to scan n items, *i.e.*, the average time complexity is $\mathcal{O}(N)$, where N is the total number of items. If we know the positional mapping key of the item we are fetching (which is often not the case), and we have a traditional B+tree index on this key, then the complexity of this operation is $\mathcal{O}(\log N)$. Similarly, the complexity of inserting an item if we know the positional mapping key, determined based on the positional mapping keys of neighboring items, is $\mathcal{O}(\log N)$, which is the effort spent to update the underlying indexing structure. In Table 4(b), we experimentally observe that benefits from monotonic positional mapping for the insert operations come at the expense of the fetch operation, leading to unacceptable latencies.

Hierarchical Positional Mapping. We now describe a scheme, titled *hierarchical positional mapping*, that enhances monotonic positional mapping, by adding a new indexing structure that alleviates the cost of insert and delete operations, while not sacrificing the performance of the fetch operation. This new indexing structure adapts classical work on *order-statistic trees* [19]. Just like a typical B+Tree is used to capture the mapping from keys to the corresponding records, we can use the same structure to map positions to positional mapping keys. Here, instead of storing a key we store the count of elements stored within the entire sub-tree. The leaf nodes store the values, while the remaining nodes store pointers to the children along with counts.

For the positional mapping shown in Figure 10(a), we show the corresponding hierarchical positional mapping index structure in Figure 10(b). Similar to a B+tree of order m, our structure satisfies the following invariants. (a) Every node has at most m children. (b) Every non-leaf node (except root) as at-least $\lceil \frac{m}{2} \rceil$ children. (c) All leaf nodes appear at the same level. Again similar to B+tree, we ensure the invariants by either splitting a node into two when the number of children overflow or merging two nodes into one when the number of children underflow. This ensures that the height of the tree is at most $\log_{\lceil m/2 \rceil} N$.

Hierarchical Positional Mapping: Fetch. Our hierarchical indexing structure makes accessing the item at the n^{th} position efficient, using the following steps: (i) We start from the root node. (ii) At a node, we identify the child node to traverse next, by subtracting the count associated with the children iteratively from n, left to right, as long as the remainder is positive. This step adjusts the value of n; we then move one level down in the tree to that child node. (iii) We repeat the previous step until we reach a leaf node, after which we extract the n^{th} element from this node. Now, we have the key with which to probe a traditional B+tree index on the positional mapping keys, as in monotonic positional mapping. Overall, the complexity of this operation is $\mathcal{O}(\log N)$.

Hierarchical Positional Mapping: Insert/Delete. Insert and delete operations require updating the counts associated with all of the nodes that fall on the path between the root and the leaf node corresponding to the position that is being updated. As before, we first identify the leaf node as discussed for a fetch operation, followed by updating the item at the leaf node, and traversing back up the tree to the root. Simultaneously, we use the traditional B+tree index on the positional mapping keys to update the corresponding positional mapping key. Once again, the complexity of this operation is $\mathcal{O}(\log N)$.

In Table 5, we contrast the complexity of the hierarchical positional mapping scheme against other positional mapping schemes, and demonstrate that it dominates the other schemes. We empirically evaluate our positional mapping schemes in Section 7.

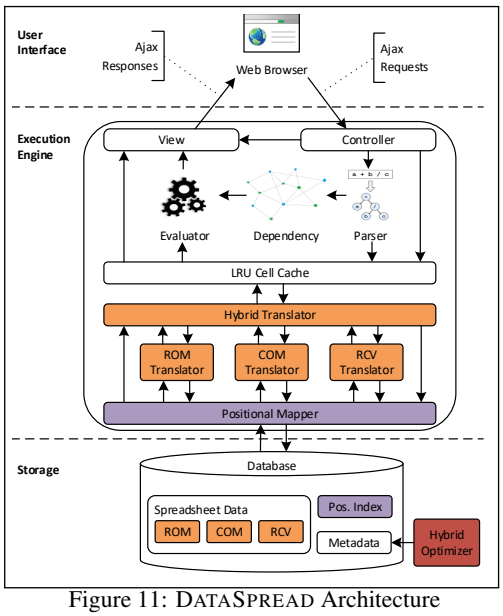


Figure 11: DATASPREAD Architecture

Positional Mapping Method	Operation on n^{th} record. Fetch	Insert/Delete
Row Number as-is	$\mathcal{O}(\log N)$	$\mathcal{O}(N)$
Monotonic Positional Mapping	$\mathcal{O}(N)$	$\mathcal{O}(\log N)$
Hierarchical Positional Mapping	$\mathcal{O}(\log N)$	$\mathcal{O}(\log N)$

Table 5: Complexity of different positional mapping methods.

6. DATASPREAD ARCHITECTURE

We have implemented DATASPREAD as a web-based tool on top of a PostgreSQL relational database implementing the Model-View-Controller approach. The system currently supports basic spreadsheet operations, *e.g.*, scrolling to arbitrary positions, insertion of rows or columns, and formulae insert and evaluation, on large spreadsheets that are persisted in the PostgreSQL database.

Figure 11 illustrates DATASPREAD's architecture, which at a high level can be divided into three main layers, *i.e.*, (a) user interface, (b) execution engine, and (c) storage. The *user interface layer* consists of a *spreadsheet widget*, which presents a spreadsheet on a web-based interface to users and records the interactions on it. The *execution engine layer* is a web application developed in Java that resides on an application server. The *controller* accepts user interactions in form of events and identifies the corresponding actions, *e.g.*, a formula update is sent to the formula parser, an update to a cell is sent to the cell cache. The *dependency graph* captures the formula dependencies between the cells and aids in triggering the computation of dependent cells. The *positional mapper* translates the row and column numbers into the corresponding stored identifiers and vice versa. The *ROM, COM, RCV, and hybrid translators* use their corresponding spreadsheet representations and provide a "collection of cells" abstraction to the upper layers. This collection of cells are then cached in memory via an *LRU cell cache*. The storage layer consists of a relational database, which is responsible for persisting data. This data is persisted using a combination of *ROM, COM and RCV data models* (as described in Section 4) along with *positional indexes*, which map row and column numbers to corresponding stored identifiers (as described in Section 5), and *metadata*, which records information about the hybrid data model, and which tables are responsible for handling which rectangular areas on the spreadsheet. The *hybrid optimizer* determines the optimal hybrid data model and is responsible for migrating data across different tables and primitive data models.

