Benchmarking Spreadsheet Systems (Technical Report)

Sajjadur Rahman
University of Illinois (UIUC)
srahman7@illinois.edu

Kelly Mack*
University of Washington
kmack3@uw.edu

Mangesh Bendre*
VISA Research
mbendre@visa.com

Ruilin Zhang*
University of Southern California
rzhang74@usc.edu

Richard Lin
University of California, Berkeley
richard.lin.047@berkeley.edu

Karrie Karahalios
University of Illinois (UIUC)
kkarahal@illinois.edu

Aditya Parameswaran*
University of California, Berkeley
adityagp@berkeley.edu

ABSTRACT

Spreadsheet systems are used for storing and analyzing data across domains by programmers and non-programmers alike. While spreadsheet systems have continued to support increasingly large datasets, they are prone to hanging and freezing while performing computations even on much smaller ones. We present a benchmarking study that evaluates and compares the performance of three popular systems, Microsoft Excel, LibreOffice Calc, and Google Sheets, on a range of canonical spreadsheet computation operations. We find that spreadsheet systems lack interactivity for several operations, on datasets well below their advertised scalability limits. We further evaluate whether spreadsheet systems adopt database optimization techniques such as indexing, intelligent data layout, and incremental and shared computation, to efficiently execute computation operations. We outline several ways future spreadsheet systems can be redesigned to offer interactive response times on large datasets.

KEYWORDS

Spreadsheet systems; Scalability; Use cases

1 INTRODUCTION

Spreadsheets are everywhere—we use them for managing our class grades, our daily food habits, scientific experiments, real-estate developments, financial portfolios, and even fantasy football scores [26]. Recent estimates from Microsoft peg spreadsheet use at about $\frac{1}{10}^{\text{th}}$ of the world's population. Responding to user demands, spreadsheet systems now advertise support for increasingly large datasets. For example, Microsoft Excel supports more than 10s of billions of cells within a spreadsheet [30]. Even web-based Google Sheets now supports five million cells [17], a 12.5X increase from its previous limit of 400K cells. With increasing data sizes, however, spreadsheets have started to break down to the point of being unusable, displaying a number of scalability problems. They often freeze during computation, and are unable to import datasets well below their advertised size limits. Anecdotes from a recent paper report that computation on spreadsheets with as few as 20,000 rows can lead to hanging and freezing [26]. Further, importing a spreadsheet of 100,000 rows in Excel (10% of the size limit of one million rows) can take over 10 minutes [7].

These anecdotes beg the following questions: *How are spreadsheets actually implemented? For what sorts of operations and workloads do they return responses in interactive time-scales? When do they exhibit delays, become non-responsive, or crash? How do they perform when data and operations scale up? Do they employ "database-style" optimizations to support large datasets, such as query planning and optimization, indexing, or materialization?* These are important questions, since answering these questions can help make spreadsheet systems more *usable*, on large and complex datasets that are increasingly the norm. Unfortunately, it is hard for us to compare the internals of popular spreadsheet systems such as Microsoft Excel and Google Sheets, since they are closed-source. Online documentation about these systems is restricted to help manuals as opposed to architectural details. Our best proxy for understanding how spreadsheet systems work is to use a familiar and time-tested approach from databases: *benchmarking*. Benchmarking has been the cornerstone of database systems research, allowing us to measure progress on several problems, e.g., transaction processing [21], data analysis [36], and cloud computing [10].

In this paper, we present, to the best of our knowledge, *the first benchmarking study of spreadsheet systems*. We study the following popular spreadsheet systems: Microsoft Excel (Excel hereafter), Google Sheets, and LibreOffice Calc (Calc hereafter). Excel is a closed-source desktop spreadsheet system; Google Sheets is a web-based collaborative spreadsheet

*This work began when these authors were part of the University of Illinois.

system; and Calc is a open-source desktop spreadsheet system. These systems were selected to provide a diversity in terms of maturity (Excel is more mature), platform (desktop vs. web-based), and openness (open vs. closed source).

We construct two different kinds of benchmarks to evaluate these spreadsheet systems: *basic complexity testing (BCT)*, and *optimization opportunities testing (OOT)*. We have released the source code for both of these benchmarks[1].

Basic Complexity Testing (BCT). The BCT benchmark aims to assess the performance of basic operations on spreadsheets. We construct a taxonomy of operations—encapsulating opening, structuring, editing, and analyzing data—based on their expected time complexity, and evaluate the relative performance of the spreadsheet systems on a range of data sizes. Our goal is to understand the impact of the type of operation, the size of data being operated on, and the spreadsheet system used, on the latency. Moreover, we want to quantify when each spreadsheet system fails to be interactive for a given operation, violating the 500ms mark widely regarded as the bound for interactivity [25].

Optimization Opportunities Testing (OOT). Spreadsheet systems have continued to increase their size limits over the past few decades [17, 30]. On the other hand, research on data management has, over the past four decades, identified a wealth of techniques for optimizing the processing of large datasets. We wanted to understand whether spreadsheet systems take advantage of techniques such as indexes, incremental updates, workload-aware data layout, and sharing of computation. The OOT benchmark constructs specific scenarios to explore whether such optimizations are deployed by existing spreadsheet systems while performing spreadsheet formula computation. Our goal is to identify new opportunities for improving the design of spreadsheet systems to support computation on large datasets.

Benchmark Construction. Constructing these benchmarks and performing the evaluation was not straightforward. There were three primary challenges we had to overcome: interaction effects, implementation, and coverage.

1. Interaction effects. Unlike typical database benchmarking settings where there is a clear separation between the datasets and the queries, here the datasets and queries are mixed, since the computation is embedded on the spreadsheet as formulae alongside the data. Thus, there are interaction effects—any change on the spreadsheet, in addition to triggering the computation of the operation (or formula) being benchmarked, may also trigger the recomputation of other embedded formulae. To isolate the impact of embedded formulae, we operate on real-world datasets containing both formulae and raw data, as well as datasets with raw data only.

2. Implementation. Making a change to or performing an operation on the spreadsheet and measuring the time manually does not provide high accuracy times. Instead, we had to programmatically make changes to the sheet and measure the corresponding time(s). Unfortunately, all three systems: Excel, Google Sheets, and Calc, embed slightly different programming (macro) languages for this purpose, requiring an implementation from scratch for each system, for each operation. For Calc, the documentation for this language is minimal, requiring us to look at online forums for assistance. Additional challenges emerged with Google Sheets, since the variance in response times for certain operations was very high—possibly due to the variable load on the server where the operation is being performed.

3. Coverage. Spreadsheet systems support a wide variety of operations—*e.g.*, over 400 operations according to this source [29]—making it difficult to evaluate each operation individually. Instead, we classified the operations into several categories based on their expected complexity, type of inputs, and generated outputs, helping us perform targeted evaluation for the BCT benchmark. For the OOT benchmark, on the other hand, we relied on our creativity in identifying settings where "database-style" optimizations may be relevant. We targeted a number of settings related to formula execution, including accelerating the execution of a single formula at-a-time via indexing, incremental view updates, and intelligent data layouts, as well as that of multiple formulae, via pruning of redundant computation, and sharing of partial results.

Takeaways. Here are some interesting takeaways from our evaluation:

A. Spreadsheets are not interactive for many standard operations, even for as few as $50k$ *rows.* Spreadsheet systems often fail to return responses in interactive time-scales (*i.e.*, 500ms) for datasets well below their documented scalability limits; see Table 2 that depicts when each system becomes non-interactive for a given operation in our benchmark (described later). For example, both the desktop-based spreadsheets and Google Sheets allow importing of datasets with one million rows and five million cells, respectively. However, all three spreadsheet systems, *i.e.*, Excel, Calc, and Google Sheets, require more than 500ms to sort a spreadsheet with 10k, 6k, and 10k rows, respectively. Even when computing a simple aggregate operation like COUNTIF, Calc and Google Sheets violate the interactivity bound on a spreadsheet with 110k, and 10k rows, respectively. While Excel outperforms the other two spreadsheet systems for operations like aggregate, look up, and conditional formatting, there are other operations for which Calc (filter, pivot table), and Google Sheets (sort) have lowest latency on large datasets.

B. Spreadsheet systems, for the most part, do not employ any database-style optimizations. Apart from a lookup operation on sorted data in Excel, our benchmarking experiments do not reveal any evidence of spreadsheet systems adopting

[1] https://github.com/dataspread/spreadsheet-benchmark

relational database-style optimizations. Some egregious examples include the fact that (1) recomputing a formula due to a single cell update (an $O(1)$ operation if incremental view update is used), requires the same time as computing the formula from scratch; (2) n repeated instances of the exact same formula take $O(n)$ time instead of the formula being computed once and the results being reused; (3) "finding" a nonexistent value (*e.g.*, via find-and-replace) takes $O(n)$ time where n is the size of the data, despite the fact that inverted indexing of tokens can make it near-constant time.
We believe our evaluation can benefit spreadsheet system development in the future for supporting interactivity on larger datasets, and also provide a starting point for database researchers to contribute to the emergent discipline of spreadsheet computation optimization.

The rest of the paper is organized as follows: in Section 2, we provide an overview of the three spreadsheet systems being benchmarked, *i.e.*, Excel, Calc, and Google Sheets. We then explain our benchmarking experiment design and settings in Section 3. We present the results of our BCT and OOT benchmarking experiments in Section 4 and 5, respectively. We summarize the key takeaways from the experiments while discussing possible optimization opportunities in Section 6.

2 SPREADSHEET SYSTEMS OVERVIEW

We provide a brief overview of the spreadsheet systems that we are benchmarking, namely, Excel, Calc, and Google Sheets. While Excel and Calc are desktop-based systems, Google Sheets is web-based. We selected these three systems due to their popularity among users and adoption by major office suite software. Excel, part of the Office 365 suite [33], is the most popular desktop-based spreadsheet system, boasting about 700M registered users [32]. Google Sheets, part of G suite [15], is the most popular web-based spreadsheet system, with users numbering in the 100s of millions [39]. Calc is an open-source spreadsheet system used by two major open-source office software suites, Apache OpenOffice and LibreOffice [41]. We first explain the general constructs of a spreadsheet system and then discuss the aforementioned three systems in detail.

2.1 Spreadsheet Concepts

Spreadsheets provide a direct manipulation interface for organization, analysis, and storage of data in tabular form [34]. A spreadsheet is essentially a collection of cells arranged into rows and columns. Each cell within a spreadsheet has a style (e.g. color, height, or width) as well as data of a specific type. Cells in a spreadsheet can accommodate either values or formulae. Value data types include numbers, dates, percentages, among others. A formula, on the other hand, is an expression that evaluates to a value displayed in the cell. For instance, if the cell at the first column of the second row, *i.e.*, cell $B1$, contains the formula "=SUM(A1:A3)", $B1$ would display the sum of the contents $A1$, $A2$, and $A3$. If a user updates one of the cells $A1$, $A2$, or $A3$, the formula in $B1$ is recomputed to the correct consistent value. These computations take place *synchronously*, leading to performance issues as documented in recent work [8, 26]. Note that all systems determine the data type automatically at a cell level, with formulae ignoring cells with inadmissible data types. Apart from basic arithmetic and mathematical formulae, spreadsheet systems also provide built-in formulae for common finance and statistics functions [1], string manipulation operations, as well as GUI-based data summarization, *e.g.*, Pivot Table [4], and chart creation commands. In our experiments, we employ several of the most popular formulae including COUNTIF and VLOOKUP (described later).

2.2 Spreadsheet Systems

Existing spreadsheet systems can be divided into two categories based on the operating environment, namely, desktop-based or web-based systems. We now discuss the two types of systems.

2.2.1 Desktop-based Systems. The most popular desktop-based spreadsheet systems include Excel, Calc, and Numbers. Numbers only operates in MacOS, while Excel operates in both Windows and MacOS. Calc operates in Linux, MacOS, and Windows.

Excel. Excel can support up to 1M rows and 17,000 columns in a given spreadsheet [30]. The Windows version of Excel supports programming through Microsoft's Visual Basic for Applications (VBA), a dialect of Visual Basic [5]. VBA enables executing user-defined functions (UDFs), automating processes, and programmatically executing built-in Excel formulae. Programmers may write VBA code directly using the Visual Basic Editor, an IDE that can be launched from within Excel.

Calc. Calc is the spreadsheet system of the LibreOffice suite. Calc forked from Apache OpenOffice Calc, which suffers from various performance and security issues [3]. Calc can support up to one million rows in a spreadsheet [40]. Calc supports most of the basic functionalities provided by Excel. Calc also supports programming through Calc Basic [23], which can be written in an IDE similar to Visual Basic Editor.

2.2.2 Web-based Systems. The most notable web-based spreadsheet system is Google Sheets. Both Excel and Calc also have online counterparts: Excel Online, and LibreOffice Online, respectively, both of which are excluded from consideration. While Excel Online doesn't support macros to programmatically run experiments, development support for LibreOffice Online has been discontinued [2].

Google Sheets. Google Sheets is part of a web-based software office suite, G Suite, offered within Google Drive [15].

Google Sheets provides many of the basic functionalities of the desktop-based systems. The scale of data supported by Google Sheets is smaller than desktop-based spreadsheets, *i.e.*, five million cells per spreadsheet [17]. Google Sheets also supports programming through Google Apps Script where users can write custom functions and macros in JavaScript [16]. Unlike desktop-based systems, Google Sheets supports collaboration, for example, simultaneous editing of spreadsheets.

