Benchmarking Spreadsheet Systems

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

Kelly Mack*
knmack3@uw.edu
University of Washington

Mangesh Bendre*
mbendre@visa.com
VISA Research

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

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

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

ABSTRACT

Spreadsheet systems are used for storing and analyzing data across virtually every domain by programmers and non-programmers alike. While spreadsheet systems have continued to support storage and analysis of increasingly large scale datasets, they are prone to hanging and freezing while performing computations even on much smaller datasets. We present a benchmarking study that evaluates and compares the performance of three popular spreadsheet 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 documented scalability limits. We further evaluate whether spreadsheet systems adopt optimization techniques from the database community 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.

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 [32]. Recent estimates from Microsoft peg spreadsheet use at about $\frac{1}{10}^{\text{th}}$ of the world's population. Spreadsheets systems now support increasingly larger datasets. For example, Microsoft Excel supports more than 10s of billions of cells within a spreadsheet [3]. Even web-based Google Sheets now supports five million cells [9], 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 the size limits posed by current spreadsheet systems. Anecdotes from a recent paper report that computation on spreadsheets with as few as 20,000 rows can lead to hanging and freezing behavior [32]. And importing a spreadsheet of 100,000 rows in Excel (10% of the scalability limit of one million rows) can take over 10 minutes [21].

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? And 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 scale 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 impossible for us to compare the internals of popular spreadsheet systems such as Microsoft Excel and Google Sheets, since they are closed-source. Moreover, 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 a variety of problems, e.g., transaction processing [29], large-scale data analysis [35], and cloud computing [24].

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 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)*. Both of these benchmarks are open-sourced[1] and can be used to test future spreadsheet systems.

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

[1]Link omitted for anonymity.

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 on the sheet—based on their expected time complexity, and evaluate the relative performance of the three spreadsheet systems on a range of data sizes. Our goal is to understand the impact of the type of the 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 [31].

Optimization Opportunities Testing (OOT). As explained earlier, spreadsheet systems have continued to increase their scalability limits over the past few decades [3, 9]. Industry and academic 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 computation in the form of formulae embedded in the spreadsheet. 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). This leads to additional complications. 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 variation in the load on the server where the operation is being performed.

3. Coverage. Spreadsheet systems support a wide variety of operations that make it difficult to evaluate each operation individually[2]. To ensure that we have a thorough coverage of all types of operations, 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 of the interesting takeaways from our evaluation:

- *Spreadsheets are not interactive for many standard operations, even for as few as* 50*k 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.
- *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 relational database-style optimizations. Some egregious examples include the fact that

[2]Counting the operations in this list of Excel functions yields around 400 operations: https://support.office.com/en-us/article/Excel-functions-alphabetical-b3944572-255d-4efb-bb96-c6d90033e188.

(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 [16], is the most popular desktop-based spreadsheet system, boasting about 700M registered users [10]. Google Sheets, part of G suite [7], is the most popular web-based spreadsheet system, with users numbering in the 100s of millions [5]. Calc is an open-source spreadsheet system used by two major open-source office software suites, Apache OpenOffice and LibreOffice [15]. 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 [33]. 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 $B1$ 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 [22, 32]. Apart from basic arithmetic and mathematical formulae, spreadsheet systems also provide built-in formulae for common finance and statistics functions [2], string manipulation operations, as well as GUI-based data summarization, *e.g.*, Pivot Table [19], 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 [3]. The Windows version of Excel supports programming through Microsoft's Visual Basic for Applications (VBA), a dialect of Visual Basic [20]. 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 [18]. Calc can support up to one million rows in a spreadsheet [13]. Calc supports most of the basic functionalities provided by Excel. Calc also supports programming through Calc Basic [12], 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 [14].

Google Sheets. Google Sheets is part of a web-based software office suite, G Suite, offered within Google Drive [7]. 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,

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	Simple	Add or Sub	Value	Value	$O(1)$
		Now()	×	Value	$O(1)$
	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	$O(m_x n_x m_y n_y)$

i.e., five million cells per spreadsheet [9]. Google Sheets also supports programming through Google Apps Script where users can write custom functions and macros in JavaScript [8]. Unlike desktop-based systems, Google Sheets supports collaboration, for example, simultaneous editing of spreadsheets.

3 BENCHMARK SETUP

Now that we have completed an overview of spreadsheets, we are ready to present details of our benchmark. We first 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. Any categories excluded from our benchmarking experiments are highlighted in Gray. Here, we briefly explain the high level categories. We provide a detailed explanation of the three aforementioned dimensions of each operation being benchmarked in Section 4.

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 five sub-categories: simple, select, report, aggregate, and lookup. As the formulae corresponding to the *simple* sub-category operate on a constant number of inputs, the expected complexity is $O(1)$. Therefore, we exclude these from the benchmarking experiments.

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")". So the value of each of those cells is either 0 or 1. We selected a real dataset to ensure that (a) the organization of data within the spreadsheet is reflective of actual spreadsheets used in practice, and (b) the ratio of formula and value data types within the spreadsheet is 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, 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.