Relational Operations in Spreadsheet. Since DATASPREAD is built on top of a traditional relational database, it can leverage the SQL engine of the database and seamlessly support SQL queries on the front-end spreadsheet interface. We describe how we support standard relational operations in more detail in Appendix C.

7. EXPERIMENTAL EVALUATION

In this section, we present an evaluation of DATASPREAD. Our high-level goals are to evaluate the feasibility of DATASPREAD to work with large spreadsheets with billions of cells; in addition, we attempt to understand the impact of the hybrid data models, and the impact of the positional mapping schemes. Recent work has identified 500ms as a yardstick of interactivity [29], and we aim to verify if DATASPREAD can actually meet that yardstick.

7.1 Experimental Setup

Environment. Our data models and positional mapping techniques were implemented on top of a PostgreSQL (version: 9.6) database. The database was configured with default parameters. We run all of our experiments on a workstation with the following configuration: Processor: `Intel Core i7-4790K 4.0 GHz`, RAM: `16 GB`, Operating System: `Windows 10`. Our test scripts are single-threaded applications developed in `Java`. While we have also developed a full-fledged web-based front-end application (see Figure 4), our test scripts are independent of this front-end, so that we can isolate the back-end performance implications. We ensured fairness by clearing the appropriate cache(s) before every run.

Datasets. We evaluate our algorithms on a variety of real and synthetic datasets. Our real datasets are the ones listed in Table 1: Internet, ClueWeb09, Enron, and Academic. The first three have over 10,000 sheets each while the last one has about 700 sheets. To test scalability, our real-world datasets are insufficient, because they are limited in scale by what current spreadsheet tools can support. Therefore, we constructed additional large synthetic spreadsheet datasets. The spreadsheets in the datasets each have between 10–100 columns, with the number of rows varying from 10^3 to 10^7, and a density between 0–1; this last quantity indicates the probability that a given cell within the spreadsheet area is filled-in. Our largest synthetic dataset has a billion non-empty cells, enabling us to explicitly verify the premise of the title of this work.

We identify several goals for our experimental evaluation:

Goal 1: Impact of Hybrid Data Models on Real Datasets. We evaluate the hybrid data models selected by our algorithms against the primitive data models, when the cost model is optimized for storage. The algorithms evaluated include: *ROM, COM, RCV* (the primitive data models, using a single table to represent a sheet), *DP* (the dynamic programming algorithm from Section 4.4), and *Greedy* and *Agg* (the greedy and aggressive-greedy algorithms from Section 4.5). We evaluate these data models on both *storage*, as well as *formulae access cost*, based on the formulae embedded within the spreadsheets. In addition, we evaluate the *running time* of the hybrid optimization algorithms for *DP, Greedy, and Agg*.

Goal 2: Scalability on Synthetic Datasets. Since our real datasets aren't very large, we turn to synthetic datasets for testing out the scalability of DATASPREAD. We focus on the primitive data models, *i.e.*, *ROM* and *RCV*, coupled with positional mapping schemes, and evaluate the performance of *select, update, and insert/delete* on these data models on varying the *number of rows, number of columns, and the density* of the dataset.

Goal 3: Impact of Positional Mapping Schemes. We evaluate the impact of our positional mapping schemes in aiding positional access on the spreadsheet. We focus on *Row-number-as-is, Monotonic, and Hierarchical* positional mapping schemes applied on the ROM primitive model, and evaluate the performance of *fetch, insert, and delete* operations on varying the *number of rows*.

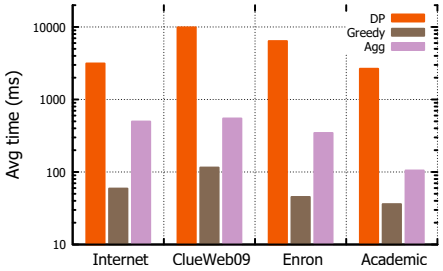


Figure 13: Hybrid optimization algorithms: Running time.

7.2 Impact of Hybrid Data Models

Takeaways: Hybrid data models provide substantial benefits over primitive data models, with up to ***20% reductions in storage, and up to 50% reduction in formula access or evaluation time*** *on PostgreSQL on real spreadsheet datasets, compared to the best primitive data model. While DP has better performance on storage than Greedy and Agg, it suffers from high running time;* ***Agg is able to bridge the gap between Greedy and DP****, while taking only marginally more running time than Greedy. Lastly, if we were to design a database storage engine from scratch, the hybrid data models would provide* ***up to 50% reductions*** *in storage compared to the best primitive data model.*

The goal of this section is to evaluate our data models—both our primitive and hybrid data models—on real datasets. For each sheet within each dataset, we run the dynamic programming algorithm (denoted DP), the greedy algorithm (denoted Greedy), and the aggressive greedy algorithm (denoted Agg) that help us identify effective hybrid data models. We compare the resulting data models against the primitive data models: ROM, COM and RCV, where the entire spreadsheet is stored in a single table.

Storage Evaluation on PostgreSQL. We begin with an evaluation of storage for different data models on PostgreSQL. The costs for storage on PostgreSQL as measured by us is as follows: s_1 is 8 KB, s_2 is 1 bit, s_3 is 40 bytes, s_4 is 50 bytes, and s_5 is 52 bytes. We plot the results in Figure 12(a): here, we depict the average normalized storage across sheets: for the Internet, ClueWeb09, and Enron datasets, we found RCV to have the worst performance, and hence normalized it to a cost of 100, and scaled the others accordingly; for the Academic datasets, we found COM to have the worst performance, and hence normalized it to a cost of 100, and scaled the others accordingly. For the first three datasets, recall that these datasets are primarily used for data sharing, and as a result are quite *dense*. As a result, the ROM and COM data models do well, using about 40% of the storage of RCV. At the same time, DP, Greedy and Agg perform roughly similarly, and better than the primitive data models, providing an additional reduction of 15-20%. On the other hand, the last dataset, which is primarily used for computation as opposed to sharing, and is very sparse, RCV does better than ROM and COM, while DP, Greedy, and Agg once again provide additional benefits.