3 BENCHMARK SETUP

Next, we describe a taxonomy that groups spreadsheet operations into high level categories. The taxonomy enables us to perform targeted benchmarking of representative operations within each category. We then explain the datasets used and the experimental settings for the systems being benchmarked.

3.1 Taxonomy of Spreadsheet Operations

We first group spreadsheet operations into three categories: data load, update, and query. We then group the operations in each category further based on three dimensions: input, output, and expected complexity, as shown in Table 1. We omit simple operations such as addition/subtraction, which are $O(1)$. Here, we briefly explain the high level categories, and defer a detailed discussion for the next section.

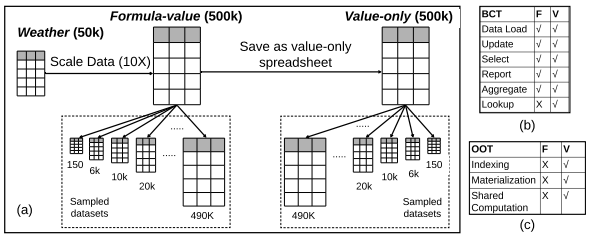
Data load operations involve loading data from disk (desktop-based systems) or a server (web-based systems). Two operations that fall under this category are *import* of a file into a spreadsheet and *open* of an existing spreadsheet.

Update operations change the content or style (or both) of spreadsheet cells. Depending on their goals, different operations may update a few cells at a time, *e.g.*, find and replace or conditional formatting, or an entire range of cells, *e.g.*, sort, copy-paste.

Query operations involve different statistical, arithmetic, data organization, summarization, and lookup formulae. We divide the query operations into four sub-categories: select, report, aggregate, and lookup.

3.2 Dataset

Following a university-wide survey that yielded 26 responses, we selected the largest real-world spreadsheet that was submitted—a spreadsheet on weather data across the states in US, containing 50000 rows and 17 columns. Cells within seven of those columns contained COUNTIF formulae. Each formula counts the presence of a value (natural disaster) in the corresponding cell of a preceding column, *e.g.*, the formula at cell $k2$ is: "=COUNTIF($C2$,"STORM")", evaluating to 0 or 1. We selected a real dataset to ensure that the organization of data and the ratio of formulae to values within the spreadsheet are both representative. Using this dataset as the starting point, we created various synthetic datasets and settings to evaluate different categories of spreadsheet operations and accommodate different dimensions of the benchmarking experiments. We repeated our experiments with other typical spreadsheet datasets as a starting point (see appendix for benchmarking results on other datasets), and we did not learn any new insights; so, we focus our attention on this dataset, and consider a number of its variations to stress-test various operations.



BCT	F	V
Data Load	√	√
Update	√	√
Select	√	√
Report	√	√
Aggregate	√	√
Lookup	X	√

OOT	F	V
Indexing	X	√
Materialization	X	√
Shared Computation	X	√

Figure 1: Datasets and benchmarks. (a) Creating synthetic datasets from a real-world spreadsheet by scaling and sampling. Outline of experiments along with datasets used in (b) BCT and (c) OOT benchmark.

Figure 1a shows how the synthetic datasets were created and Figure 1b and 1c show the experiments where each dataset was used. We first created a scaled-up version of the weather dataset, called Formula-value (F for short). This dataset has 500k rows—$10X$ the original dataset—where cells can contain either formulae or values. As explained in Section 1, the embedding of other formulae within a spreadsheet can influence the outcomes of a specific experiment due to recomputation of these embedded formulae. To isolate the effect of the embedded formulae, we converted the Formula-value spreadsheet to a value-only spreadsheet, called Value-only (V for short), where any formulae were replaced by the corresponding value. To evaluate how computation time varies with size, we created 51 different versions of Value-only and Formula-value with increasing row sizes simulating input ranges. The number of columns in each dataset was fixed. We created multiple dataset versions (51) by uniformly sampling rows based on the *state* column of the $500k$ rows dataset. The two smallest dataset versions contained 150 and 6000 rows. For the rest of the 49 dataset versions, the number of rows were $N_i = 10000 + (i-3) \times 10000$, where $i = 3, 4, 5, \ldots, 51$.

Figure 1b and 1c shows the experiments where each dataset was used. Except for the experiment involving the lookup operation, all of the BCT experiments were run on both datasets. As the lookup operation has four parameters, we simplified the experiment by only using the Value-only dataset to better understand the impact of the parameters. We conducted a total of seven BCT experiments that benchmark six categories of spreadsheet operations (discussed in Section 4)—these categories encompass a wide range of spreadsheet operations from formulae to GUI-based operations. For the OOT benchmark, we conducted targeted experiments to identify the existence of database-style optimizations within spreadsheet systems. These experiments required us to run spreadsheet

Table 1: Categorizing Spreadsheet Operations. For input type "Range", there are m rows and n columns.

Category	Sub-category	Example	Input	Output	Expected Complexity
Data Load	—	Open, Import	Filename	Range ($m \times n$)	$O(mn)$
Update	—	Find and Replace	Range ($m \times n$), Value X and Y	Updated cells	$O(mn)$
		Copy-Paste	Range ($m \times n$)	Range ($m \times n$)	$O(mn)$
		Sort	Range ($m \times n$)	Range ($m \times n$)	$O(m \log m)$
		Conditional Formatting	Range ($m \times n$), Condition	Updated cells	$O(mn)$
Query	Select	Filter	Range ($m \times n$), Condition	List	$O(mn)$
	Report	Pivot Table	Range ($m \times n$), Condition	Aggregate Table	$O(mn)$
	Aggregate	SUM,AVG,COUNT	Range ($m \times n$)	Value	$O(mn)$
		Conditional Variants	Range ($m \times n$), Condition	Value	$O(mn)$
	Lookup	Vlookup, Switch	Range X ($m_x \times n_x$) Value, Range Y ($m_y \times n_y$)	Value	$O(m_x n_x m_y n_y)$

operations in isolation, without being impacted by the recomputation of the embedded formulae within spreadsheets. Therefore, we only used the Value-only datasets in the OOT benchmark. We conducted six OOT experiments for identifying a number of optimizations, *i.e.*, indexing, columnar data layout, shared computation, and incremental computation.

3.3 Settings

For the desktop-based spreadsheet systems, we conducted all the experiments on a Dell Precision 490 workstation with Intel Xeon E5335 2.0GHz CPU and 16GB RAM running 64 bit versions of Windows 10 and Ubuntu 16.04. The Excel-based experiments were conducted with Microsoft Excel 2016 running on Windows, while the Calc-based experiments were conducted on LibreOffice Calc 6.0.3.2 running on Ubuntu. The Google Sheets-based experiments were run on a university allocated G Suite account. For all three spreadsheet systems, we implemented the experiments in their corresponding scripting language, *i.e.*, Visual basic (*VBA*) for Excel, Calc basic for Calc, and Google apps script (*GAS*) for Google Sheets. All the experiments were single threaded. Note that Excel 2016 can be configured to support multi-threaded recalculation of formulae [31]. However, the default setting is to evaluate a formula on the main thread of Excel. To ensure that all experiments operated on the entire dataset, we selected the desired data within the spreadsheet via a macro command.

For each experiment in Excel, we first created an Excel Macro-Enabled Workbook (*xlsm*) [43] which can execute embedded macros programmed in VBA. Unlike Excel, LibreOffice Calc macros, programmed in Calc Basic, can be enabled and executed from the default workbook—OpenSpreadsheet Document (*ods*) [42]. We created the Google App Scripts in G Suite Developer Hub [16]. Given an experiment, all three scripting languages can invoke a formula, *e.g.*, COUNTIF, or operation, *e.g.*, sort, for their respective systems via an API call. We used default library functions of the corresponding scripting languages to measure the execution time of each experimental trial. For each experiment, we passed the file path of the relevant datasets as an argument for the scripts (macros) of the desktop-based systems, and a URL for GAS in Google Sheets. All the datasets used in the Excel and Calc-based experiments were in *xlsx* and *ods* format, respectively. The datasets used in the Google Sheets experiments were uploaded as *xlsx* files and then manually converted to Google Sheets from the Google Drive menu.

For each experiment, we ran ten trials and measured the running time. We report the average run time of eight trials while removing the maximum and minimum reported time. Note that the Google Sheets experimental settings were limited by the daily quotas and hard limits imposed by Google Apps Script services on some features, like API calls and the number of spreadsheets created and accessed. As a result, for experiments with Google Sheets, we restricted the maximum size of the data to 90k rows to fit in the experiment trials for different test cases within the allocated daily quotas . Therefore, we report the results of Google Sheets in separate charts alongside the desktop-based systems. Moreover, we display error bars for the Google Sheets experiments as the trends exhibited higher variances across trials. Repeating these experiments with various settings, *e.g.*, randomization of trials or using a unique sheet per trial, reduced the variance while exhibiting similar trends. These experiments with lower variance, as well as other experiments we tried are detailed in Section 4.

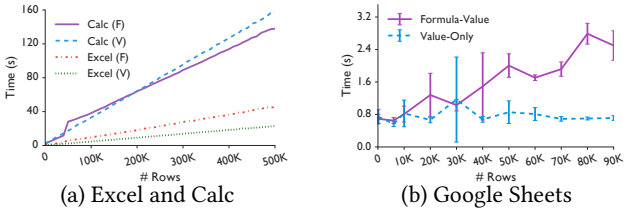
4 BCT BENCHMARK

In this section, we present the results from the BCT benchmarking experiments. The BCT benchmark is designed to quantify the impact of three aspects on the latency of an operation: (a) type of operation, (b) size of data operated on, and (c) spreadsheet system used. For each experiment, we select a representative operation from each category in Table 1. Given an operation, we gradually increase the data size being operated on, record the time taken to complete the operation for each system, and compare the observed time complexity with the expected one. We further evaluate when, if at all, the execution time for a given formula violates the interactivity bound of 500ms [25]. We denote the number of rows and columns in a spreadsheet by m and n, respectively. In our experiments, we typically vary m while keeping n fixed. Therefore, we expect the time complexity of a formula to vary with row count, m.

4.1 Data Load Operations

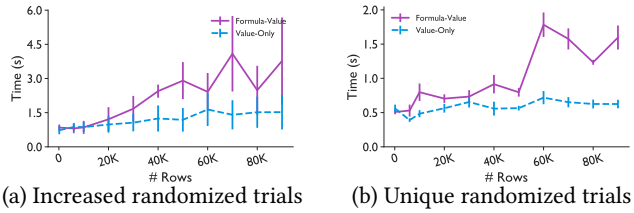
Users can perform data load operations (see Table 1) via button-clicks from the spreadsheet menu bar. While import involves loading data from any existing file in the disk to a blank spreadsheet in memory, the open operation loads an existing spreadsheet from disk to memory. For the data load operations, the expected worst-case complexity is $O(mn)$,

i.e., the total number of cells. As these operations are essentially equivalent, we only evaluate the open operation. This operation takes the file path as input and loads the file from disk to memory. We document the time to open Formula-value (F) and Value-only (V) datasets, while varying row sizes m, where $m = 150, 6k, 10k, 20k, \ldots, 500k$. As we keep the number of columns fixed, the expected complexity is $O(m)$.



(a) Excel and Calc (b) Google Sheets

Figure 2: *Open* in Excel, Calc is slow; it is faster on Google Sheets due to lazy loading of data not in the user window.



(a) Increased randomized trials (b) Unique randomized trials

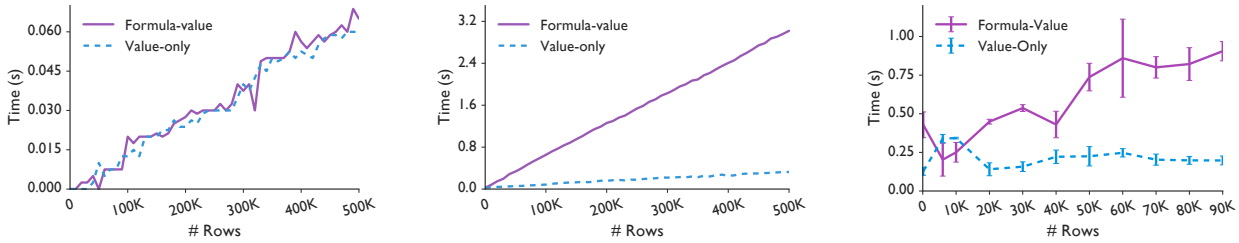
Figure 3: For *Open* operation in Google Sheets, using a unique sheet per trial while randomizing order helped reduce the variance, compared to increasing the number of trials.