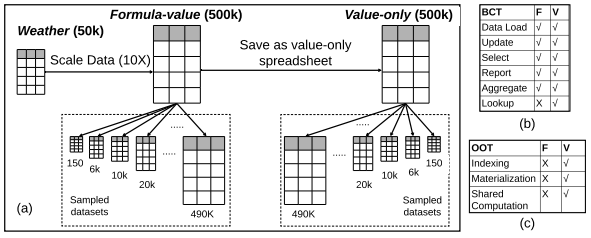


Figure 1: Datasets and benchmarks. (a) Creating synthetic datasets from a real-world spreadsheet by scaling and sampling. Outline of experiments along with datasets used in (b) BCT and (c) OOT benchmark.

Figure 1a shows how the synthetic datasets were created. To understand the effect of scale on different spreadsheet operations, we first created a scaled-up weather dataset called Formula-value (F in short). This dataset has 500k rows—10X

the original dataset—where cell data types can be either formula or value. As explained in Section 1, the embedding of other formulae within a spreadsheet can impact 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* in short), where any formulae within cells were replaced by the corresponding value. To evaluate how formula computation time varies with scale, we created 51 different versions of each of the aforementioned datasets 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 500*k* 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 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 for spreadsheet operations.

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 operating systems. 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 in the three spreadsheet systems were single threaded. Note that Excel 2016 can be configured to support multi-threaded recalculation of formulae [4]. However, the default setting is to evaluate a formula on the main thread of Excel. We now explain how the experiments are implemented in these systems.

For each experiment in Excel, we first created an Excel Macro-Enabled Workbook (*xlsm*) [1] which can execute embedded macros that are programmed in VBA. Unlike Excel, LibreOffice Calc macros, programmed in Calc Basic, can be enabled and executed from the default workbook—OpenSpreadsheet Document (*ods*) [17]. We created the Google App Scripts in G Suite Developer Hub [8]. Given an experiment, all three scripting languages can invoke a spreadsheet formula, *e.g.*, COUNTIF, or operation, *e.g.*, sort, for their respective spreadsheet systems via an API call. We used default library functions of the corresponding scripting languages to measure the execution time of each trial of an experiment. 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-based experiments were uploaded as *xlsx* files in Google Drive and then manually converted to Google Sheets from the Google Drive menu.

For each experiment, we ran ten trials and measured the running time of each trial. 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 limis imposed by Google Apps Script services on some features, like API calls and the number of spreadsheets created and accessed. As a result, for both BCT and OOT 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.

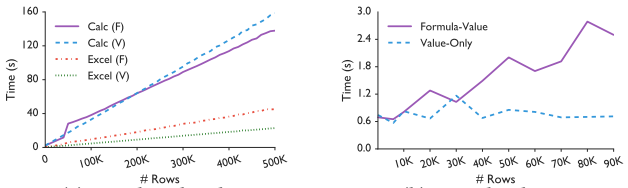
4 BCT BENCHMARK

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 a computational 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 of the categories in the taxonomy in Table 1. Given an operation, we gradually increase the scale of the data 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 [31] and at what data size. For a spreadsheet we denote the number rows and columns 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 both these 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. The open 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 with varying row sizes m, where $m = 150, 6k, 10k, 20k, \ldots, 500k$. As we keep the number of columns fixed in our experiment, the expected complexity for this operation 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.

Observations. Figure 2a shows the time taken by Excel and Calc to open datasets of different size. Recall that the Formula-value datasets have formulae embedded alongside values while the Value-only datasets only contain values. According to Figure 2a, the time complexity for desktop-based spreadsheets appears to be linear in m for all datasets and systems, as expected. 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 time-scale of 500ms to load even a screenful of data. The latency can be attributed to network delay as well as rendering of HTML DOM elements, which can be expensive for web-based spreadsheets [11]. On the other hand, *Excel and Calc violate the interactivity bound while opening only 6000 and 150 row* Value-only datasets, respectively, which is well below their scalability limit of one million rows, and is surprisingly poor.

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. According to the official Excel documentation, when a spreadsheet is opened, Excel first determines a calculation sequence of the embedded formulae and then recalculates the formulae [6]. We speculate that the other spreadsheet systems also perform similar recalculation of embedded formulae. As the number of embedded formulae increases with the size of our Formula-value datasets, *the latency of the open operation is exacerbated for both Excel and Calc, going past the one minute mark at* $40k$ *and* $6k$ *rows, respectively.* Google Sheets performs much better compared to the desktop-based spreadsheets, taking ≈ 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.

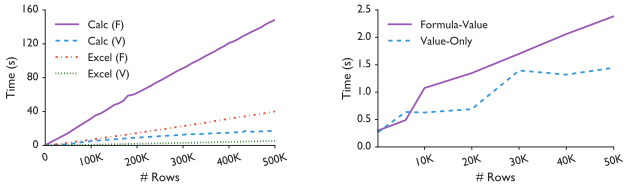
As we can see, there are many opportunities to reduce the latency of data load, including prioritizing returning the first "window" of the spreadsheet (already done by Google Sheets, but not by the other two systems), and prioritizing formula computation for the first "window" (done by none of the systems.) In Section 6, we discuss how spreadsheet systems can optimize data load using ideas like these.