Storage Evaluation on an Ideal Database. Note that the reason why RCV does so poorly for the first three datasets is because PostgreSQL imposes a high overhead per tuple, of 50 bytes, considerably larger than the amount of storage required to store each cell. So, to explore this further, we investigated the scenario if we had the ability to redesign our database storage engine from scratch. We consider a theoretical "ideal" cost model, where additional overheads are minimized. For this cost model, the cost of a ROM or COM table is equal to the number of cells, plus the length and

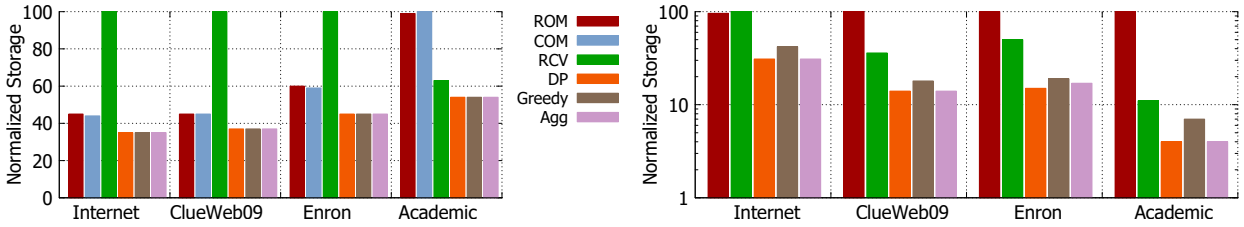


Figure 12: (a) Storage Comparison for PostgreSQL (b) Storage Comparison on an Ideal Database

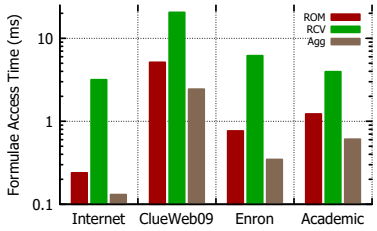


Figure 14: Average access time for formulae

breadth of the table (to store the data, the schema, as well as positional identifiers), while the cost of an RCV row is simply 3 units (to store the data, as well as the row and column number). We plot the results in Figure 12(b) in log scale for each of the datasets—we exclude COM for this chart since it has the same performance as ROM. Here, we find that ROM has the worst cost across most of the datasets since it no longer leverages benefits from minimizing the number of tuples. (For Internet, ROM and RCV are similar, but RCV is slightly worse.) As before, we normalize the cost of the ROM model to 100 for each sheet, and scaled the others accordingly, followed by taking an average across all sheets per dataset. As an example, we find that for the ClueWeb09 corpus, RCV, DP, Greedy and Agg have normalized costs of about 36, 14, 18, and 14 respectively—with the hybrid data models more than halving the cost of RCV, and getting $\frac{1}{7}^{th}$ the cost of ROM. Furthermore, in this ideal cost model, DP provides additional benefits relative to Greedy, and Agg ends up bringing us close to or equal to DP performance.

Running Time of Hybrid Optimization Algorithm. Our next question is how long our hybrid data model optimization algorithms for DP, Greedy, and Agg, take on real datasets. In Figure 13, we depict the average running time of these algorithms on the four real datasets. The results for all datasets are similar—as an example, for Enron, DP took 6.3s on average, Greedy took 45ms (a 140× reduction), while Agg took 345ms (a 20× reduction). Thus DP has the highest running time for all datasets, since it explores the entire space of models that can be obtained by recursive partitioning. Between Greedy and Agg, Greedy turns out to take less time. Note that these observations are consistent with our complexity analyses from Section 4.5. That said, Agg allows us to trade off a little bit more running time for improved performance on storage (as we saw earlier). We note that for the cases where the spreadsheets were large, we terminated DP after about 10 minutes, since we want our optimization to be relatively fast. (Note that using a similar criterion for termination, Agg and Greedy did not have to be terminated for any of the real datasets.) To be fair across all the algorithms, we excluded all of these spreadsheets from this chart—if we had included them, the difference between DP and the other algorithms would be even more stark.

Formulae Access Evaluation on PostgreSQL. Next, we wanted to evaluate if our hybrid data models, optimized only on storage, have any impact on the access cost for formulae within the real datasets. Our hope is that the formulae embedded within spreadsheets end up focusing on "tightly coupled" tabular areas, which our hybrid data models are able to capture and store in separate tables. For this evaluation, we focused on Agg, since it provided the best trade-off between running time and storage costs. Given a sheet in a dataset, for each data model, we measured the time taken to evaluate the formulae in that sheet, and averaged this time across all sheets and all formulae. We plot the results for different datasets in Figure 14 in log scale in ms. As a concrete example, on the Internet dataset, ROM has a formula access time of 0.23, RCV has 3.17, while Agg has 0.13. Thus, Agg provides a substantial reduction of 96% over RCV and 45% over ROM—even though Agg was optimized for storage and not for formula access. This validates our design of hybrid data models to store spreadsheet data. Note that while the performance numbers for the real spreadsheet datasets are small for all data models (due to the size limitations in present spreadsheet tools) when scaling up to large datasets, and formulae that operate on these large datasets, these numbers will increase in a proportional manner, at which point it is even more important to opt for hybrid data models.

7.3 Scalability of Data Models

Takeaway: Our primitive data models, augmented with positional mapping provide ***interactive (<500ms) response time on spreadsheet datasets ranging up to 1 billion cells*** *for select, insert, and update operations.*

Since our real datasets did not have any spreadsheets that are extremely large, we now evaluate the scalability of the DATASPREAD data models in supporting very large synthetic spreadsheets. We focus on the two primitive data models *i.e.*, ROM and RCV, with the spreadsheet being represented as a single table in these data models. Since we use synthetic datasets where cells are "filled in" with a certain probability, we did not involve hybrid data models, since they would (in this artificial context) typically end up preferring the ROM data model. These primitive data models are augmented with hierarchical positional mapping. We consider the performance on varying several parameters of these datasets: the density (*i.e.*, the number of cells that are filled in), the number of rows, and the number of columns. The default values of these parameters are 1, 10^7 and 100 respectively. We repeat each operation 500 times and report the averages.

In Figure 15, we depict the charts corresponding to average time to perform a random select operation on a region of 1000 rows and 20 columns. This is, for example, the operation that would correspond to a user scrolling to a certain position on our spreadsheet. As can be seen in Figure 15(a), ROM starts dominating RCV beyond a certain density, at which point it makes more sense to store the data in as tuples that span rows instead of incurring the penalty of creating a tuple for every cell. Nevertheless, the best of these two models takes less than 150ms across sheets of varying densities. In Figure 15(b)(c), since the spreadsheet is very dense (density = 1), ROM takes less time than RCV. Overall, in all cases, even on spreadsheets with 100 columns and 10^7 rows and a density of 1, the average time to select a region is well within 500ms.