Observations. Figure 2a shows that the time taken by the desktop-based spreadsheets is linear in m for both datasets. Recall that the Formula-value datasets have formulae embedded alongside values while the Value-only datasets only contain values. On the other hand, in Google Sheets, the time to open the Value-only spreadsheet is almost the same, independent of the size of the dataset, *i.e.*, $O(1)$ (see Figure 2b). When opening a spreadsheet for the first time, *Google Sheets appears to load the first m rows visible within the screen, and then load the rest on-demand* as the user scrolls. We have confirmed this observation by manually scrolling through a Google Sheets spreadsheet. However, Google Sheets breaks the interactivity threshold of 500ms to load even a screenful of data, possibly due to network or web rendering delays [18]. On the other hand, *Excel and Calc violate this bound while opening only 6000 and 150 row* Value-only datasets, respectively, well below their advertised scalability limit of one million rows. The delay is even worse for Formula-value datasets. Even though the row sizes at which the interactivity bound breaks for Excel, Calc, and Google Sheets is roughly the same as that for Value-only datasets, *i.e.*, 6000, 150, and 150 rows, respectively, the slope of the line chart for the Formula-value dataset is steeper than that for the Value-only datasets. The only difference between the Formula-value and Value-only datasets is the presence of embedded formulae. When the spreadsheet is opened, the spreadsheet systems recalculate embedded formulae (as discussed in Excel documentation [28], but we expect other systems are similar), and as the number of embedded formulae increases, the latency of open increases as well.

Specifically, Excel and Calc go past the one minute mark at 40k and 6k rows, respectively. Google Sheets performs much better compared to the desktop-based spreadsheets, taking $\approx$ 40 seconds to load a 90k rows spreadsheet. Surprisingly, even after loading a screenful of data, the time to open a spreadsheet in Google Sheets increases linearly with the size for the Formula-value datasets. We speculate that the latency may stem from performing additional computation on the server to resolve issues like formula dependencies on the entire spreadsheet, before sending data to the client. Thus, beyond prioritizing loading the first "window" of the spreadsheet, there are additional opportunities to reduce the latency of data load by prioritizing formula computation for the first window, done by no systems currently. In Section 6, we discuss how spreadsheet systems can optimize data load using ideas like these.

Variance Across Trials for Google Sheets: As mentioned in Section 3, for all the Google Sheets experiments, we observed higher variance across trials for different datapoints, *i.e.*, row sizes. For example, for the *Open* experiment, Google Sheets exhibited higher variance for 20k and 40k Value-only and 30k Formula-value datasets (see Figure 2b). Therefore, we repeated these experiments with two different settings in an attempt to further reduce the variance. For both the settings, we randomized the order of the trials for each datapoint as opposed to the setting presented in this paper, where we run all the trials for each datapoint sequentially. One of these settings, called *increased randomized trials*, involved running 20 trials while using the same spreadsheet for each trial, for each datapoint. The other setting, called *unique randomized trials*, involved running 10 trials while using a new spreadsheet for each trial, for any given datapoint.

According to Figure 3, using a unique sheet per trial resulted in a reduction of variance across trials for all the datapoints. We speculate that using a different spreadsheet for each trial ensured a similar setting for all the trials where, the *Open* operation underwent a cold-start while loading a completely new spreadsheet. Note that across different settings, the underlying trend of the operation is still the same—while Google Sheets performs lazy loading of data outside of the currently visible user window, the operation is impacted by the presence of embedded formulae in the Formula-value datasets. We observed similar behavior for other experiments as well. The implementation details of the *unique randomized trials* setting of all the experiments are available in our benchmarking repository.



(a) Excel (b) Calc (c) Google Sheets

Figure 4: While *conditional formatting* on Formula-value is slow for Calc and Google Sheets due to formula recomputation, no such recomputation is triggered in Excel. Google Sheets is faster for Value-only due to formatting cells in a lazy fashion.

Takeaway: *The desktop-based spreadsheets violate the interactivity bound when opening even small spreadsheets of less than* 10*k rows. The presence of formulae makes the open operation even slower for all the systems. Google Sheets lazily loads data outside of the first user window, thereby returning control quickly, but fails to do so for sheets with embedded formulae.*

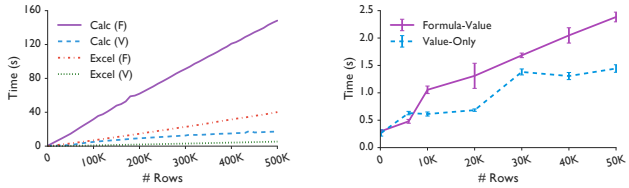
4.2 Update Operations

We now consider two update operations: conditional formatting and sort. We present the results for find-and-replace along with the OOT benchmark results in Section 5.

4.2.1 Conditional Formatting. The conditional formatting operation takes a data range and a conditional expression as input and updates the style of the cells within the range that satisfy the condition. As before, we ran this experiment on Value-only and Formula-value while varying the row count, m. We measured the time to execute an operation to color cells in a column green if they contains the value 1. The expected complexity for this experiment is $O(m)$, where m is the row count.

Observations. Figure 4 shows that although Excel and Calc exhibit a linear trend for Value-only datasets, Google Sheets takes almost the same time to complete the operation irrespective of the size of the dataset. We again speculate that Google Sheets updates the style of visible cells, doing the rest lazily. Excel and Google Sheets complete the operation within an interactive bound for both datasets. On a 90k spreadsheet, Excel completes the operation in 7.5ms, which is 10.6X and 26.31X faster than Calc (79.5ms) and Google Sheets (197.375ms), respectively. For Formula-value datasets, Excel requires almost the same time as the Value-only datasets. However, for both Calc and Google Sheets, the trend is much steeper. Both the systems violate the interactivity bound with datasets much smaller than their scalability limits—at 80k and 50k rows, respectively. The values of the cells being formatted for Formula-value datasets are derived from formulae; the gap between Formula-value and Value-only for Calc and Google Sheets may stem from an unnecessary recomputation triggered by formatting.

Takeaway: *While all systems perform conditional formatting somewhat efficiently, Google Sheets appears to do the formatting lazily for data not in the user window. Calc and Google Sheets perform unnecessary formula recomputation for Formula-value, violating interactivity at datasets with fewer than* 80*k rows.*



(a) Excel and Calc (b) Google Sheets

Figure 5: Sort on Formula-value is substantially worse than Value-only, thanks to formula recomputation on sort.

4.2.2 Sort. As shown in Table 1, the sort operation takes one or more column references, the sort order, *i.e.*, ascending or descending, and a range of cells, all as input and reorganizes the range of cells in the order of the referenced columns. Unless specified explicitly, the input range is the entire spreadsheet. In our experiments, we sort the data by a single attribute—column A of unique integer values, with an expected complexity of $O(m \log m)$, where m is the row count (or size of dataset); see Figure 5.

Observations. The deceptively linear trend for sorting for all systems is due to the size of the datasets used in our experiments—even row size m = 500k is not large enough for the logarithmic factor to be pronounced for the $O(m \log m)$ trend. For Google Sheets, we could not run our experiments beyond the 50k row dataset due to a G-Suite imposed limit on the time budgeted for an experiment. Similar to data load operations, Excel, Calc, and Google Sheets violate the interactivity bound for both Value-only (70k, 10k, and 6k rows, respectively) and Formula-value (10k, 150, and 10k rows, respectively). Again, the recomputation of embedded formulae increases the latency with interactivity bounds violated much earlier—compared to the Value-only dataset (70k), Excel breaks the bound with 7X smaller Formula-value dataset (10k). As the sort operation reorganizes the data, the regions within the spreadsheet that a formula referred to prior to sorting could possibly possibly be populated with new data, triggering a recomputation of formulae, as we saw for the open operation [28]. However, such recomputation is not always necessary—when the formulae references are relative, sorting the entire spreadsheet does not change the results of

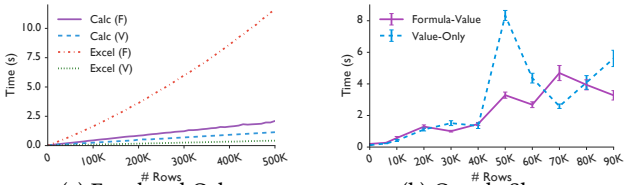
a formula. For example, if every entry of column C is simply the sum of the entries of column A and column B, *e.g.*, $C1 = A1 + B1$, then sorting the spreadsheet across rows based on column A should not require a recomputation of the formulae. In this case, the recomputation is wasted computation: here, formulae generate derived columns and are specific or local to a row, and therefore do not require a recomputation when the rows are sorted. In Section 6, we discuss how spreadsheets systems can adopt dynamic reordering strategies to perform sorting in interactive times [38].

Takeaway: *All three spreadsheet systems violate the interactivity bound for sort on very small datasets (less than* 10*k for Formula-value), with Excel doing better than Calc. Sorting triggers formula recomputation that is often unnecessary and can take an unusually long time.*

4.3 Query Operations

We now discuss the results for four query operation categories: select, report, aggregate, and lookup. Both the input and output of such operations can vary (Table 1); inputs can include values, conditions, or ranges; outputs can include values, ranges, lists, or aggregates.

4.3.1 Select (Filter). In this experiment, we filter a given spreadsheet by state *SD* (South Dakota). Filter operations in spreadsheets hide the rows that do not satisfy the filtering condition and is therefore like filters in relational databases. For example, in our experiments, any row for which state $\neq SD$ will be hidden. We vary the row count, m, and expect the run time to be linear in m. This is because the filter would require a full scan of the dataset.



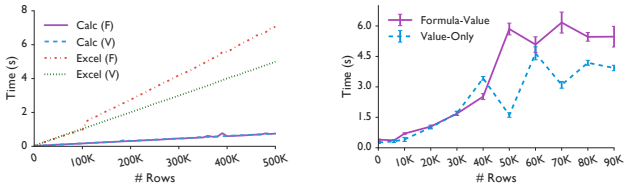
(a) Excel and Calc (b) Google Sheets

Figure 6: *Filter* on Formula-value in Excel does unnecessary recomputation. Google Sheets is slower than the other two.

Observations. As can be seen in Figure 6, all systems exhibit a linear trend for Value-only. Excel completes the operation within 500ms for even for 500k row dataset. However, Calc and Google Sheets violate the bound at 200k and 20k datasets, respectively. *Excel exhibits a super-linear trend for Formula-value datasets and violates the* 500*ms bound at* 40*k rows* (Figure 6a). Filtering likely triggers unnecessary formula recalculation in Excel [28], but why the trend is super-linear is a mystery to us. For Formula-value, the times for Calc and Google Sheets is similar to Value-only, with interactivity violated at sizes 120k and 10k, respectively. Filter likely does not trigger recalculation in these systems.

Takeaway: *Filtering takes a suspiciously long time for Formula-value for Excel, violating interactivity at* 40*k rows, possibly due to formula recomputation. The other systems avoid this recomputation, but are slower than Excel for Value-only datasets.*

4.3.2 Report (Pivot Table). The pivot table operation [13] creates a table with summary statistics (similar to a SQL GROUP BY). Users can generate a pivot table on one or more dimension attributes and measure attributes. The operation scans the entire dataset and creates a summary table in a new or existing worksheet with the results. In this experiment, we create a pivot table that shows the *sum of storms* per state in a new worksheet. Here, the dimensions attribute corresponds to the state column while the measure attribute corresponds to the number of storms column. We again expect the results to be linear with respect to the number of rows of the dataset.



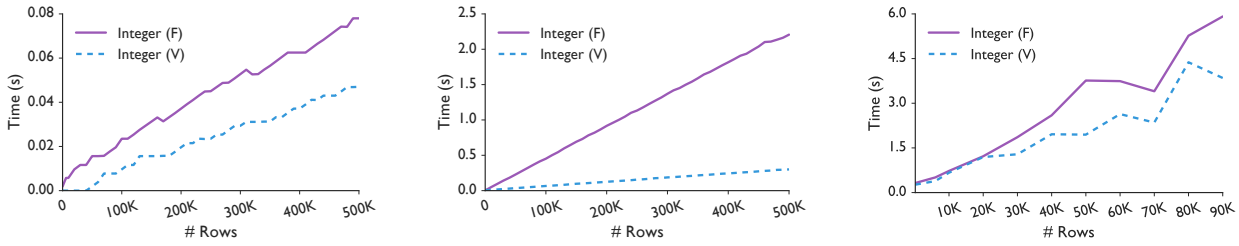
(a) Excel and Calc (b) Google Sheets

Figure 7: Calc is faster than the other two for Pivot Tables

Observations. Figure 7 demonstrates linear complexity for both types of datasets. For Value-only datasets, Calc outperforms (330k rows) both Excel and Google Sheets—the latter two violate interactivity at 50k and 20k rows, respectively. Similar patterns emerge for Formula-value where Calc outperforms (340k rows) Excel (50k rows) and Google Sheets (10k). Moreover, while Calc is unaffected by embedded formulae, both Excel and Google Sheets exhibit higher latency for Formula-value. We hypothesize that insertion of a new worksheet in the workbook triggers formula recomputation for Excel and Google Sheets.