> **Takeaway:** *The desktop-based spreadsheets violate the interactivity bound when opening even small spreadsheets of less than* $10k$ *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: sort and conditional formatting. We present the results for find-and-replace along with the OOT benchmark results in Section 5. The results for copy-paste were found to be similar to find-and-replace, and is therefore excluded.

4.2.1 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



(a) Excel and Calc (b) Google Sheets

Figure 3: Sort on Formula-value is substantially worse than Value-only, thanks to formula recomputation on sort.

entire spreadsheet. In our experiments, we sort the data by a single attribute—column A of unique integer values. The expected complexity of sort for our setting is $O(m \log m)$, where the row count m varies with the size of the dataset.

Observations: Figure 3 shows the run time for sorting for Formula-value and Value-only datasets across spreadsheet systems. Note that 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. 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. 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 presence 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 formulae referred to prior to sorting could possibly be populated with new data, *triggering an often unnecessary recomputation of formulae* [6]. Similar recalculation is likely triggered in both Calc and Google Sheets. Therefore, formula recomputation again contributes to higher latency for an operation. 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. This is therefore wasted computation. In Section 6, we discuss how spreadsheets systems can adopt dynamic reordering strategies to perform sorting in interactive times [37].

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

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

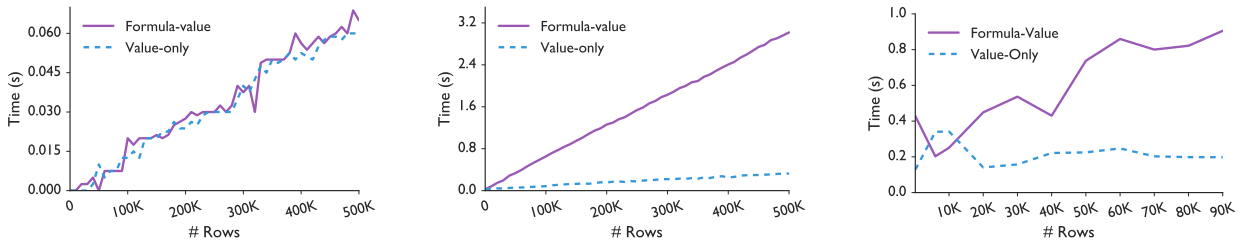
Observations. Figure 4 (we split Excel and Calc into two charts for clarity) shows that although Excel and Calc exhibit a linear trend for Value-only datasets, Google Sheets takes almost the same time to complete the operation irrespective of the size of the dataset. We again speculate that Google Sheets updates the style of the visible cells, doing the rest lazily. Unlike open and sort, all three spreadsheet systems complete the operation within an interactive time bound with Excel being the fastest. 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 column on which the formatting is performed for Formula-value datasets is a formula column. As the value of the cell being formatted is derived from a formula, we speculate that conditional formatting triggers an unnecessary formula recomputation for both Calc and Google Sheets.

> **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* 80k *rows.*

4.3 Query Operations

We now discuss the results for the four categories of query operations: select, report, aggregate, and lookup. As shown in Table 1, both input and output of query operations can be quite diverse. Inputs can vary from a value, condition, range, or any combination thereof, whereas outputs can be any of value, range, list, or aggregate.

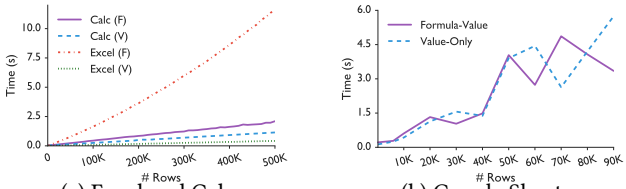
4.3.1 Select (Filter). In this experiment, we filter a given spreadsheet by state *SD* (South Dakota). Filter operations in spreadsheets hide the rows that do not satisfy the filtering condition and is therefore like filters in relational databases. For example, in our experiments, any row for which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



(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.

be linear in m as the filter would require a full scan of the entire dataset.



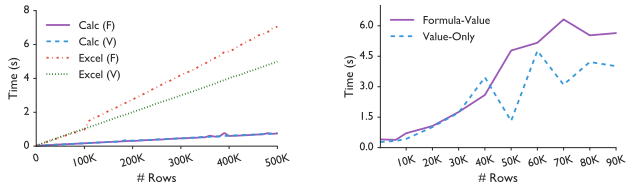
(a) Excel and Calc (b) Google Sheets

Figure 5: *Filter* on Formula-value in Excel does unnecessary recomputation. Google Sheets is slower than the other two.