We report briefly on the update and insert performance—detailed results and charts can be found in the Appendix. Overall, for both RCV and ROM, for inserting a row, the time is well below 500ms

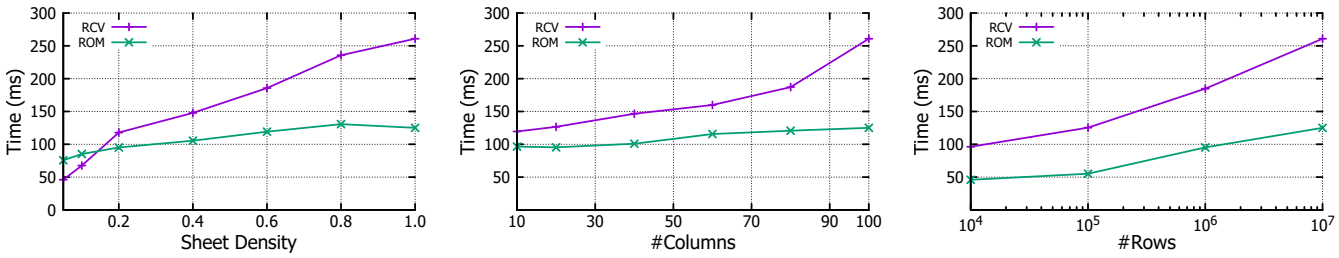


Figure 15: Select performance vs — (a) Sheet Density (b) Column Count (c) Row Count

for all of the charts; for updates of a large region, while ROM is still highly interactive, RCV ends up taking longer since 1000s of queries need to be issued to the database. In practice, users won't update such a large region at a time, and we can batch these queries. We discuss this further in the appendix.

7.4 Evaluation of Positional Mapping

Takeaway: Hierarchical positional mapping retains the rapid fetch benefits of row-number-as-is, while also providing the rapid insert and update benefits of monotonic positional mapping. Overall, ***hierarchical positional mapping is able to perform positional operations within a few milliseconds****, while the other positional mapping schemes scale poorly, taking seconds on large datasets for certain operations.*

We report detailed results and charts for this evaluation in Appendix D.

8. RELATED WORK

Our work draws on related work from multiple areas; we review papers in each of the areas, and describe how they relate to DATASPREAD. We discuss 1) efforts that enhance the usability of databases, 2) those that attempt to merge the functionality of the spreadsheet and database paradigms, but without a holistic integration, and 3) using array-based database management systems. We described our vision for DATASPREAD in an earlier demo paper [10].

1. Making databases more usable. There has been a lot of recent work on making database interfaces more user friendly [4, 23]. This includes recent work on gestural query and scrolling interfaces [22, 31, 33, 32, 36], visual query builders [6, 16], query sharing and recommendation tools [24, 18, 17, 25], schema-free databases [34], schema summarization [49], and visual analytics tools [14, 30, 37, 21]. However, none of these tools can replace spreadsheet software which has the ability to analyze, view, and modify data via a direct manipulation interface [35] and has a large user base.

2a. One way import of data from databases to spreadsheets. There are various mechanisms for importing data from databases to spreadsheets, and then analyzing this data within the spreadsheet. This approach is followed by Excel's Power BI tools, including Power Pivot [45], with Power Query [46] for exporting data from databases and the web or deriving additional columns and Power View [46] to create presentations; and Zoho [42] and ExcelDB [44] (on Excel), and Blockspring [43] (on Google Sheets [39]) enabling the import from a variety of sources including the databases and the web. Typically, the import is one-shot, with the data residing in the spreadsheet from that point on, negating the scalability benefits derived from the database. Indeed, Excel 2016 specifies a limit of 1M records that can be analyzed once imported, illustrating that the scalability benefits are lost; Zoho specifies a limit of 0.5M records. Furthermore, the connection to the base data is lost: any modifications made at either end are not propagated.

2b. One way export of operations from spreadsheets to databases. There has been some work on exporting spreadsheet operations into database systems, such as the work from Oracle [47, 48] as well as startups 1010Data [40] and AirTable [41], to improve the performance of spreadsheets. However, the database itself has no awareness of the existence of the spreadsheet, making the integration superficial. In particular, positional and ordering aspects are not captured, and user operations on the front-end, e.g., inserts, deletes, and adding formulae, are not supported.

2c. Using a spreadsheet to mimic a database. There has been some work on using a spreadsheet as an interface for posing traditional database queries. For example, Tyszkiewicz [38] describes how to simulate database operations in a spreadsheet. However, this approach loses the scalability benefits of relational databases. Bakke et al. [9, 8, 7] support joins by depicting relations using a nested relational model. Liu et al. [28] use spreadsheet operations to specify single-block SQL queries; this effort is essentially a replacement for visual query builders. Recently, Google Sheets [39] has provided the ability to use single-table SQL on its frontend, without availing of the scalability benefits of database integration. Excel, with its Power Pivot and Power Query [46] functionality has made moves towards supporting SQL in the front-end, with the same limitations. Like this line of work, we support SQL queries on the spreadsheet frontend, but our focus is on representing and operating on spreadsheet data within a database.

3. Array database systems. While there has been work on array-based databases, most of these systems do not support edits: for instance, SciDB [13] supports an append-only, no-overwrite data model.

9. CONCLUSIONS

We presented DATASPREAD, a data exploration tool that holistically unifies spreadsheets and databases with a goal towards working with large datasets. We proposed three primitive data models for representing spreadsheet data within a database, along with algorithms for identifying the optimal hybrid data model arising from recursive decomposition to give one or more primitive data models. Our hybrid data models provide substantial reductions in terms of storage (up to 20–50%) and formula evaluation (up to 50%) over the primitive data models. Our primitive and hybrid data models, coupled with positional mapping schemes, make working with very large spreadsheets—over a billion cells—interactive.

10. REFERENCES

[1] Google sheets. https://www.google.com/sheets/about/.
[2] Microsoft excel. http://products.office.com/en-us/excel.
[3] ZK Spreadsheet. https://www.zkoss.org/product/zkspreadsheet.
[4] S. Abiteboul, R. Agrawal, P. Bernstein, M. Carey, S. Ceri, B. Croft, D. DeWitt, M. Franklin, H. G. Molina, D. Gawlick, J. Gray, L. Haas, A. Halevy, J. Hellerstein, Y. Ioannidis, M. Kersten, M. Pazzani, M. Lesk, D. Maier, J. Naughton,

H. Schek, T. Sellis, A. Silberschatz, M. Stonebraker, R. Snodgrass, J. Ullman, G. Weikum, J. Widom, and S. Zdonik. The lowell database research self-assessment. *Commun. ACM*, 48(5):111–118, May 2005.