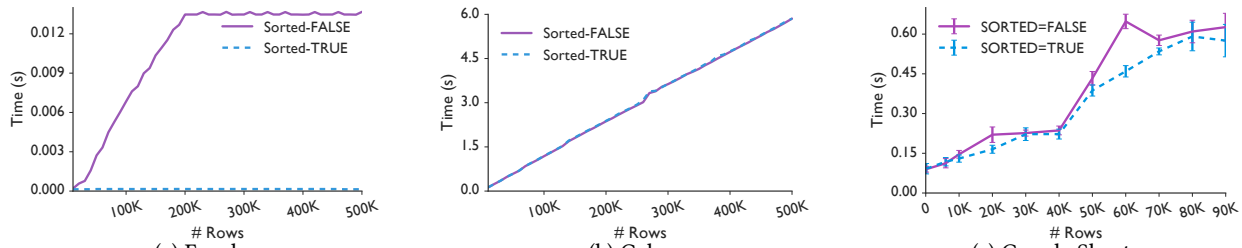
Takeaway: *Calc accommodates* 6× *(Excel) or* 15× *(Google Sheets) the dataset size for Value-only before violating interactivity for pivot tables. Calc avoids a costly formula recomputation for Formula-value, while the others do not.*

4.3.3 Aggregate operation. An aggregate formula, *e.g.*, COUNT, takes a range as input and then computes the aggregate of the values within that range. The conditional variant of an aggregate formula, *e.g.*, COUNTIF, takes an additional condition as input. For conditional variants, only the cell values that satisfy the condition are aggregated. We first measured the execution time of the non-conditional variants, *e.g.*, AVERAGE, SUM, and COUNT, and observed that their execution times were very similar for any given dataset. We ran similar equivalence tests between the conditional variant of the operations and observed the same pattern as above. Moreover, both formula variants, *i.e.*, non-conditional and conditional, exhibited a similar trend. Therefore, here we discuss the



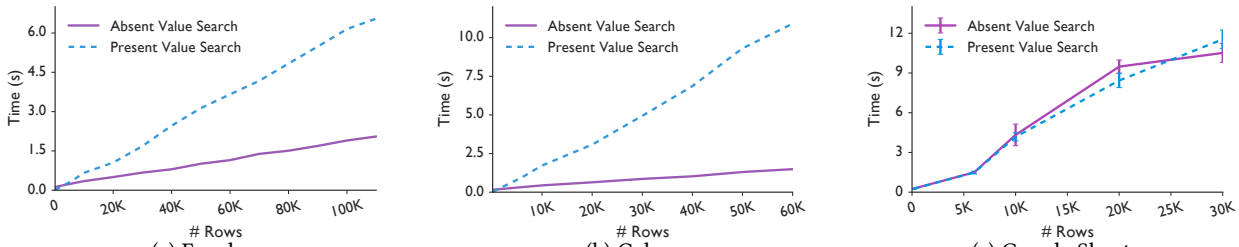
(a) Excel (b) Calc (c) Google Sheets

Figure 8: COUNTIF **is extremely fast in Excel compared to Calc and Google Sheets. However, for both Excel and Calc, latency is higher in Formula-value due to formula recomputation.**



(a) Excel (b) Calc (c) Google Sheets

Figure 9: For VLOOKUP**, while Excel terminates after finding a matching value, Calc and Google Sheets continue to scan the entire data. Excel optimizes approximate search (Sorted=True) via an efficient searching algorithm,** *e.g.*, **binary search.**



(a) Excel (b) Calc (c) Google Sheets

Figure 10: A linear trend for ***Find and Replace*** **indicates the absence of an index.**

results of a representative conditional aggregate formula, COUNTIF. A COUNTIF formula counts the number of cells in the input range that satisfy the given condition. We used the following formula in our experiment: "=COUNTIF($K2 : Km$, 1)", where $m = 150, 6k, 10k, 20k, \ldots, 500k$. We expect the time to execute the formula to scale linearly with the the number of rows in the input range. For Formula-value, the cells in the column K contain a COUNTIF formula themselves the result of which is either 0 or 1. The formula for cell ki is "=COUNTIF(Ci,$'Storm'$)", *i.e.*, this formula counts whether the cell Ci contains the string "Storm" ($Ki = 1$) or not ($Ki = 0$).

Observations. Figure 8 shows the results of the COUNTIF formula execution. For all of the Value-only datasets, Excel and Calc complete the operation in less than 500ms. However, Excel completes the operation earlier than Calc. The execution time is even worse for Google Sheets which often takes minutes while violating the interactivity bound at 10k rows of data. For Formula-value, again the order of execution time performance is Excel < Calc < Google Sheets. While Excel completes the operation within 500ms for Formula-value, both Calc and Google Sheets violate the interactivity bound at 110k rows and 10k rows, respectively. We speculate that issuing a COUNTIF formula over a cell in column K, *i.e.*, Ki, the value of which is a result of another formula, triggers a recalculation at that cell.

> **Takeaway:** *Even though the aggregate computation times scale linearly with the size of the data, both Calc and Google Sheets violate the interactivity bound well below their documented limits. The presence of formulae within spreadsheets severely impacts the aggregation performance of Google Sheets.*

4.3.4 Look Up. These operations look up a specific value X within a given input range and returns the value of another cell within the same row where X was found, *e.g.*, VLOOKUP. In our experiment, we perform a VLOOKUP on column A where $A_i = i$, *i.e.*, the value of the cell A in the i-th row is i. The VLOOKUP formula scans the column Asearching for an integer X and return the corresponding US state for the row i such that $A_i = X$, where $X = 200000$. For this reason, one can imagine this operation to be akin to a tuple-wise foreign-key lookup from a tuple in one relation to another relation.For all systems, VLOOKUP takes an optional binary parameter, indicating an approximate match (True) or an exact one (False). In our experiment, we also varied this parameter to see how the formula behaves with different search requirements. The

spreadsheet must be sorted for approximate match to work properly; so we sorted the dataset by column A first. The worst case expected complexity of the VLOOKUP operation is $O(m)$, *i.e.*, the entire input range is scanned when the value being looked up does not exist; however, with appropriate indexing this operation can be expected to take near-constant time. As explained earlier, to better understand the impact of the matching criteria interacting with the size, we only used the Value-only datasets. Using Formula-value datasets would have introduced another dimension to the experiment, presence of embedded formulae, making it difficult to understand the impact of the input parameters.

Observations. Figure 9 shows that VLOOKUP times vary significantly across systems. When the parameter is set to *False*, *i.e.*, exact match, Excel terminates after finding the value at the $200k$-th row. For datasets with $N < 200k$, Excel ends up scanning the entire data as no matching value is found. In both cases, Excel completes the lookup operation in less than 500ms. When it is set to *True*, *i.e.*, approximate match, Excel exhibits almost constant run time. We speculate that Excel performs additional optimizations, *e.g.*, binary search, for fast computation on sorted data. As $\log_2 500000 \approx 19$, this amounts to roughly 19 comparisons in memory, which should be extremely fast. Surprisingly, even with the sorted dataset, if the matching criteria is set to False, Excel reverts to a linear trend. In Section 5, we argue that a lack of indexing of data leads to such behavior.Unfortunately, neither Calc nor Google Sheets perform any optimizations and scan the entire dataset even after finding the value being looked up, violating interactivity at 50k and 60k respectively. Recall that a single VLOOKUP is like a single foreign key-based lookup; a collection of such lookups is therefore a foreign key join. For example, a popular usage of VLOOKUP is to look up grades from a grade table (X) for a collection of scores (Y). While this operation on a few hundreds of thousands of rows would take minutes in memory for spreadsheets, it would take less than a second within a database, as was mentioned in recent work [35].

Takeaway: *Calc and Google Sheets end up scanning the entire dataset for VLOOKUP irrespective of whether a matching value is found, violating the interactivity bound for datasets more than $50k$ and $60k$ rows, respectively. However, Excel is often efficient for sorted data, but requires the user to explicitly set the parameter that decides the lookup strategy.*

4.4 Discussion

Table 2 summarizes the results of the BCT experiments, showing the percentage of their advertised limits, *i.e.*, 1M rows for Excel and Calc and 5M cells for Google Sheets, at which the corresponding system begins violating the interactivity bound of 500ms. To obtain this percentage, we first identify the the number of rows at which interactivity is violated. We then divide that number of rows by 1M for desktop-based spreadsheets. For Google Sheets, we compute the total number of cells, given the number of rows and then divide that by 5M. *Overall, despite performing computation in memory, except for a handful of cases in Gray in Table 2, spreadsheet systems fail to provide interactive responses for even small datasets.* The interactivity is even worse with embedded formulae. While spreadsheet systems perform optimizations such as visible window prioritization or binary search, these methods are applied to bespoke conditions, resulting in high latency for most operations.

Table 2: A summary of BCT experiments. For each experiment, we show at what percentage of their advertised limits, Excel (E), Calc (C), and Google Sheets (G), violate interactivity. A value of 100% means it wasn't violated.

	Formula-value			Value-only		
	E (%)	**C** (%)	**G** (%)	**E** (%)	**C** (%)	**G** (%)
Open	**0.6**	0.015	0.05	**0.6**	0.015	0.05
Sort	1	0.6	**3.4**	**7**	1	2.04
Conditional Formatting	**100**	8	17	**100**	**100**	**100**
Filter	4	**12**	3.4	**100**	20	6.8
Pivot Table	5	**34**	3.4	5	**33**	6.8
COUNTIF	**100**	11	3.4	**100**	**100**	3.4
VLOOKUP	×	×	×	**100**	5	23.8

We aim to uncover the causes for high latency in the next section. We try to understand how spreadsheets systems store and organize datasets. Do they use indexing? Do they optimize the layout of the data in-memory to allow for efficient data access for computation? Next, spreadsheet systems tend to perform poorly when an operation triggers recomputation of embedded formulae. Therefore, we want to understand how spreadsheet formula computation happens: How do spreadsheets perform recomputation after an update? Do they reuse the results of the previous or other computations to optimize a given formula? We attempt to answer these questions in the next section.

5 OOT BENCHMARK

Next, we present results from the OOT benchmark that investigates whether spreadsheet systems adopt classic "database-style" optimizations such as indexing, intelligent and compact data layout, shared computation, eliminating redundant computation, and incremental updates. We focus on Value-only as we want to eliminate the effects of other embedded formulae. We evaluate indexing-based optimization opportunities for both querying and update operations, while focusing on querying operations like aggregate, report, and lookup for the rest, *i.e.*, data layout, shared and incremental computation, since they can benefit most from these optimizations, using COUNTIF, SUM, and VLOOKUP as representatives.

5.1 Indexing

We now explore whether spreadsheets maintain indexes on the columns to facilitate faster computation for find-and-replace, COUNTIF, and VLOOKUP. With indexes, such operations will be executed in near constant time, *e.g.*, logarithmic in the data size, say with B+ trees.

5.1.1 COUNTIF and VLOOKUP. To understand whether spreadsheets maintain indexes, we first briefly recap the results of the BCT experiments on aggregation and lookup operations. As we saw in Figure 8, the observed complexity of COUNTIF is linear in the size of the dataset for all three spreadsheet systems. VLOOKUP also exhibits a similar linear trend in Calc (see Figure 9b) and Google Sheets (see Figure 9c). However, in Excel, for sorted data, with the matching criteria set to "approximate match", *i.e.*, True, VLOOKUP may be performing some optimizations in the form of binary search. However, even with sorted data, when the matching criteria is set to "False", Excel exhibits a linear trend which indicates an absence of indexes. All of the other operations except data load that were benchmarked in Section 4 exhibit linear trends or worse, *e.g.*, superlinear trend for filter, further confirming that spreadsheet systems do not employ indexes.

5.1.2 Find and Replace. For find-and-replace, we wanted to see if spreadsheet systems perform inverted indexing, a popular indexing mechanism employed by search engines for efficient information retrieval [44]. Find-and-replace takes three inputs: an input range and two values, X and Y, and then scans the input range, one cell at a time, replacing any X with Y. For this experiment, we randomly insert a predefined fixed search string X within one column and replace X with another string Y. We run the following experiments: (a) find a predefined string and replace it with another, and (b) search for a nonexistent value. With an inverted index, we expect the time complexity of this operation to be constant.

Observations. For Excel, Calc, and Google Sheets, we run the experiments up to $110k$, $60k$, and $30k$ rows, respectively (see Figure 10). For Google Sheets, the operation timed out beyond 30k rows. The desktop-based systems also took seconds to complete the operation for larger datasets. Therefore, we discontinued our experiments beyond the row ranges mentioned.For all three systems, we see a linear trend that violates interactivity at 10k, indicating the absence of indexes. Even when searching a non-existent value, the search time scales linearly. As the value doesn't exist, replace is skipped, leading to faster completion for a non-existent value. Surprisingly, Google Sheets takes the same time in both cases.

Takeaway: *None of the spreadsheet systems maintain indexes, as is evidenced by the fact that the execution time linear in the size of the data. Find-and-replace is especially problematic, taking more than* $500ms$ *for all datasets* $> 10k$.

5.2 Efficient and Intelligent Data Layout

Next, we wanted to see whether spreadsheets employ an intelligent layout of data in memory. As most formulae operate on contiguous cells, physically laying out cells near each other on the sheet close to each other can benefit from cache locality. For the first set of experiments, we use three different sizes of Value-only: 100k, 300k, and 500k. For our next experiment, we aim to evaluate the dataset sizes in memory vs. that on disk, to evaluate how efficiently various systems represent data in memory.