Observations. As can be seen in Figure 5, 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. On the other hand, *Excel exhibits a super-linear trend for Formula-value datasets and violates the* 500*ms bound at* 40*k rows* (Figure 5a). Filtering likely triggers unnecessary formula recalculation in Excel [6], but why the trend is super-linear is a mystery to us. For Formula-value, the completion time of the filter operation in both Calc and Google Sheets is similar to the Value-only datasets, with interactivity bound violated at datasets with row sizes 120k and 10k, respectively. We speculate that filter operation does not trigger recalculation of 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 is similar to group by queries in databases; it computes summary statistics of groups of data in a tabular for [26]. Users can generate a pivot table on one or more dimension attributes and measure attributes. The pivot table operation scans the entire dataset and creates a summary table in a new or existing worksheet with the results. In this experiment, given a dataset, 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.



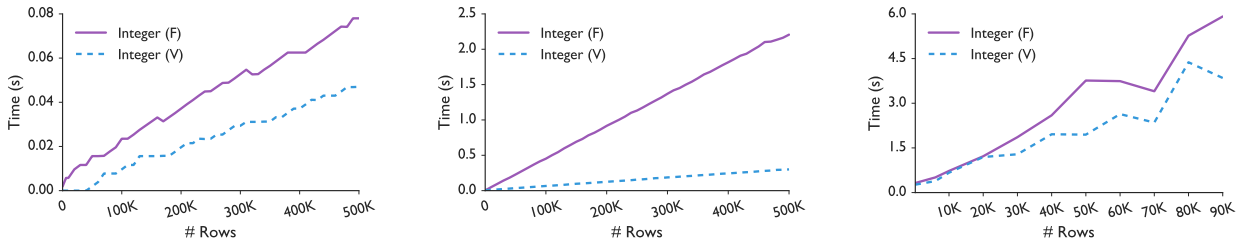
(a) Excel and Calc (b) Google Sheets

Figure 6: Calc is faster than the other two for Pivot Tables

Observations. Figure 6 shows the results of the experiment. For both Formula-value and Value-only datasets the observed complexity is linear, as expected. For Value-only datasets, Calc outperforms (330k rows) both Excel and Google Sheets—the latter two violate the interactivity bound for 50k and 20k rows datasets, respectively. Similar patterns emerge for Formula-value datasets where Calc again outperforms (340k rows) Excel (50k rows) and Google Sheets (10k). Moreover, while pivot table computation in Calc is unaffected by the presence of embedded formulae, both Excel and Google Sheets exhibit higher latency Formula-value. We hypothesize that insertion of a new worksheet in the workbook triggers formula recomputation for Excel and Google Sheets.

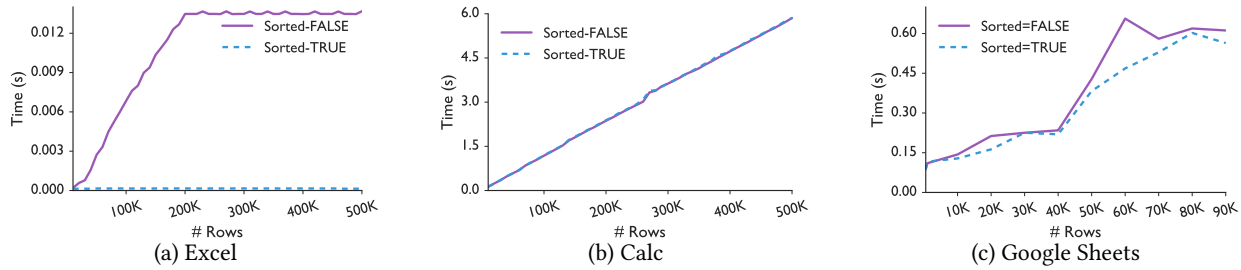
> **Takeaway:** *Calc accommodates* 6× *(Excel) or* 15× *(Google Sheets) the dataset size for Value-only before violating interactivity for pivot tables. Calc avoids a costly formula recomputation for Formula-value, while the other systems do not.*

4.3.3 Aggregate operation. As shown in Table 1, an aggregate formula, *e.g.*, COUNT, takes a spreadsheet range as input and then computes the aggregate of the cell values within that region. The conditional variant of an aggregate formula, *e.g.*, COUNTIF, takes an additional conditional expression 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 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



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

Figure 7: 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 8: 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.

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 7 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 using the VLOOKUP formula. The VLOOKUP formula involved in the experiment operates on the 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 A searching for an integer X and returns the corresponding "state" for the row i such that $A_i = i = X$. 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 three spreadsheet systems, VLOOKUP takes an optional boolean parameter and depending on its value, the formula either performs an approximate match (True) or an exact one (False). In our experiment, apart from varying the input data range, we also varied the aforementioned parameter to see how the formula behaves with different search requirements. The official documentation of the spreadsheet systems require the spreadsheet to be sorted for the approximate match (True) option to work properly. Therefore, we sort the dataset by column A before running our experiments. The worst case expected complexity of the 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. In our experiment, we search for a value of $X = 200000$. 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 8 shows the results: 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 $200k$-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. We speculate that Excel performs additional optimizations, *e.g.*, binary search, to facilitate faster 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. The execution times for Calc seems to indicate that no optimization is performed for VLOOKUP. Calc also ends up scanning the entire dataset even after finding the value being looked up. For Google Sheets, we see the same trend for both search conditions, *i.e.*, *True* and *False*. We again speculate that Google Sheets scans entire dataset even after a matching value is found. For either search conditions, both Calc and Google Sheets violate the interactivity bound for datasets with more than 50k and 60k rows, 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 [34].