[5] A. Abouzied, J. Hellerstein, and A. Silberschatz. Dataplay: Interactive tweaking and example-driven correction of graphical database queries. In *Proceedings of the 25th Annual ACM Symposium on User Interface Software and Technology*, UIST '12, pages 207–218, New York, NY, USA, 2012. ACM.

[6] A. Abouzied, J. Hellerstein, and A. Silberschatz. DataPlay: interactive tweaking and example-driven correction of graphical database queries. In *Proceedings of the 25th annual ACM symposium on User interface software and technology*, pages 207–218. ACM, 2012.

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

[8] E. Bakke, D. Karger, and R. Miller. A spreadsheet-based user interface for managing plural relationships in structured data. In *Proceedings of the SIGCHI conference on human factors in computing systems*, pages 2541–2550. ACM, 2011.

[9] E. Bakke and D. R. Karger. Expressive query construction through direct manipulation of nested relational results. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1377–1392. ACM, 2016.

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

[11] M. Bendre, B. Sun, X. Zhou, D. Zhang, K. Chang, and A. Parameswaran. Dataspread: Unifying databases and spreadsheets. In *VLDB*, volume 8, 2015.

[12] D. Bricklin and B. Frankston. Visicalc 1979. *Creative Computing*, 10(11):122, 1984.

[13] P. G. Brown. Overview of scidb: Large scale array storage, processing and analysis. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 963–968, New York, NY, USA, 2010. ACM.

[14] S. P. Callahan, J. Freire, E. Santos, C. E. Scheidegger, C. T. Silva, and H. T. Vo. VisTrails: visualization meets data management. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 745–747. ACM, 2006.

[15] J. Callan, M. Hoy, C. Yoo, and L. Zhao. Clueweb09 data set, 2009.

[16] T. Catarci, M. F. Costabile, S. Levialdi, and C. Batini. Visual query systems for databases: A survey. *Journal of Visual Languages & Computing*, 8(2):215–260, 1997.

[17] U. Cetintemel, M. Cherniack, J. DeBrabant, Y. Diao, K. Dimitriadou, A. Kalinin, O. Papaemmanouil, and S. B. Zdonik. Query Steering for Interactive Data Exploration. In *CIDR*, 2013.

[18] G. Chatzopoulou, M. Eirinaki, and N. Polyzotis. Query recommendations for interactive database exploration. In *Scientific and Statistical Database Management*, pages 3–18. Springer, 2009.

[19] C. E. L. Cormen, Thomas H. and R. L. Rivest. *Introduction to Algorithms. Cambridge*. MA: MIT, 1990.

[20] D. Flax. Gesturedb: An accessible & touch-guided ipad app for mysql database browsing. 2016.

[21] H. Gonzalez, A. Y. Halevy, C. S. Jensen, A. Langen, J. Madhavan, R. Shapley, W. Shen, and J. Goldberg-Kidon. Google fusion tables: web-centered data management and collaboration. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 1061–1066. ACM, 2010.

[22] S. Idreos and E. Liarou. dbTouch: Analytics at your Fingertips. In *CIDR*, 2013.

[23] H. V. Jagadish, A. Chapman, A. Elkiss, M. Jayapandian, Y. Li, A. Nandi, and C. Yu. Making database systems usable. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 13–24. ACM, 2007.

[24] N. Khoussainova, M. Balazinska, W. Gatterbauer, Y. Kwon, and D. Suciu. A Case for A Collaborative Query Management System. In *CIDR*. www.cidrdb.org, 2009.

[25] N. Khoussainova, Y. Kwon, M. Balazinska, and D. Suciu. SnipSuggest: Context-aware autocompletion for SQL. *Proceedings of the VLDB Endowment*, 4(1):22–33, 2010.

[26] B. Klimt and Y. Yang. Introducing the enron corpus. In *CEAS*, 2004.

[27] A. Lingas, R. Y. Pinter, R. L. Rivest, and A. Shamir. Minimum edge length partitioning of rectilinear polygons. In *Proceedings - Annual Allerton Conference on Communication, Control, and Computing*, pages 53–63, 1982.

[28] B. Liu and H. V. Jagadish. A Spreadsheet Algebra for a Direct Data Manipulation Query Interface. pages 417–428. IEEE, Mar. 2009.

[29] Z. Liu and J. Heer. The effects of interactive latency on exploratory visual analysis. *IEEE Trans. Vis. Comput. Graph.*, 20(12):2122–2131, 2014.

[30] J. Mackinlay, P. Hanrahan, and C. Stolte. Show me: Automatic presentation for visual analysis. *Visualization and Computer Graphics, IEEE Transactions on*, 13(6):1137–1144, 2007.

[31] A. N. L. J. M. Mandel, A. Nandi, and L. Jiang. Gestural Query Specification. *Proceedings of the VLDB Endowment*, 7(4), 2013.

[32] A. Nandi. Querying Without Keyboards. In *CIDR*, 2013.

[33] A. Nandi and H. V. Jagadish. Guided interaction: Rethinking the query-result paradigm. *Proceedings of the VLDB Endowment*, 4(12):1466–1469, 2011.

[34] L. Qian, K. LeFevre, and H. V. Jagadish. CRIUS: user-friendly database design. *Proceedings of the VLDB Endowment*, 4(2):81–92, 2010.

[35] B. Shneiderman. Direct Manipulation: A Step Beyond Programming Languages. *IEEE Computer*, 16(8):57–69, 1983.

[36] M. Singh, A. Nandi, and H. V. Jagadish. Skimmer: rapid scrolling of relational query results. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 181–192. ACM, 2012.

[37] C. Stolte, D. Tang, and P. Hanrahan. Polaris: A system for query, analysis, and visualization of multidimensional relational databases. *Visualization and Computer Graphics, IEEE Transactions on*, 8(1):52–65, 2002.

[38] J. Tyszkiewicz. Spreadsheet as a relational database engine. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 195–206. ACM, 2010.

[39] http:/google.com/sheets. Google Sheets (retrieved March 10, 2015).