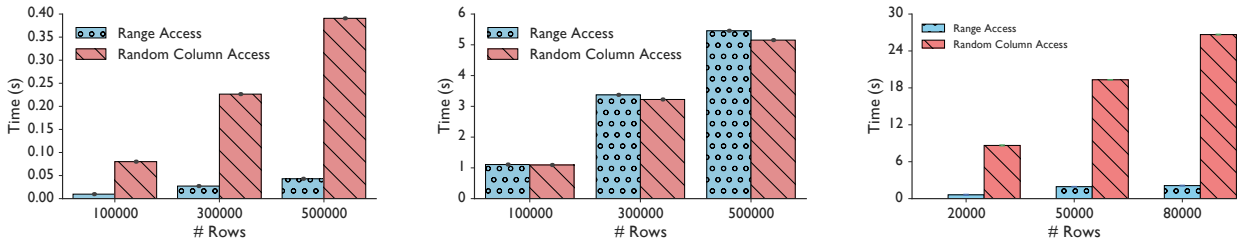
5.2.1 Range vs. column access. To assess how data is laid out for various systems, we first run two experiments: range access and random column access. For range access, we scan a spreadsheet range and count the total number of cells in the range, *i.e.*, issue COUNT(A1:Sn), where n = 100k, 300k, 500k. For random column access, we randomly select an entire column between columns A to S, count the number of cells in that column, *e.g.*, COUNT(A1:An), and then add all the counts. If a spreadsheet system employs a row-oriented layout, we expect range access to be faster than column access. The vice-versa is true if a column-oriented layout is used.

Observations. As shown in Figure 11a and Figure 11c, range access is orders of magnitude faster than random column access for both Excel ($\approx 10X$) and Google Sheets ($\approx 11X$), respectively, indicating a row-oriented data layout. However, the execution time for both types of data access is similar in Calc (see Figure 11b). To further verify that Calc uses a column-oriented data layout, we employ a second set of experiments, discussed next.

5.2.2 Sequential vs. Random access. Our second set of experiments involve comparing sequential and random data access. For the former, we scan a spreadsheet column (A) from beginning to end while accessing the values of each cell. In all three scripting languages, VBA, Calc Basic, and GAS, we can access the value of a cell via an API call, by providing the row and column id of that cell. For the latter, we randomly select a row and access the cell corresponding to column A within that row. If a columnar layout is used, sequential access would be faster than random access due to cache locality.

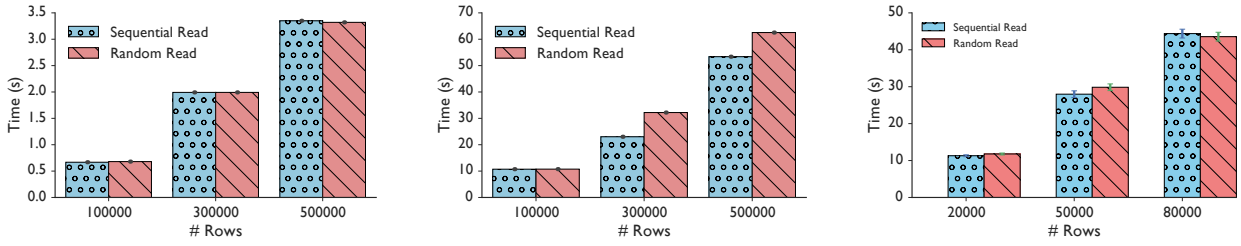
Observations. The two experiments take almost the same time for Excel and Google Sheets (see Figure 12a and c), reaffirming a row-oriented data layout. However, as shown in Figure 12b, sequential access is faster than random access in Calc, indicating the presence of a columnar data layout. We later learned from the Calc development team that Calc employs a columnar MDDS data-store [27] which explains better sequential data access performance. However, the improvement in sequential access is not proportional to the number of rows accessed.

5.2.3 Memory and Disk Consumption. So far, we have focused on performance; however, when discussing data layout, it is also valuable to consider data size. How much do datasets "blow up" in memory relative to disk? In relational databases, pages on disk are mapped into pages in the buffer pool, leading to memory consumption that is not just bounded by the buffer pool size, but also occupies similar space as on disk. We are aware that spreadsheet systems precompute



(a) Excel (b) Calc (c) Google Sheets

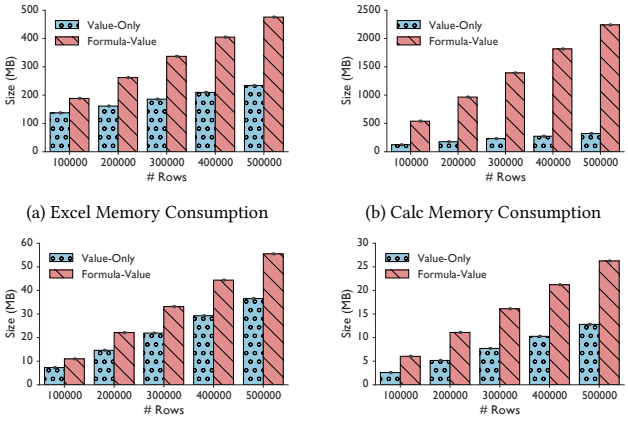
Figure 11: While Excel and Google Sheets employ a row-oriented data layout, Calc seems to employ a columnar data layout.



(a) Excel (b) Calc (c) Google Sheets

Figure 12: While Calc employs a columnar data layout, for Excel and Google Sheets, sequential and random access of a column takes roughly the same time confirming that these systems employ a row-oriented data layout.

the cell-dependency graph and formula calculation chain—both these data structures are loaded in memory along with the spreadsheet impacting Formula-value datasets [28]; so we compare Formula-value and Value-only datasets for this experiment, and focus on desktop-based systems.



(a) Excel Memory Consumption (b) Calc Memory Consumption

(c) Excel Disk Usage (d) Calc Disk usage

Figure 13: Formula-value datasets tend to consume more memory and disk space than Value-only datasets due to the added optimizations, with the representation in memory being $10-90\times$ larger.

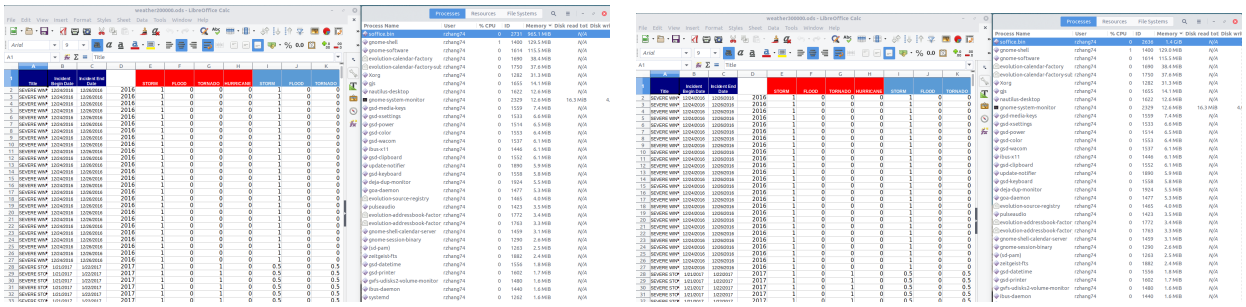
Observations. Figure 13 shows the disk and memory size of the desktop-based systems for Value-only and Formula-value. We measure the disk usage by both manually inspecting the file sizes from the *File Explorer* application of the respective operating systems and also via shell scripts. To measure memory consumption in Excel, we opened a spreadsheet via VBA macros and then invoked a Windows API call to measure the memory usage of the running Excel process. In Calc, we employed a bash script to open a spreadsheet as well as measure memory consumption. Excel's in-memory representation is up to $10\times$ that of the datasets on disk, while Calc's representation is up to $87\times$ for Formula-value, and $33\times$ for Value-only. Thus, in Calc, a 25MB spreadsheet can take more than 2GB in memory, leading to Calc exhausting memory sooner than Excel. For both systems, the relative size increase going from disk to memory is larger for Formula-value than Value-only. We also verified the excessive memory usage of Calc using the *System Monitor* application in Ubuntu (see Figure 14).

> **Takeaway:** *While Excel and Google Sheets employ a row-oriented data layout, Calc employs a columnar data layout resulting in improved columnar data access. Both desktop-oriented systems occupy* $10-90\times$ *more space in memory than on disk, with Formula-value datasets occupying relatively more space in memory than on disk.*

5.3 Shared Computation

In Section 4, we identified that recomputation of existing formulae severely impacts the execution time of any new formula. We want to understand why this recomputation is so expensive. As many formulae reference the same region, we wanted to see if these formulae share accesses, and if possible, share computation of sub-expressions.

We conduct an experiment where we insert a formula within each cell i of a column that computes the following: $\sum_{j=1}^{i} Aj$, *i.e.*, the cumulative sum of cells of column A up to row i (see Figure 15a) where $10k \leq i \leq 100k$—we use the Value-only dataset while varying the row count from 10k to 100k with a step size of 10k. One way to compute this cumulative sum is the *repeated* computation approach, using



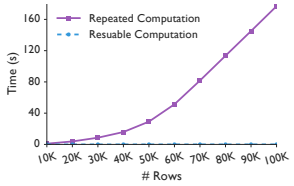
(a) 200k (b) 300k

(c) 400k (d) 500k

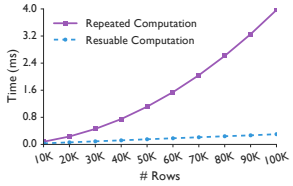
Figure 14: Screenshot of Calc memory usage.

A	B	C
1	=SUM(A1:A1)	=A1
2	=SUM(A1:A2)	=A2+C1
3	=SUM(A1:A3)	=A3+C2
4	=SUM(A1:A4)	=A4+C3
• • •	• • •	• • •
n	=SUM(A1:An)	=An+C(n-1)

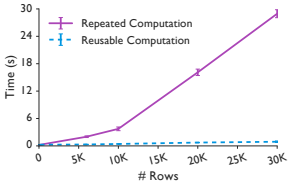
(a) Sample Data



(b) Excel

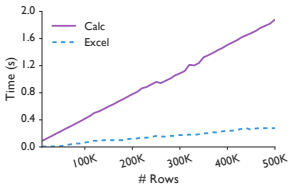


(c) Libre

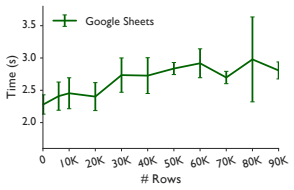


(d) GS

Figure 15: Expressing the same computation in two different ways (repeating the computation vs. reusing as much as possible) leads to substantial differences in runtime complexity (quadratic vs. linear), indicating no sharing of computation.



(a) Excel and Calc



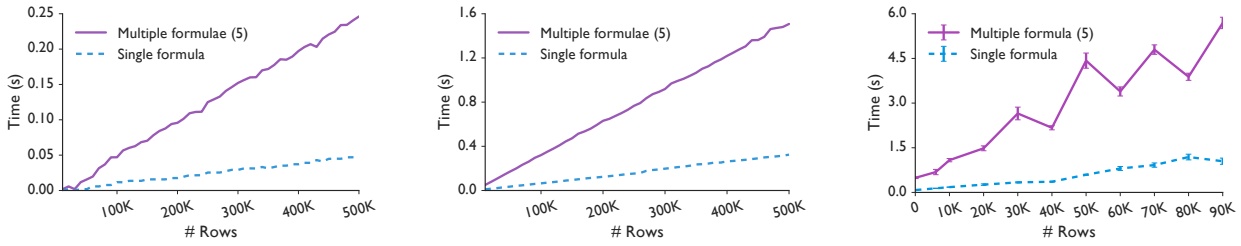
(b) Google Sheets

Figure 16: All three systems recompute the results of a COUNTIF formula from scratch after a single cell update.

the SUM formula (see column *B* in Figure 15a) which calculates the sum over the entire input range. Another efficient way, which we call the *reusable* computation approach, is by adding the already computed cumulative sum up to row $i-1$ with the value of cell *Ai*. In a shared computation scenario, we expect the time complexity of both approaches (computing the same final result) to scale linearly with the number of formulae (see column *C* in Figure 15a).

Observations. Figure 15 shows that, for all systems, repeated computation takes quadratic time as the number of rows increases. The quadratic time can be attributed to the increasing number of cell references. As i increases, the total number of cell references of the repeated computation approach increases in a quadratic fashion, *i.e.*, $\sum_{i=1}^{m} i = O(m^2)$. For $m = 10k$, that leads to 50 million references. We speculate that the way spreadsheets perform computation is to *individually look up all cells mentioned in the formula independently without any regard for sharing sub-expressions or accesses across formulae*. Therefore, this cell-by-cell reference model severely impacts the formula computation performance. On the other hand, reusable computation, where the number of cell references increases linearly with the number of formulae, exhibits an $O(m)$. This approach mimics a shared computation scenario: a collection of formulae whose input range overlap can share computation to optimize performance. However, it appears current systems do not employ any such optimizations.

Takeaway: *Spreadsheet systems do not employ sharing of computation for formulae with overlapping regions.*



(a) Excel (b) Calc (c) Google Sheets

Figure 17: All three systems redundantly compute duplicate instances of a COUNTIF formula instead of reusing the previously computed result, causing the execution time to increase linearly with the number of duplicates.

5.4 Eliminating Redundant Computation

Our previous experiment revealed a setting where shared computation was not used by spreadsheet systems; the systems were not able to detect sharing opportunities and use them to reduce computation. We wanted to test an extreme (and very obvious to detect) version of shared computation—one where the formulae being computed were *exactly* the same.