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 scalability limits, *i.e.*, one million rows for Excel and Calc and five million cells for Google Sheets, at which the corresponding system starts to violate the interactivity bound of 500ms. To obtain this percentage for each pair of experiment and dataset, we first identify the the number of rows at which the interactivity bound is violated by the spreadsheet systems. We then divide that number of rows by one million for the desktop-based spreadsheets. For Google Sheets, we compute the total number of cells given the number of rows and then divide the quantity by five million. *The results highlight the fact that despite performing computations in memory, except for a handful of cases highlighted in Gray in Table 2, spreadsheet systems fail to provide interactive responses to computational operations for even small scale datasets.* The interactivity of the systems is even worse for spreadsheets with embedded formulae. While spreadsheet systems perform optimizations in the form of visible window prioritization or binary search, these methods are applied to bespoke conditions, resulting in high latency for majority of the operations.

Table 2: Summary of the BCT experiments. For each experiment, we show at what percentage of their documented scalability limits, Excel (E), Calc (C), and Google Sheets (G), violate the interactivity bound. A value of 100% indicates the bound 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

In an attempt to unmask whether spreadsheet systems perform any other optimizations, we perform a number of targeted experiments in Section 5. As we saw previously, latency is exacerbated when the dataset becomes larger. Therefore, we want 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 spreadsheets perform recomputation after an update, and whether 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

In this section, we present the results from the OOT benchmark that investigates if current spreadsheet systems adopt the following classic database-style optimizations for computational operations: indexing, columnar data layout, shared computation, eliminating redundant computation, and incremental updates. All the experiments in this section are on the Value-only dataset as we want to eliminate the effects of other embedded formulae on performance. 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 of the optimizations, *i.e.*, columnar data layout, shared computation, and incremental computation, since they can benefit most from these optimizations, employing COUNTIF, SUM, and VLOOKUP as representatives.

5.1 Indexing

We now explore whether spreadsheets maintain indexes on the columns to facilitate faster computation for the following operations: COUNTIF, VLOOKUP, and find and replace. With indexes, these 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 7, 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 8b) and Google Sheets (see Figure 8c). 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. Our initial goal for benchmarking this operation was to see whether spreadsheet systems perform inverted indexing, a popular indexing mechanism employed by search engines for efficient information retrieval [38]. As shown in Table 1, find-and-replace takes three inputs: an input range and two values, X and Y. The find-and-replace operation scans the input range, one cell at a time, replacing any occurrence of 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 nearly constant.

Observations. For Excel, Calc, and Google Sheets, we run the experiments up to $110k$, $60k$, and $30k$ rows, respectively (see Figure 9). 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 spreadsheet systems, a linear trend emerges for find-and-replace operations—an expected trend in the absence of indexes. Even when searching a non-existent value, the search time increases linearly with the size of the data. As the value doesn't exist, the replace operation is skipped, leading to faster completion time for a non-existent value. Surprisingly, Google Sheets takes the same time for both operations.

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

5.2 In-memory Data Layout

Next, we wanted to see whether spreadsheets employ an intelligent in-memory layout to ensure faster access to data relevant to a formulae. As majority of spreadsheet formulae operate on contiguous cells, physically laying out data "nearby" on the spreadsheet close to each other can benefit from cache access locality. As computational operations like aggregate and lookup typically operate on a spreadsheet column, we focus on identifying whether a columnar data layout is used. To this end, we run two experiments: sequential data access and random data access. For sequential data access, 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 random data access, we randomly select a row and then get the value of cell corresponding to column A within that row. We used three different row ranges of the Value-only dataset: 100k, 300k, and 500k. If spreadsheets use a columnar layout, the sequential access would be much faster than random access due to cache locality.

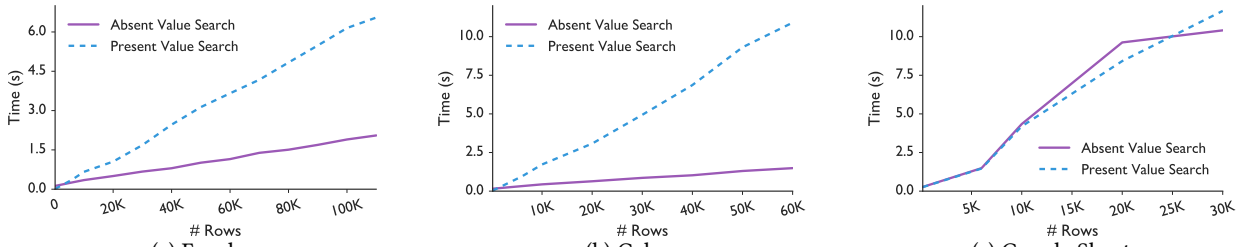
Observations. According to Figure 10, the time for sequential and random access of spreadsheet data is very similar. Therefore, none of the systems utilize any intelligent in-memory layout to speed up data access.