[40] https://www.1010data.com/. 1010 Data (retrieved March 10,

2015).
[41] https://www.airtable.com/. Airtable (retrieved March 10, 2015).
[42] https://www.zoho.com/. Zoho Reports (retrieved March 10, 2015).
[43] http://www.blockspring.com/. Blockspring (retrieved March 10, 2015).
[44] http://www.excel-db.net/. Excel-DB (retrieved March 10, 2015).
[45] http://www.microsoft.com/en-us/download/details.aspx?id=43348. Microsoft sql server power pivot (retrieved march 10, 2015).
[46] C. Webb. *Power Query for Power BI and Excel*. Apress, 2014.
[47] A. Witkowski, S. Bellamkonda, T. Bozkaya, N. Folkert, A. Gupta, J. Haydu, L. Sheng, and S. Subramanian. Advanced SQL modeling in RDBMS. *ACM Transactions on Database Systems (TODS)*, 30(1):83–121, 2005.
[48] A. Witkowski, S. Bellamkonda, T. Bozkaya, A. Naimat, L. Sheng, S. Subramanian, and A. Waingold. Query by excel. In *Proceedings of the 31st international conference on Very large data bases*, pages 1204–1215. VLDB Endowment, 2005.
[49] C. Yu and H. V. Jagadish. Schema summarization. In *Proceedings of the 32nd international conference on Very large data bases*, pages 319–330. VLDB Endowment, 2006.

APPENDIX

A. OPTIMAL HYBRID DATA MODELS

In this section, we demonstrate that the following problem is NP-HARD.

PROBLEM 2 (HYBRID-ROM). *Given a spreadsheet with a collection of cells C, identify the hybrid data model T with only ROM tables that minimizes* $\text{cost}(T)$.

As before, the cost model is defined as:

$$\text{cost}(T) = \sum_{i=1}^{p} s_1 + s_2 \cdot (r_i \times c_i) + s_3 \cdot c_i + s_4 \cdot r_i. \quad (7)$$

The decision version of the above problem has the following structure: a value k is provided, and the goal is to test whether there is a hybrid data model with $\text{cost}(T) \leq k$.

We reduce the minimum edge length partitioning problem [27] of rectilinear polygons to Problem 2, thereby showing that it is NP-Hard. First, a rectilinear polygon is a polygon in which all edges are either aligned with the x-axis or the y- axis. We consider the problem of partitioning a rectilinear polygon into disjoint rectangles using the minimum amount of "ink". In other words, the minimality criterion is the total length of the edges (lines) used to form the internal partition. Notice that this doesn't correspond to the minimality criterion of reducing the number of components. We illustrate this in Figure 19, which is borrowed from the original paper [27]. The following decision problem was shown to be NP-Hard in [27]: Given any rectilinear polygon P and a number k, is there a rectangular partitioning whose edge length does not exceed k? We now provide the reduction.

PROOF FOR PROBLEM 2. Consider an instance of the polygon partitioning problem with minimum edge length required to be at most k. We are given a rectilinear polygon P. We now represent the polygon P in a spreadsheet by filling the cells interior of the polygon, and not filling any other cell in the spreadsheet. Let $C = \{C_1, C_2, \ldots, C_m\}$ represent the set of all filled cells in the spreadsheet. We claim that a minimum edge length partition of the given rectilinear polygon P of length at most k exists iff the following setting of the optimal hybrid data model problem: $s_1 = 0, s_2 = 2|C|+1, s_3 = s_4 = 1$, where the storage cost should not exceed $k' = k + \frac{\text{Perimeter}(P)}{2} + s_2|C|$ for some decomposition of the spreadsheet.

$\Rightarrow$ Let us assume that the spreadsheet we generate using P has a decomposition of rectangles whose storage cost is less than $k' = k + \frac{\text{Perimeter}(P)}{2} + s_2|C|$. We have to show that there exists a partition with minimum edge length of at most k. We first make the following key observations:

1. There exists a valid decomposition that doesn't store any blank cell. Let's assume the contrary and consider a decomposition that stores a blank cell. Since we are now storing $|C|+1$ cells at minimum,

$$k' > s_2(|C|+1) = |C|s_2 + s_2 = |C|s_2 + 2|C| + 1$$
$$k' > \underbrace{|C|(s_2 + 1 + 1)}_{\text{storing each cell in a separate table}}$$

   Therefore, if we have a decomposition that stores a blank cell, we also have a decomposition that does not store any blank cell and has lower cost.
2. There exists a decomposition of the spreadsheet where all the tables are disjoint. The argument is similar to the previous case since storing the same cell twice in different tables is equivalent to storing an extra blank cell.

From our above two observations, we conclude that there exists a decomposition where all tables are disjoint, and no table stores a blank cell. Therefore, this decomposition corresponds to *partitioning* the given spreadsheet into rectangles. We represent this partition of the spreadsheet by $T = \{T_1, T_2, \ldots, T_p\}$. We now show that this partition of the spreadsheet corresponds to a partitioning of the rectilinear polygon P with edge-length less than k.

$$\begin{aligned}\text{cost}(T) &= \sum_{i=1}^{p} s_1 + s_2 \cdot (r_i \times c_i) + s_3 \cdot c_i + s_4 \cdot r_i \\ &= \sum_{i=1}^{p} s_1 + s_2 \sum_{i=1}^{p} \cdot (r_i \times c_i) + s_3 \sum_{i=1}^{p} c_i + s_4 \sum_{i=1}^{p} r_i\end{aligned}$$

substituting $s_1 = 0, s_2 = 2|C| + 1, s_3 = s_4 = 1$, we get:

$$= \sum_{i=1}^{p} 0 + s_2|C| + 1 \cdot \left(\sum_{i=1}^{p} c_i + \sum_{i=1}^{p} r_i \right)$$

since $\text{cost}(T) \leq k' = k + \frac{\text{Perimeter}(P)}{2} + s_2|C|$,

$$\begin{aligned}\text{cost}(T) = s_2|C| + 1 \cdot \left(\sum_{i=1}^{p} c_i + \sum_{i=1}^{p} r_i \right) \\ \implies \sum_{i=1}^{p} (r_i + c_i) \leq k + \frac{\text{Perimeter}(P)}{2} \\ \implies \sum_{i=1}^{p} \frac{\text{Perimeter}(T_i)}{2} \leq k + \frac{\text{Perimeter}(P)}{2} \\ \implies \sum_{i=1}^{p} \text{Perimeter}(T_i) \leq 2 \times k + \text{Perimeter}(P)\end{aligned}$$

Since, the sum of perimeters of all the tables T_i counts the boundary of P exactly once, and the edge length partition of P exactly