For this experiment, we executed five instances of the same COUNTIF formula "COUNTIF($J2:Jm$,'1')" on Value-only datasets of varying row count, m, by programmatically inserting each instance within the spreadsheet. An optimal approach for this such computation is to reuse the result of the first formula instance to compute the results of the subsequent instances. Therefore, the optimal approach is expected take nearly constant time.

Observations. The result shows that the completion time of five formula instances takes $\approx 5X$ more time than a single instance of the COUNTIF formula (see Figure 17). We see similar results for Calc and Google Sheets. Therefore, spreadsheet systems do not test for formula equality (*e.g.*, by hashing the formulae and identifying matches) and reuse the computation. We ran the same experiment for VLOOKUP which revealed that no elimination of redundant computation is being performed by any of the three spreadsheet systems. For both COUNTIF and VLOOKUP, we repeated the same experiment for $N = 2, 3, 4$ formula instances which yielded similar results—the computation time scales linearly with the number of formulae instances.

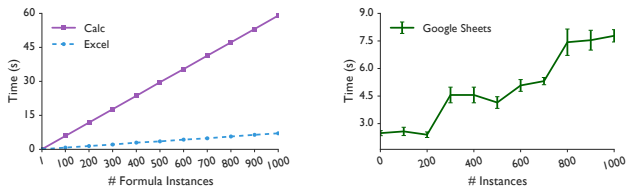
> **Takeaway:** *Spreadsheet systems do not even detect and avoid entirely redundant computation of identical formulae.*

5.5 Incremental Updates

Next, we wanted to see whether spreadsheet formulae can efficiently handle updates to cells that the formulae operate on. One approach can be to materialize the formula result, compute the difference between the old and new value of a cell and then update the results, analogous to incremental view updates. We run this experiment on the following formula "=COUNTIF($J2:Jm$,"1")" with Value-only datasets of varying row sizes. For each dataset, we change the value of the cell $J2$ from 1 to 0 and measure the time for recomputation. If results are materialized or memoized, a formula would require near constant time to recompute after the update of a single cell within the region referenced by the formula.

Observations. Figure 16 shows that the run time for Excel and Calc scales linearly with the number of rows—taking $O(m)$ time instead of $O(1)$: thus these systems recompute the formula from scratch rather than using incremental updates. Google Sheets also does not employ incremental updates the results as the run time varies with the number of the rows; however, the result is quite noisy.

Single vs multiple formulae. To further demonstrate the impact of updating a single cell, we run another experiment where we vary the number of instances of the same formula ($N = 1, 100, 200, \ldots, 1000$) while changing the value of the cell $J2$. We use the $500k$ Value-only dataset for the desktop-based spreadsheets and 90k dataset for Google Sheets.



(a) Excel and Calc (b) Google Sheets

Figure 18: While recomputing a mere 100 instances of a COUNTIF formula following a single cell update, all systems violate the interactivity bound.

Observations. Figure 18 shows that following a single cell update, recalculation time scales linearly with the number of formulae and violates the interactivity bound at 100 COUNTIF formulae. As none of the spreadsheet systems share computation and perform incremental updates, even a single update can cause the spreadsheet to freeze [26].

> **Takeaway:** *None of the spreadsheet systems perform incremental recomputation for small updates, instead recomputing formulae from scratch. The recomputation time increases linearly with the number of formula instances.*

5.6 Discussion

The OOT benchmark reveals that spreadsheet systems do not implement indexes, share computation, eliminate redundant

computation, perform incremental updates, or use compact or workload-aware data layouts to speed up execution. Ultimately, all spreadsheet systems end up leaving formulae uninterpreted, individually looking up the arguments cell-by-cell for the purposes of computation. Small changes end up becoming prohibitively expensive, leading to spreadsheet systems hanging and freezing on small changes.

6 CONCLUSION AND NEXT STEPS

We now summarize the findings from our experiments and discuss ways to improve spreadsheet systems, informed by our experimental results and previous work. We found that even though spreadsheet systems operate on in-memory data, they remain interactive for only a few operations through bespoke optimizations, *e.g.*, visible window prioritization during open for Google Sheets, binary search during lookup for Excel. As shown in Table 2, all three systems achieve interactive response times only on small datasets that are a fraction of their advertised scalability limits. On the other hand, relational databases, despite operating on disk-resident data, can achieve much better performance for datasets of similar scale through various optimizations. The OOT benchmark confirms that spreadsheet systems do not perform any such optimizations.

Database-style Optimizations. Introducing "database-style" optimizations within spreadsheet systems has the potential to substantially improve the interactivity of spreadsheets. However, there are some challenges as well.

Indexing and Data Layout. As we saw in Section 5.1, there are many settings where indexing could be valuable. We could use existing formulae as a workload to identify columns that should be indexed. Indexing may be counterproductive for spreadsheets where the raw data is being heavily edited, and may be more useful during analysis. Indexing could also be valuable for find-and-replace operations, but this would require indexing strings across cells as opposed to just a column. Note that indexing may be problematic if it explicitly uses or encodes the row or column number, because a single change (adding a row) can lead to an update of the entire index—but recent work has proposed a solution [7]. As Figure 13 indicates, there is a lot to be done in developing a more compact and workload-aware representation for spreadsheets in-memory; understanding the trade-off between computational benefits of precomputed dependency chains and memory consumption would be valuable. Compact representations of the data itself would benefit not just memory consumption but also computation. Finally, the structure of the data [9] and the formulae could be together used to reorganize the data to optimize data access.

Shared computation. It is clear from Section 5.3 that spreadsheet systems need to go beyond cell-by-cell retrieval and execution of formulae, actively identifying shared computation opportunities. These opportunities can be identified when a formula is added (*e.g.*, hashing subexpressions to see if it is already present in the sheet in an evaluated form), or in the background asynchronously. A simpler version is to wait until a change triggers computation of a collection of formulae, and then compute these formulae via an intelligent schedule to maximize cache locality [8].

Incremental updates. This optimization, whose absence we confirmed in Section 5.5, is perhaps the easiest to implement for spreadsheet systems. For many aggregation operations, the results can be recomputed using the current aggregate and the "delta", without requiring recomputation. For cases such as AVGIF (*i.e.*, compute average of cells satisfying a condition) we need to maintain the count of cells in addition to the average. An interesting question is to see if incremental updates can be used for other operations beyond aggregation, such as VLOOKUP.

Detecting what needs recomputation. Section 4 demonstrated that the Formula-value datasets often performed much worse than Value-only datasets for all three spreadsheet systems due to poor detection of what needs recomputation on changes, *e.g.*, sort or filter. Identifying clear rules to determine whether a formula needs recomputation would be the first challenge. For example, when sorting an entire spreadsheet by row, any formula with relative columnar references, *e.g.*, "C1 = A1 + B1", are unaffected, while those with absolute references, *e.g.*, "C1 = A1 + B1", require recomputation.

Additional Optimizations. There are other potential optimizations from the literature that slightly change spreadsheet semantics for increased interactivity. For example, spreadsheet systems remain unresponsive during computation and return control after completion. One can employ asynchrony to increase interactivity, covering up in-progress computation with a progress bar [8]. Asynchrony can be adapted to other operations like open and sort, targeting the visible window. For example, lazy computation is already partially employed in Google Sheets to load or open data on demand. Prior work has also proposed asynchronous sorting of spreadsheet data via *dynamic reordering* [38] to support large spreadsheet datasets [37], allowing users to operate on the data before it is completely sorted, while prioritizing visible areas. We can also use a database backend for efficient execution by translating formulae into SQL queries [7, 12, 24], *e.g.*, a join instead of a collection of VLOOKUPs. Efficient execution can also happen via *approximation*, *e.g.*, depicting confidence intervals for formulae currently under progress, as in online aggregation, as well as other approximate query processing schemes [14, 20], allowing users to terminate their execution early if needed.

Discussion with Development Teams. After the first version of this paper was posted online, we were approached by the Calc and Google Sheets development teams who expressed

an interest in our takeaways. We initiated conversations with both teams, and report some initial feedback below.

Data Layout. The Calc team confirmed the use of a columnar data layout (Section 5.1.2): Calc has a columnar MDDS data store for optimized data access, with SSE optimization [11] for columnar SUMS. The Calc team acknowledged the trade-off between performance and storage (Section 5.2.3), opting to prioritize precomputation of dependency graphs and calculation chains over memory consumption.

Optimizing Computation. The Calc team confirmed the lack of sharing and redundancy identification (Section 5.3), noting that these optimizations can benefit computationally heavy spreadsheets. The Google Sheets team had similar observations, noting that such sheets often come from enterprise clients. Both teams expressed reservations with incremental updates, specifically, precision issues resulting in unpredictable (non-idempotent) results.

Benchmarking and Architecture. The Google Sheets team identified a missing dimension in our benchmark: the addition/deletion of rows/columns, which we will address in future releases. Moreover, Google Sheets pushes some computation to the client-side browser. Our benchmark primarily evaluates server-side performance; future benchmarks should evaluate both. Since some computation happens at the client side, the source code isn't as "closed", and can be profiled on the client-side, *e.g.*, via apps script logger and cloud platform logger [19]. Calc benchmarks performance using the open-source Callgrind Test Suite [22].

Overall, there is a plethora of interesting research directions in making spreadsheet systems more effective at handling large datasets. Our evaluation and the resulting insights can benefit spreadsheet development, and also provide a starting point for database researchers to contribute to the emergent discipline of spreadsheet computation optimization.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable feedback. We also thank Richard Lin for help in re-running experiments for Google Sheets. We acknowledge support from grants IIS-1652750 and IIS-1733878 awarded by the National Science Foundation, grant W911NF-18-1-0335 awarded by the Army, and funds from Adobe, Capital One, Facebook, Google, Siebel Energy Institute, and the Toyota Research Institute. The content is solely the responsibility of the authors and does not necessarily represent the official views of the funding agencies and organizations.

REFERENCES

[1] [n. d.]. Excel functions by category. https://support.office.com/en-ie/article/excel-functions-by-category-5f91f4e9-7b42-46d2-9bd1-63f26a86c0eb. ([n. d.]).

[2] [n. d.]. LibreOffice Online. https://www.libreoffice.org/download/libreoffice-online/. ([n. d.]).

[3] [n. d.]. OpenOffice is dead. Long live LibreOffice. https://www.zdnet.com/article/openoffice-is-dead-long-live-libreoffice/. ([n. d.]).

[4] [n. d.]. Pivot Table. https://en.wikipedia.org/wiki/Pivot_table. ([n. d.]).

[5] [n. d.]. VBA. https://docs.microsoft.com/en-us/office/vba/api/overview/. ([n. d.]).

[6] Airbnb [n. d.]. The inside airbnb dataset. http://insideairbnb.com/get-the-data.html. ([n. d.]).

[7] Mangesh Bendre, Vipul Venkataraman, Xinyan Zhou, Kevin Chang, and Aditya Parameswaran. 2018. Towards a holistic integration of spreadsheets with databases: A scalable storage engine for presentational data management. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, 113–124.

[8] Mangesh Bendre, Tana Wattanawaroon, Kelly Mack, Kevin Chang, and Aditya Parameswaran. 2019. Anti-Freeze for Large and Complex Spreadsheets: Asynchronous Formula Computation. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD '19)*. 1277–1294.

[9] Zhe Chen, Michael Cafarella, Jun Chen, Daniel Prevo, and Junfeng Zhuang. 2013. Senbazuru: a prototype spreadsheet database management system. *Proceedings of the VLDB Endowment* 6, 12 (2013), 1202–1205.

[10] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*. ACM, 143–154.

[11] Intel Corportation. 2001. IA-32 Intel Architecture software developer's manual. *Intel Corportation* 127 (2001).

[12] Jácome Cunha, João Saraiva, and Joost Visser. 2009. From spreadsheets to relational databases and back. In *Proceedings of the 2009 ACM SIGPLAN workshop on Partial evaluation and program manipulation*. 179–188.

[13] Conor Cunningham, César A Galindo-Legaria, and Goetz Graefe. 2004. PIVOT and UNPIVOT: Optimization and Execution Strategies in an RDBMS. In *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*. VLDB Endowment, 998–1009.

[14] Minos N Garofalakis and Phillip B Gibbons. 2001. Approximate Query Processing: Taming the TeraBytes.. In *VLDB*. 343–352.

[15] Google. 2020. G Suite. (2020). Retrieved April 8, 2020 from https://gsuite.google.com/

[16] Google. 2020. Google Apps Script. (2020). Retrieved April 8, 2020 from https://developers.google.com/apps-script

[17] Google. 2020. Google Sheets scale. (2020). Retrieved April 8, 2020 from https://developers.google.com/drive/answer/37603

[18] Google. 2020. Layout thrashing. (2020). Retrieved April 8, 2020 from https://developers.google.com/web/fundamentals/performance/rendering/avoid-large-complex-layouts-and-layout-thrashing

[19] Google. 2020. Logging GAS. (2020). Retrieved April 8, 2020 from https://developers.google.com/apps-script/guides/logging

[20] Joseph M Hellerstein, Peter J Haas, and Helen J Wang. 1997. Online aggregation. In *Acm Sigmod Record*, Vol. 26. ACM, 171–182.

[21] Scott T Leutenegger and Daniel Dias. 1993. A modeling study of the TPC-C benchmark. *ACM Sigmod Record* 22, 2 (1993), 22–31.