Takeaway: *Spreadsheet systems do not employ a columnar data layout to improve computational (e.g., aggregation) performance.*

5.3 Shared Computation

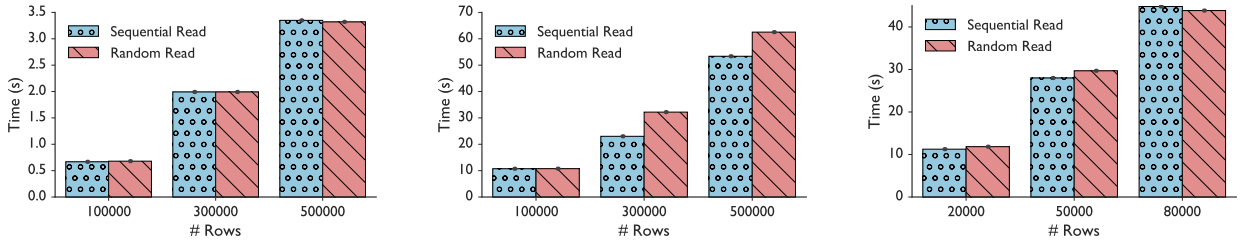
In Section 4, we identified that recomputation of a collection of formulae within spreadsheets severely impacts the execution time of any new formula. Therefore, we want to understand why formula recomputation is so expensive in spreadsheets. As many spreadsheet formulae involve referencing the same region within the spreadsheet, we wanted to see if these formulae share access to this region, and if possible, share computation of sub-expressions.

To understand whether spreadsheet systems perform shared computation, we conduct an experiment where we programmatically 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 11a). For our experiment, 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



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

Figure 9: A linear trend for *Find and Replace* indicates the absence of an index.



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

Figure 10: For all three systems, sequential and random access of a column takes roughly the same time indicating the absence of a columnar data layout.

10k. One way to compute the cumulative sum upto row i is by using the SUM formula (see column B in Figure 11a) which calculates the sum over the entire input range. We call this a *repeated* computation approach. Another efficient way to compute the same formula is by adding the already computed cumulative sum up to row $i-1$ with the value of cell Ai. We call this approach *reusable* computation. In a shared computation scenario, given this experiment, we expect the time complexity of both approaches (that are computing the same exact final result) to scale linearly with the number of formulae (see column C in Figure 11a). We now compare the completion time of each approach.

Observations. The results in Figure 11 show that, for all three spreadsheet systems, the repeated computation approach 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, the reusable computation approach, where the number of cell references increases linearly with the number of formulae, exhibits an $O(m)$. The approach mimics a shared computation scenario: a collection of spreadsheet formulae the input range of which overlap, can share computation to optimize performance. However, the current spreadsheet systems do not employ any such optimizations, as confirmed by this experiment.

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

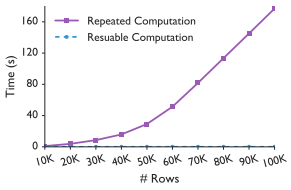
5.4 Eliminating Redundant Computation

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.

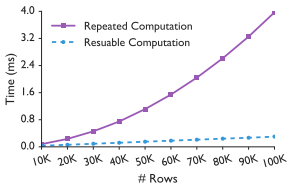
Observations. The result shows that the completion time of five formula instances takes $\approx 5X$ more time than a single instance of the COUNTIF formula (see Figure 12). We see similar results for Calc and Google Sheets. Therefore, spreadsheet systems do not test for formula equality (*e.g.*, by 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

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)

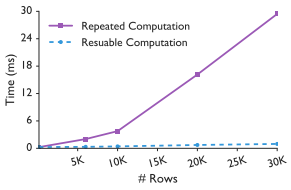
(a) Sample Data



(b) Excel

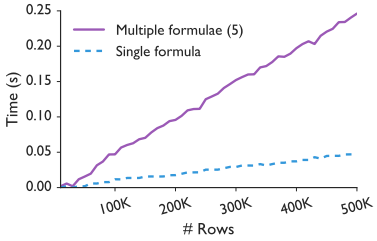


(c) Libre

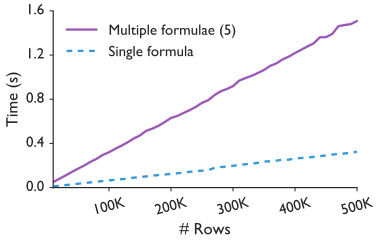


(d) GS

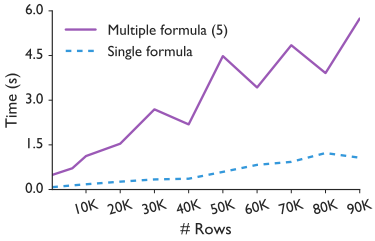
Figure 11: Expressing the same computation in two different ways (repeating the computation vs. reusing as much as possible) leads to substantial differences in runtime complexity (quadratic vs. linear), indicating no sharing of computation.