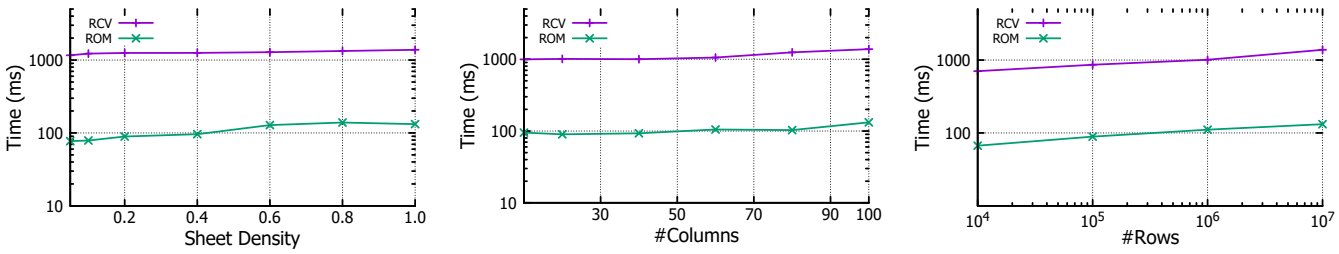


Figure 16: Update range performance vs (a) Sheet Density (b) Column Count (c) Row Count

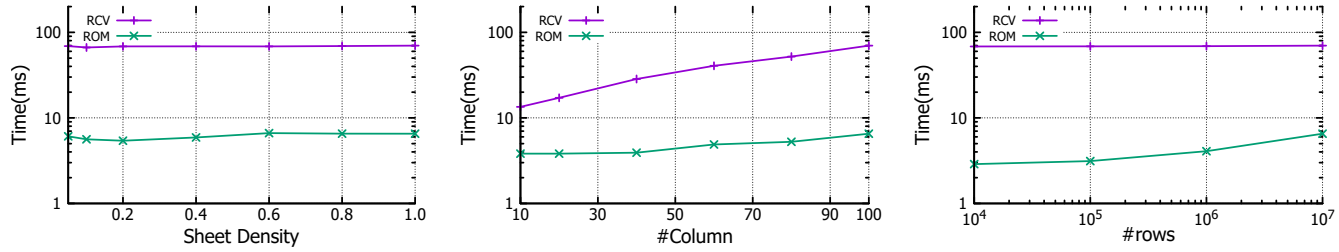


Figure 17: Insert row performance vs (a) Sheet Density (b) Column Count (c) Row Count

twice, the partition of the spreadsheet $T = \{T_1, T_2, \ldots, T_p\}$ corresponds to an edge-length partitioning of the given rectilinear polygon P with edge-lengh less than k.

$\Leftarrow$ Let us assume that the given rectilinear polygon P has a minimum edge length partition of length at most k. We have to show that there exists a decomposition of the spreadsheet whose storage cost is at most $k' = k + \frac{\text{Perimeter}(P)}{2} + s_2|C|$. Let us represent the set of rectangles that corresponds to an edge length partition of P of at most k as $T = \{T_1, T_2, \ldots, T_p\}$. We shall use the partition T of P as the decomposition of the spreadsheet itself:

$$\begin{aligned}\text{cost}(T) &= \sum_{i=1}^{p} s_1 + s_2 \cdot (r_i \times c_i) + s_3 \cdot c_i + s_4 \cdot r_i \\ &= \sum_{i=1}^{p} s_1 + s_2 \sum_{i=1}^{p} \cdot (r_i \times c_i) + s_3 \sum_{i=1}^{p} c_i + s_4 \sum_{i=1}^{p} r_i\end{aligned}$$

substituting $s_1 = 0, s_2 = 2|C| + 1, s_3 = s_4 = 1$, we get:

$$\begin{aligned}&= \sum_{i=1}^{p} 0 + s_2|C| + 1 \cdot \left(\sum_{i=1}^{p} c_i + \sum_{i=1}^{p} r_i \right) \\ &= s_2|C| + \sum_{i=1}^{p} (r_i + c_i) = s_2|C| + \sum_{i=1}^{p} \frac{\text{Perimeter}(T_i)}{2}\end{aligned}$$

since $\sum_{i=1}^{p} \text{Perimeter}(T_i) = 2 \times k + \text{Perimeter}(P)$, we have:

$$\begin{aligned}&\text{cost}(T) = s_2|C| + k + \frac{\text{Perimeter}(P)}{2} = k' \\ \implies &\text{cost}(T) = k'\end{aligned}$$

Therefore, the decomposition of the spreadsheet using T corresponds to a decomposition whose storage cost equals k'. Note that our reduction can be done in polynomial time. Therefore we can solve the minimum length partitioning problem in polynomial time if we have a polynomial time solution to the optimal storage problem. However, since it is shown in [27] that the minimum length partitioning problem is NP-Hard, the optimal hybrid data model problem is NP-Hard. This completes our proof. □

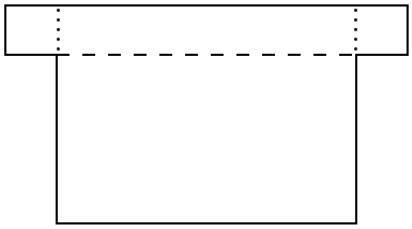


Figure 19: Minimum number of rectangles (– – –) does not coincide with minimum edge length (· · ·)

B. HYBRID DATA MODEL: EXTENSIONS

In this section, we discuss a number of extensions to the cost model of the hybrid data model. We will describe these extensions to the cost model, and then describe the changes to the basic dynamic programming algorithm; modifications to the greedy and aggressive greedy decomposition algorithms are straightforward.

Access Cost. So far, within our cost model, we have only been focusing on storage. As it turns out, our cost model can be extended in a straightforward manner to handle access cost — both scrolling-based operations, and formulae, and our dynamic programming algorithms can similarly be extended to handle access cost without any substantial changes. We focus on formulae since they are often the more substantial cost of the two; scrolling-based operations can be similarly handled. For formulae, there are multiple aspects that contribute to the time for access: the number of tables accessed, and within each table, since data is retrieved at a tuple level, the number of tuples that need to be accessed, and the size of these tuples. Once again, each of these aspects can be captured within the cost model via constants similar to $s_1, \ldots, s_5$, and can be seamlessly incorporated into the dynamic programming algorithm. Thus, we have:

THEOREM 5 (OPTIMALITY WITH ACCESS COST). *The optimal ROM, COM, and RCV-based hybrid data model based on recursive decomposition, across both storage and access cost, can be determined via dynamic programming.*

Size Limitations of Present Databases. Current databases impose limitations on the number of columns within a relation[1]; since spreadsheets often have an arbitrarily large number of rows and

[1]Oracle column number limitations: https://docs.oracle.com/cd/B19306_01/server.