[22] LibreOffice. 2020. Callfrind: Calc Profiler. (2020). Retrieved April 8, 2020 from https://perf.libreoffice.org

[23] LibreOffice. 2020. LibreOffice Basic. (2020). Retrieved April 8, 2020 from https://documentation.libreoffice.org/en/english-documentation/macro/

[24] Bin Liu and HV Jagadish. 2009. A spreadsheet algebra for a direct data manipulation query interface. In *2009 IEEE 25th International Conference on Data Engineering*. IEEE, 417–428.

[25] Zhicheng Liu and Jeffrey Heer. 2014. The effects of interactive latency on exploratory visual analysis. *IEEE transactions on visualization and computer graphics* 20, 12 (2014), 2122–2131.

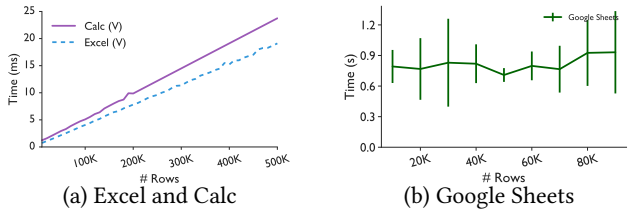
[26] Kelly Mack, John Lee, Kevin Chang, Karrie Karahalios, and Aditya Parameswaran. 2018. Characterizing scalability issues in spreadsheet software using online forums. In *Extended Abstracts of the 2018 CHI Conference on Human Factors in Computing Systems*. ACM, CS04.
[27] MDDS. 2020. Multi-dimensional data structures. (2020). Retrieved April 8, 2020 from https://gitlab.com/mdds/mdds
[28] Microsoft. 2020. Excel calculation engine. (2020). Retrieved April 8, 2020 from https://docs.microsoft.com/en-us/office/client-developer/excel/excel-recalculation/
[29] Microsoft. 2020. Excel functions list. (2020). Retrieved April 8, 2020 from https://support.office.com/en-us/article/Excel-functions-alphabetical-b3944572-255d-4efb-bb96-c6d90033e188
[30] Microsoft. 2020. Excel limit. (2020). https://support.office.com/en-us/article/excel-specifications-and-limits-1672b34d-7043-467e-8e27-269d656771c3
[31] Microsoft. 2020. Excel threading. (2020). Retrieved April 8, 2020 from https://docs.microsoft.com/en-us/office/client-developer/excel/multithreaded-recalculation-in-excel
[32] Microsoft. 2020. Excel user statistics. (2020). Retrieved April 8, 2020 from https://enterprise.microsoft.com/en-gb/articles/roles/finance-leader/how-finance-leaders-can-drive-performance/
[33] Microsoft. 2020. Office 365. (2020). Retrieved April 8, 2020 from https://www.office.com/
[34] Bonnie A Nardi and James R Miller. 1990. *The spreadsheet interface: A basis for end user programming*. Hewlett-Packard Laboratories.
[35] Aditya Parameswaran. 2019. Enabling Data Science for the Majority. *Proceedings of the VLDB Endowment* 12, 12, 2309–2322.
[36] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J Abadi, David J DeWitt, Samuel Madden, and Michael Stonebraker. 2009. A comparison of approaches to large-scale data analysis. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*. ACM, 165–178.
[37] Vijayshankar Raman et al. 1999. Scalable Spreadsheets for Interactive Data Analysis. In *ACM SIGMOD Workshop on DMKD*.
[38] Vijayshankar Raman, Bhaskaran Raman, and Joseph M Hellerstein. 2000. Online dynamic reordering. *The VLDB Journal* 9, 3 (2000), 247–260.
[39] Google user statistics. 2020. Excel vs. Google Sheets usage—nature and numbers. (2020). https://medium.com/grid-spreadsheets-run-the-world/excel-vs-google-sheets-usage-nature-and-numbers-9dfa5d1cadbd/
[40] Wikipedia. 2020. LibreOffice Calc. (2020). Retrieved April 8, 2020 from https://en.wikipedia.org/wiki/LibreOffice_Calc/
[41] Wikipedia. 2020. List of spreadsheet softwares. (2020). Retrieved April 8, 2020 from https://en.wikipedia.org/wiki/List_of_spreadsheet_software
[42] Wikipedia. 2020. OpenDocument format. (2020). Retrieved April 8, 2020 from https://en.wikipedia.org/wiki/OpenDocument
[43] Wikipedia. 2020. XLSM file. (2020). Retrieved April 8, 2020 from https://en.wikipedia.org/wiki/List_of_Microsoft_Office_filename_extensions
[44] Justin Zobel and Alistair Moffat. 2006. Inverted files for text search engines. *ACM computing surveys (CSUR)* 38, 2 (2006), 6.

A ADDITIONAL EXPERIMENTAL RESULTS

We now present results from benchmarking another dataset. The setup for all of the experiments is the same as that in Section 4 and 5. The only difference is that, we use the *Inside Airbnb* dataset [6], a dataset of Airbnb listings across different US cities. This dataset contains 109,000 rows and 15 columns. We created different synthetic versions of the dataset following the process explained in Section 3. Note that the Airbnb dataset is a Value-only dataset, since it doesn't contain any formulae.

A.1 BCT Benchmark

We first discuss the results from the BCT benchmarking experiments on the Airbnb dataset.



(a) Excel and Calc (b) Google Sheets

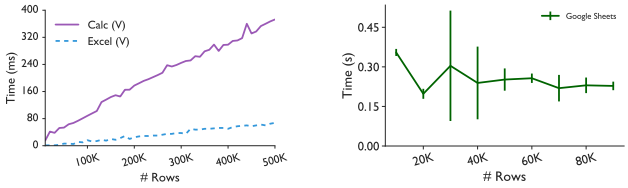
Figure 19: *Open* in Excel, Calc is slow; it is faster on Google Sheets due to lazy loading of data not in the user window.

A.1.1 Data Load Operations. The open operation loads an existing spreadsheet from disk to memory. We document the time to open a spreadsheet, while varying row sizes m, where $m = 150, 6k, 10k, 20k, \ldots, 500k$. Figure 19a shows the time taken by Excel and Calc to open datasets of different sizes. Like the weather dataset results in Figure 2 (see Section 4.1), for the Airbnb dataset, the time complexity appears to be linear in the size, *i.e.*, number of rows, of the spreadsheet for desktop-based systems (Figure 19a) and independent of the size of the spreadsheet for Google Sheets (Figure 19b). Both *Excel and Calc violate the interactivity bound while opening as few as 6000 and 150 rows* of the Airbnb Value-only dataset, respectively, while Google Sheets breaks the interactivity time bound of 500ms while ing even a screenful of data.

A.1.2 Update Operations. We now discuss the benchmarking results for update operations.

Conditional Formatting. Recall that the conditional formatting operation takes a data range and a conditional expression as input and updates the style of the cells within the range that satisfy the condition. For this experiment, we measure the time to execute an operation to color cells in a column green if the value is greater than 1. Figure 20 shows that while Excel and Calc exhibit a linear trend, Google Sheets takes almost the same time to complete the operation irrespective of the size of the dataset. All three spreadsheet systems complete the operation within an interactive time bound, with Excel being the fastest. We observed similar trends for the weather dataset (see Figure 4 in Section 4.2).

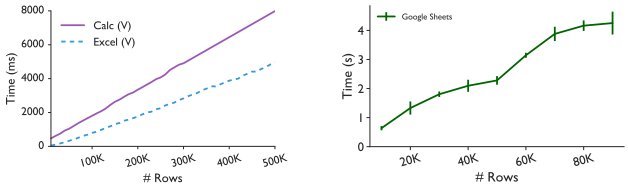
Sort. The sort operation takes one or more column references, the sort order, and a range of cells and reorganizes the range of cells in the order of the referenced columns. In our experiments, we sort the data by a single attribute—column A of unique integer values. Figure 21 shows the run time for sorting the Airbnb Value-only datasets across spreadsheet systems. Similar to the weather dataset in Figure 5



(a) Excel and Calc (b) Google Sheets

Figure 20: All three spreadsheet systems satisfy the interactivity bound for the conditional formatting operation. Google Sheets is faster for Value-only due to formatting cells in a lazy fashion.

(see Section 4.2), we a linear trend for sorting for all three systems.

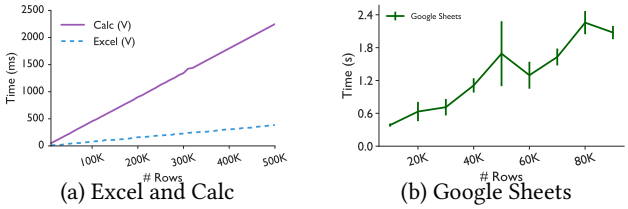


(a) Excel and Calc (b) Google Sheets

Figure 21: All three systems violate the interactivity bound for Sort.

A.1.3 Query Operations. We now discuss the results for the four categories of query operations: select, report, aggregate, and lookup.

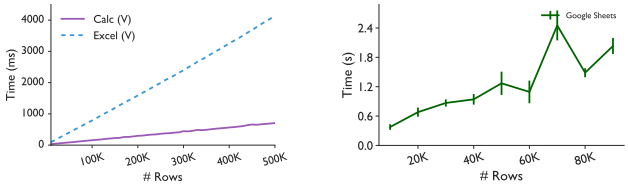
Select (Filter). The filter operation hides rows that do not satisfy the filtering condition. In our experiments, any row for which city ≠ *LA* is hidden.



(a) Excel and Calc (b) Google Sheets

Figure 22: For *Filter*, Google Sheets is slower than the other two.

As can be seen in Figure 22, all systems exhibit a linear trend. We observed a similar trend for the weather dataset (see Figure 6 in Section 4.3). Excel completes the operation within 500ms even with 500k row Airbnb dataset. However, Calc and Google Sheets violate the interactivity bound at 200k and 20k rows, respectively.



(a) Excel and Calc (b) Google Sheets

Figure 23: Calc is faster than the other two for Pivot Tables

Report (Pivot Table). The pivot table operation creates a table with summary statistics. In this experiment, we create a pivot table that computes and shows the *sum of prices* per city in a new worksheet. Figure 23 shows the results. Similar to the weather dataset (see Figure 7 in Section 4.3), the observed complexity is linear for all three systems with Calc outperforming both Excel and Google Sheets. While Calc remains interactive up to $330k$ rows, the other two systems violate the interactivity bound at 50k and 20k rows, respectively.

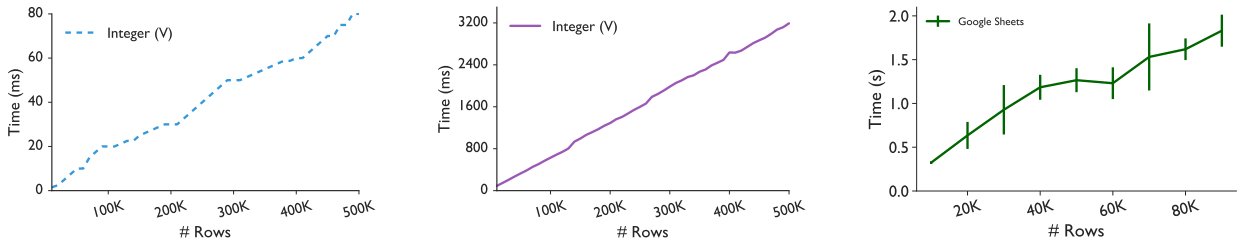
Aggregate operation. An aggregate formula, *e.g.*, COUNT, takes a range as input and then computes the aggregate of the values within that range. The conditional variant of an aggregate formula, *e.g.*, COUNTIF, takes an additional condition as input. We use the following formula in our experiment: "=COUNTIF($K2 : Km, 1$)", where $m = 150, 6k, 10k, 20k, \ldots, 500k$. Figure 24 shows the results of the COUNTIF formula execution. For all Airbnb datasets, Excel and Calc complete the operation in less than 500ms. However, Excel completes the operation earlier than Calc. The execution time is even worse for Google Sheets, which often takes minutes while violating the interactivity bound at 10k rows of data. We observed similar trends for the weather dataset as well (see Figure 8 in Section 4.3).

Look Up. These operations look up a specific value X within a given input range and returns the value of another cell within the same row where X was found. In our experiment, we perform a VLOOKUP on column A searching for an integer X and return the corresponding US state for the row i such that $A_i = X$, where $X = 200000$. Figure 25 shows the results which is similar to that of the weather dataset (see Figure 9): the execution time of VLOOKUP varies significantly across systems. When the search parameter is set to *False*, *i.e.*, exact match, Excel terminates execution after finding the value at the $200k$-th row. For datasets with $N < 200k$, Excel ends up scanning the entire data as no matching value is found. In both cases, Excel completes the lookup operation in less than 500ms. When the search parameter is set to *True*, *i.e.*, approximate match, Excel exhibits almost constant run time. Calc ends up scanning the entire dataset even after finding the value being looked up, as does Google Sheets.