(a) Excel



(b) Calc



(c) Google Sheets

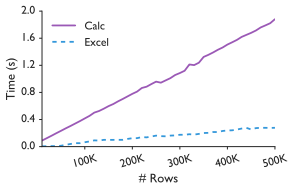
Figure 12: All three systems redundantly compute duplicate instances of a COUNTIF formula instead of reusing the previously computed result, causing the execution time to increase linearly with the number of duplicates.

$N = 2, 3, 4$ formula instances which yielded similar results—the computation time scales linearly with the number of formulae instances.

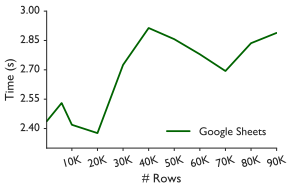
Takeaway: *Spreadsheet systems do not even detect and avoid entirely redundant computation of identical formulae.*

5.5 Incremental Updates

The purpose of this experiment is to see whether spreadsheet formulae can efficiently handle updates to cells that the formulae operate on. One way to handle updates is to materialize the results of the formula, compute the difference (or delta) between the old and new value of a cell and then update the results. This is analogous to incremental view updates in relational databases. 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 how much time it requires to recompute the formulae. With results of formulae being materialized, a formula would require near constant time to recompute after the update of a single cell within the region referenced by the formula.



(a) Excel and Calc

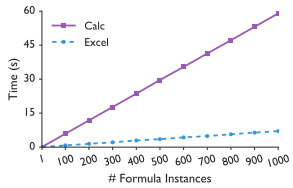


(b) Google Sheets

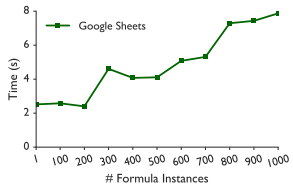
Figure 13: All three systems recompute the results of a COUNTIF formula from scratch after a single cell update.

Observations. According to Figure 13, the run time for both 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 view 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 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, we use the 500k Value-only dataset for the desktop-based spreadsheets and 90k Value-only dataset for Google Sheets.



(a) Excel and Calc



(b) Google Sheets

Figure 14: While recomputing a mere 100 instances of a COUNTIF formula following a single cell update, all systems violate the interactivity bound.

Observations. As shown in Figure 14, 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.

Takeaway: *None of the spreadsheet systems perform incremental recomputation for small updates, instead recomputing formulae from scratch.*

5.6 Discussion

The OOT benchmark reveals that none of the spreadsheet systems employ any optimization strategies adopted by relational databases. Spreadsheet systems do not implement indexes, share computation, eliminate redundant computation, perform incremental updates, or use 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, as is detailed in recent work [32].

6 CONCLUSION AND DISCUSSION

We now summarize the findings from our benchmarking experiments and discuss ways to improve spreadsheet systems, informed by our experimental results and previous work. Our BCT benchmark highlights the fact 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 spreadsheet systems achieve interactive response times only when operating over small datasets that are often a single-digit percentage fraction of their documented 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 optimization techniques. The OOT benchmark confirms that spreadsheet systems do not perform any such optimizations.

Database-style Optimizations. Introducing "database-style" optimizations within spreadsheet systems is a promising research direction, with the potential to substantially improve the interactivity of spreadsheets on moderate-to-large datasets; 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 indicate the columns that may benefit from indexing. Indexing may be counterproductive for spreadsheets where the raw data is being heavily edited, and may be more useful for those in the "analysis" phase. Indexing could also be valuable for find-and-replace-type operations, but this would require indexing the strings in all of the cells of the sheet as opposed to just a column. We should also recognize 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 to this issue [21]. Finally, the structure of the data [23] and the formulae could be together used to reorganize the data to optimize data access.

Shared computation. It is clear from Section 5.4 that spreadsheet systems need to go beyond cell-by-cell retrieval and execution of formulae, actively identifying sharing opportunities. These shared computation 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.

Incremental updates. This optimization, whose absence we confirmed in Section 5.5, is perhaps the easiest to implement for spreadsheet systems. For many aggregation style operations, *e.g.*, AVG, SUM, COUNTIF, the results can be recomputed using the current aggregate value and the "delta", without requiring a recomputation from scratch. In some cases such as AVGIF (*i.e.*, compute average of a set of cells, if a condition is met) we may want to additionally maintain the count of the number of cells that meet that condition in addition to the average. An interesting research question is to see if incremental updates can be used for other types of computation beyond aggregation, such as VLOOKUP.

Detecting what needs recomputation. Section 4 demonstrated that the Formula-value datasets often performed much worse than Value-only for all three spreadsheet systems thanks to poor detection of what actually needs to be recomputed 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 formulae with absolute references, *e.g.*, "C1 = A1 + B1", require recomputation.

Additional Optimizations. There are other potential optimizations from the research literature that slightly change spreadsheet semantics for increased interactivity.