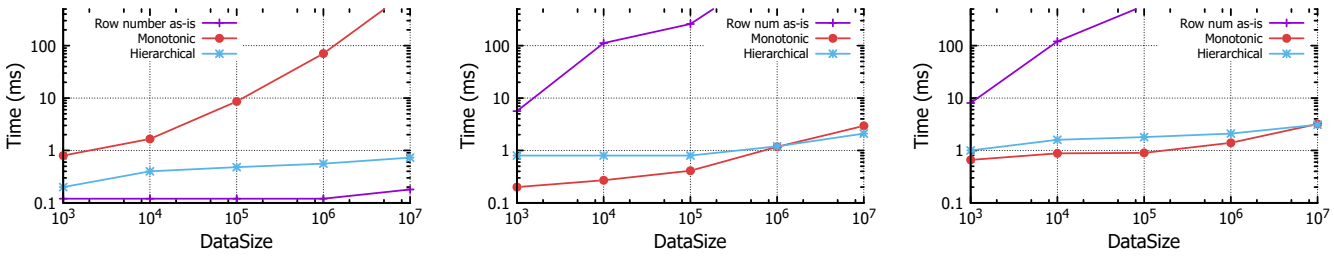


Figure 18: Positional mapping performance for (a) Select (b) Insert (c) Delete

columns (sometimes 10s of thousands each), we need to be careful when trying to capture a spreadsheet area within a collection of tables that are represented in a database.

This is relatively straightforward to capture in our context: in the case where we don't split (Equation 3), if the number of columns is too large to be acceptable, we simply return ∞ as the cost.

THEOREM 6 (OPTIMALITY WITH SIZE CONSTRAINTS). *The storage optimal ROM, COM, and RCV-based hybrid data model, with the constraint that no tables violate size constraints, based on recursive decomposition, can be determined via dynamic programming.*

Incorporating the Costs of Indexes. Within our cost model, it is straightforward to incorporate the costs associated with storage of indexes, since the size of the indexes are typically proportional to the number of tuples for a given table, and the cost of instantiating an index is another fixed constant cost. Since our cost model is general, by suitably reweighting one or more of s_1, s_2, s_3, s_4, we can capture this aspect within our cost model, and apply the same dynamic programming algorithm.

THEOREM 7 (OPTIMALITY WITH INDEXES). *The storage optimal ROM-based hybrid data model, with the costs of indexes included, based on recursive decomposition, can be determined via dynamic programming.*

C. RELATIONAL OPERATIONS SUPPORT

In addition to standard spreadsheet operations, DATASPREAD benefits from being built on a standard relational database, and as a result, seamlessly supports standard relational operations as well. To support relational operations from the spreadsheet interface, and in particular to enable table declaration and query execution, we introduce two functions in our system, namely DBTable and DBSQL.

DBTable enables a user to declare a portion of the spreadsheet front-end as a database table. Here, the displayed table cells reflect the content of the database table. This is a cue for the hybrid optimizer to "force" this region to be stored as a separate ROM table. Note that there is a two-way synchronization for such a table, *i.e.*, any updates to the table from the front-end is reflected at the back-end and vice versa.

DBSQL enables a user to execute arbitrary SQL queries combining data present on the spreadsheet, and other tables present in the relational database. To support positional addressing or referencing of spreadsheet data using DBSQL, we introduce two functions: RangeValue and RangeTable. RangeValue allows a user to refer a scalar value contained in a cell, *e.g.*, SELECT FROM Actors WHERE ActorId = RangeValue(A1); here, RangeValue(A1) refers to the value of cell A1. RangeTable allows a user to refer to a range, and perform operations on this range like a database table. This enables any range on a spreadsheet to be treated as a table, *e.g.*, SELECT FROM Actors NATURAL JOIN RangeTable(A1:D100).

102/b14237/limits003.htm#i288032; MySQL column limitations: https://dev.mysql.com/doc/mysql-reslimits-excerpt/5.5/en/column-count-limit.html; PostgreSQL column limitations: https://www.postgresql.org/about/

D. ADDITIONAL EXPERIMENTS

D.1 Scalability of Inserts and Deletes

We now supplement our evaluation of the scalability of selects in the main body of the paper with an evaluation of the scalability of inserts and updates for the primitive data models on a synthetic dataset. Figures 16 and 17 depict the corresponding charts for updating a region of 100 rows and 20 columns, and inserting one row of 100 columns for the primitive data models. In Figures 16, we find that the update time taken for RCV is a lot higher than the time for inserts or selects. This is because in this benchmark, DATASPREAD assumes that the entire region update happens at once, and fires $100 \times 20 = 2000$ update queries one at a time to the underlying database, to update each individual cell. In practice, users may only update a small number of cells at a time; and further, we may be able to batch these queries or issue them in parallel to further save time. In Figures 17, we find that like in Figures 16, the time taken for updates on ROM is faster than RCV since it only needs to issue one query, while RCV needs to issue multiple queries. However, in this case, since the number of queries issued is small, the response time is always within 100ms.

D.2 Impact of Positional Mapping

We now compare the performance of our different positional mapping methods as described in Section 5. Specifically, we contrast between (i) storing row-number-as is (denoted row-number-as-is), (ii) monotonic positional mapping (denoted monotonic), and (iii) hierarchical positional mapping (denoted hierarchical). As described previously, we operate on a dense dataset ranging from 10^3 to 10^7 rows, with 100 columns, all of whose cells are filled. The evaluation was performed on a single ROM table that captures all of the data on the sheet; evaluations for other primitive data models are similar. Figure 18 displays the average time taken to perform a fetch, insert, and delete of a single (random) row, averaged across 1000 iterations.

We see that the storing the row number as-is performs well for the fetch operation. However, the time for insert and delete operations increases rapidly with the data size, due to cascading updates of subsequent rows; thus, beyond a data size of 10^5, row number-as-is is no longer interactive ($> 500ms$) for insert and delete. On the other hand, the response time of the monotonic positional mapping for fetch operation increases rapidly with data size. This is again expected, as we need to search linearly through the positional mapping keys to retrieve the required records—making it infeasible to use on large datasets. Lastly, we find that hierarchical positional mapping performs well *for all operations* and performance does not get degrade even with data sizes of 10^9 tuples. In comparison with the other schemes, hierarchical positional mapping performs all the three aforementioned operations in few milliseconds, which makes it the practical choice for positional mapping for DATASPREAD.