A.2 OOT Benchmark

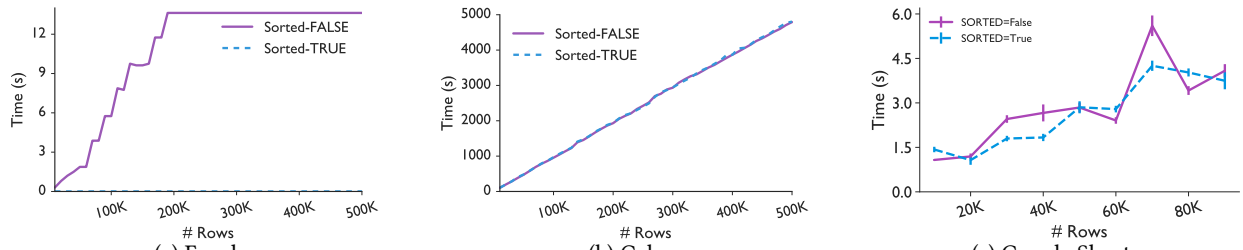
Here we present the results from the OOT benchmark for the Airbnb dataset that investigates presence of database-style optimizations.

A.2.1 Indexing. To see if spreadsheet systems perform inverted indexing, we run the following experiments: (a) find a predefined string and replace it with another, and (b) search for a nonexistent value. For all three systems, a linear trend emerges for find-and-replace operations (see Figure 26). Even when searching a non-existent value, the search time increases linearly with the size of the data. We observed a



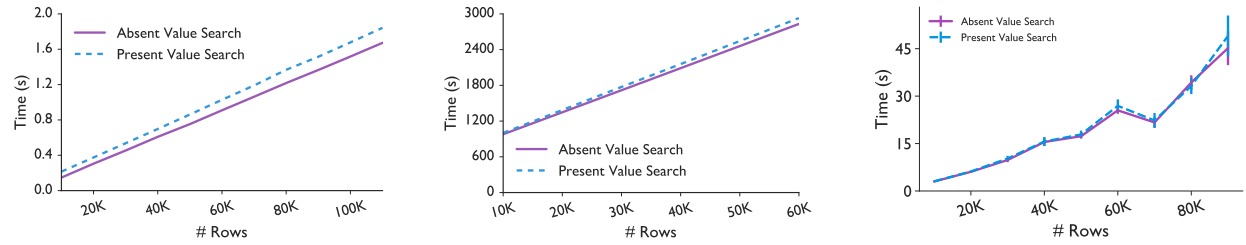
(a) Excel (b) Calc (c) Google Sheets

Figure 24: COUNTIF **is extremely fast in Excel compared to Calc and Google Sheets.**



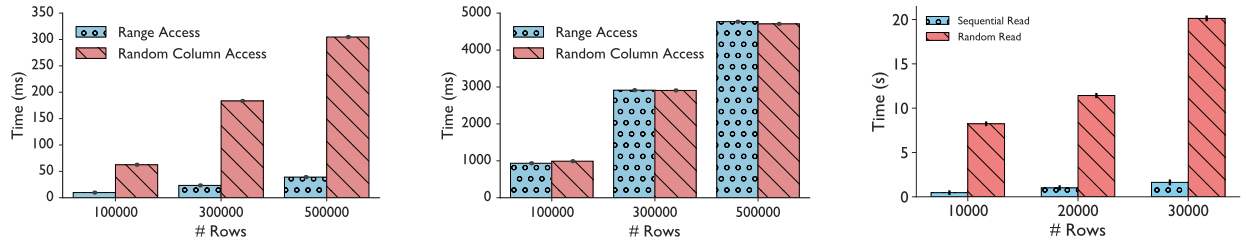
(a) Excel (b) Calc (c) Google Sheets

Figure 25: For VLOOKUP**, while Excel terminates after finding a matching value, Calc and Google Sheets continue to scan the entire data. Excel optimizes approximate search (Sorted=True) via an efficient searching algorithm,** *e.g.*, **binary search.**



(a) Excel (b) Calc (c) Google Sheets

Figure 26: A linear trend for ***Find and Replace*** **indicates the absence of an index.**



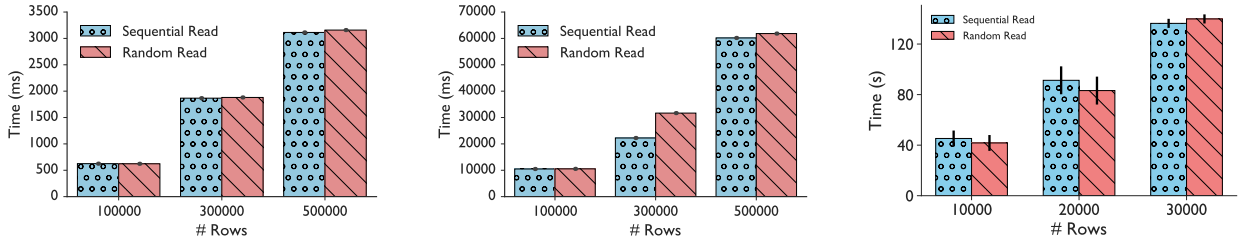
(a) Excel (b) Calc (c) Google Sheets

Figure 27: Excel and Google Sheets seems to employ a row-oriented data layout.

similar trend for the weather dataset (see Figure 10 in Section 5.1). However, the difference in time between searching for an existing value and a non-existent value is not as prominent for the Airbnb dataset.

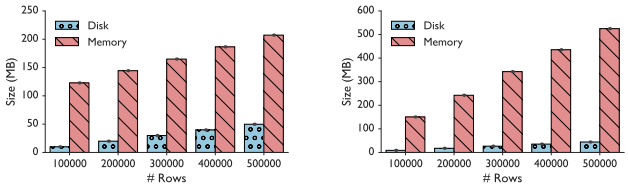
A.2.2 Efficient and Intelligent Data Layout. Similar to the experiments in Section 5.1.2, we run three different experiments comparing: a) range and columnar data access, b) sequential and random data access, and c) disk and memory consumption.

Range vs. column access. For range access, we scan a spreadsheet range and count the total number of cells in the range. For random column access, we randomly select an entire column, count the number of cells in that column, and then add all the counts. If a spreadsheet system employs a row-oriented layout, we expect range access to be faster than column access. The vice-versa is true if a column-oriented layout is used. According to Figure 27, for both Excel and



(a) Excel (b) Calc (c) Google Sheets

Figure 28: While Calc employs a columnar data layout, for Excel and Google Sheets, sequential and random access of a column takes roughly the same time confirming that these systems employ a row-oriented data layout.



(a) Excel (b) Calc

Figure 29: For both the desktop-based systems, the in memory representation of Value-only datasets consume more space compared to the disk.

Google Sheets, range access is orders of magnitude faster than random column access. Therefore, we speculate that these systems employ a row-oriented data layout. On the other hand, for Calc both experiments take the same time indicating the absence of a row-oriented layout. We observed similar outcomes for the weather dataset (see Figure 11 in Section 5.1.2).

Sequential vs. random access. For sequential data access, we scan a spreadsheet column A from beginning to end while accessing the values of each cell. For random data access, we randomly select a row and access the cell corresponding to column A within that row. As seen in Figure 28, the time for sequential and random access is very similar for Excel and Google Sheets further verifying a row-oriented data layout. On the other hand, sequential access is faster in Calc compared to random access, indicating the existence of employs a columnar data layout. Again, the improvement in sequential access is not proportional to the number of rows accessed. We observed similar outcomes for the weather dataset (see Figure 11 in Section 5.1.2).

Memory and Disk Consumption. In this experiment, we compare the memory and disk usage of desktop-based spreadsheet systems for the Value-only Airbnb datasets. We use five datasets with $100k$, $200k$, $300k$, $400k$, and $500k$ rows, respectively. As shown in Figure 29, Excel's in-memory representation is up to 7× that of the datasets on disk, while Calc's representation is up to 13×. We observed similar outcomes for Value-only weather datasets also (see Figure 13 in Section 5.2.3).

A.2.3 Shared Computation. Similar to the experiment in Section 5.3, we compare two different cumulative sum computation approaches: repeated and reusable computation. The reusable computation approach shares computation results between formulae with overlapping regions. The repeated computation approach doesn't perform any such optimization. Figure 30 shows that for all three systems, the repeated computation approach takes quadratic time while the reusable computation approach is linear with the number of rows. Similar to the weather dataset experiment (see Figure 15 in Section 5.3), the results confirm that the current spreadsheet systems do not share computation when executing multiple formulae with overlapping regions.

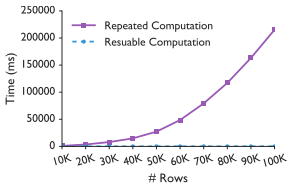
A.2.4 Eliminating Redundant Computation. For this experiment, we executed five instances of the same COUNTIF formula "COUNTIF($J2 : Jm$,'1')" on Value-only datasets with varying row count, m, by programmatically inserting each instance within the spreadsheet. The result shows that, for all three systems, the completion time of five formula instances takes $\approx 5X$ more time than a single instance of the COUNTIF formula (see Figure 31). We observed similar outcomes for the weather dataset (see Figure 17 in Section 5.4) and again confirm that these systems do not eliminate redundant computation.

A.2.5 Incremental Updates. We run this experiment on the following formula "=COUNTIF($J2 : Jm$,"1")" with Value-only datasets of varying row sizes. For each dataset, we change the value of the cell $J2$ from 1 to 0 and measure the time for recomputation. As seen in Figure 32, the run time for all three systems scales linearly with the number of rows—taking $O(m)$ time instead of $O(1)$: thus these systems recompute the formula from scratch rather than using incremental view updates. We observed similar outcomes for weather dataset with Excel (Figure 16a) and Calc (Figure 16b). However, the results for Google Sheets on Airbnb dataset is quite smooth compared to weather dataset (Figure 16c).

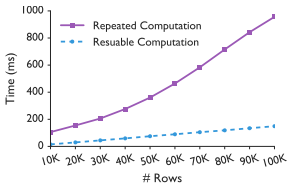
Single vs multiple formulae. To further demonstrate the impact of updating a single cell value on formulae computation, we run another experiment where we vary the number of instances of the same formula ($N = 1, 100, 200, \ldots, 1000$) while changing the value of the cell $J2$. For this experiment,

A	B	C
1	=SUM(A1:A1)	=A1
2	=SUM(A1:A2)	=A2+C1
3	=SUM(A1:A3)	=A3+C2
4	=SUM(A1:A4)	=A4+C3
• • •	• • •	• • •
n	=SUM(A1:An)	=An+C(n-1)

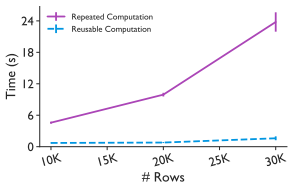
(a) Sample Data



(b) Excel

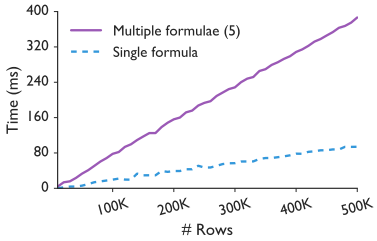


(c) Libre

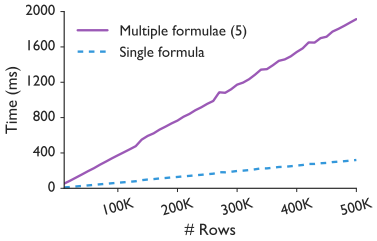


(d) GS

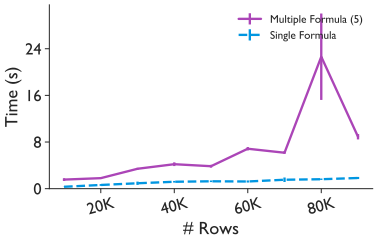
Figure 30: Expressing the same computation in two different ways (repeating the computation vs. reusing as much as possible) leads to substantial differences in runtime complexity (quadratic vs. linear), indicating no sharing of computation.



(a) Excel

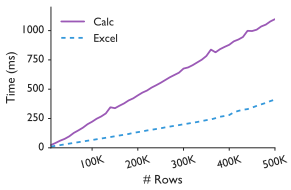


(b) Calc

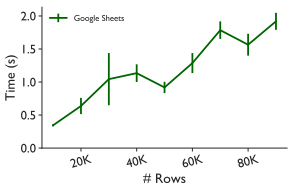


(c) Google Sheets

Figure 31: All three systems redundantly compute duplicate instances of a COUNTIF **formula instead of reusing the previously computed result, causing the execution time to increase linearly with the number of duplicates.**

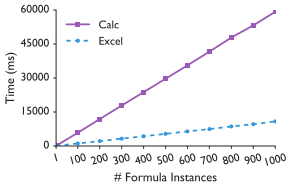


(a) Excel and Calc

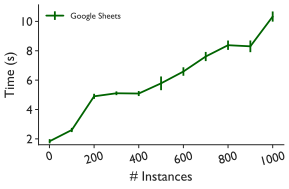


(b) Google Sheets

Figure 32: All three systems recompute the results of a COUNTIF **formula from scratch after a single cell update.**



(a) Excel and Calc



(b) Google Sheets

Figure 33: While recomputing a mere 100 instances of a COUNTIF **formula following a single cell update, all systems violate the interactivity bound.**

we use the 500*k* Value-only dataset for the desktop-based spreadsheets and 90k Value-only dataset for Google Sheets. As shown in Figure 33, following a single cell update, recalculation time scales linearly with the number of formulae and violates the interactivity bound at 100 COUNTIF formulae. We observed similar outcomes for the weather dataset (see Figure 18 in Section 5.5).