For example, spreadsheet systems operate synchronously; they remain unresponsive while performing computation and return control after computation completes. Recent work has employed asynchronous computation to make spreadsheets more interactive, covering up in-progress formula computation with a progress bar [22]. Asynchrony can be adapted to other operations like open and sort. 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* [37] to support large spreadsheet

datasets [36], allowing users to operate on the data before it is completely sorted, while prioritizing the visible areas.

Another promising direction for increasing interactivity is to use a database backend for efficient execution by translating formulae into SQL queries [21, 25, 30], *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 [28], as well as other approximate query processing schemes [27], allowing users to terminate their execution early if needed.

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

REFERENCES

[1] Excel file extensions. https://en.wikipedia.org/wiki/List_of_Microsoft_Office_filename_extensions.

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

[3] Excel limit. https://support.office.com/en-us/article/excel-specifications-and-limits-1672b34d-7043-467e-8e27-269d656771c3.

[4] Excel Multi-threaded Calculation. https://docs.microsoft.com/en-us/office/client-developer/excel/multithreaded-recalculation-in-excel.

[5] Excel vs. Google Sheets usage—nature and numbers. https://medium.com/grid-spreadsheets-run-the-world/excel-vs-google-sheets-usage-nature-and-numbers-9dfa5d1cadbd/.

[6] Formula calculation in Microsoft Excel. https://docs.microsoft.com/en-us/office/client-developer/excel/excel-recalculation/.

[7] G Suite. https://gsuite.google.com/.

[8] Google Apps Script. https://developers.google.com/apps-script.

[9] Google sheets limit. https://support.google.com/drive/answer/37603.

[10] How finance leaders can drive performance. https://enterprise.microsoft.com/en-gb/articles/roles/finance-leader/how-finance-leaders-can-drive-performance/.

[11] Layout thrashing. https://developers.google.com/web/fundamentals/performance/rendering/avoid-large-complex-layouts-and-layout-thrashing.

[12] LibreOffice Basic. https://documentation.libreoffice.org/en/english-documentation/macro/.

[13] LibreOffice Calc. https://en.wikipedia.org/wiki/LibreOffice_Calc.

[14] LibreOffice Online. https://www.libreoffice.org/download/libreoffice-online/.

[15] List of spreadsheet software. https://en.wikipedia.org/wiki/List_of_spreadsheet_software.

[16] Office 365. https://www.office.com/.

[17] OpenDocument. https://en.wikipedia.org/wiki/OpenDocument.

[18] OpenOffice is dead. Long live LibreOffice. https://www.zdnet.com/article/openoffice-is-dead-long-live-libreoffice/.

[19] Pivot Table. https://en.wikipedia.org/wiki/Pivot_table.

[20] VBA. https://docs.microsoft.com/en-us/office/vba/api/overview/.

[21] M. Bendre, V. Venkataraman, X. Zhou, K. Chang, and A. Parameswaran. 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)*, pages 113–124. IEEE, 2018.

[22] M. Bendre, T. Wattanawaroon, K. Mack, K. Chang, and A. Parameswaran. Anti-freeze for large and complex spreadsheets: Asynchronous formula computation. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, pages 1277–1294, 2019.

[23] Z. Chen, M. Cafarella, J. Chen, D. Prevo, and J. Zhuang. Senbazuru: a prototype spreadsheet database management system. *Proceedings of the VLDB Endowment*, 6(12):1202–1205, 2013.

[24] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154. ACM, 2010.

[25] J. Cunha, J. Saraiva, and J. Visser. From Spreadsheets to Relational Databases and Back. In *SIGPLAN*, PEPM '09, pages 179–188, New York, NY, USA, 2009. ACM.

[26] C. Cunningham, C. A. Galindo-Legaria, and G. Graefe. Pivot and unpivot: Optimization and execution strategies in an rdbms. In *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*, pages 998–1009. VLDB Endowment, 2004.

[27] M. N. Garofalakis and P. B. Gibbons. Approximate query processing: Taming the terabytes. In *VLDB*, pages 343–352, 2001.

[28] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. In *Acm Sigmod Record*, volume 26, pages 171–182. ACM, 1997.

[29] S. T. Leutenegger and D. Dias. *A modeling study of the TPC-C benchmark*, volume 22. ACM, 1993.

[30] B. Liu and H. V. Jagadish. A spreadsheet algebra for a direct data manipulation query interface. In *Proceedings of the 25th International Conference on Data Engineering, ICDE 2009, March 29 2009 - April 2 2009, Shanghai, China*, pages 417–428, 2009.

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

[32] K. Mack, J. Lee, K. Chang, K. Karahalios, and A. Parameswaran. Characterizing scalability issues in spreadsheet software using online forums. In *Extended Abstracts of the 2018 CHI Conference on Human Factors in Computing Systems*, page CS04. ACM, 2018.

[33] B. A. Nardi and J. R. Miller. *The spreadsheet interface: A basis for end user programming*. Hewlett-Packard Laboratories, 1990.

[34] A. Parameswaran. Enabling data science for the majority. volume 12, pages 2309–2322. VLDB Endowment, 2019.

[35] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 165–178. ACM, 2009.

[36] V. Raman et al. Scalable spreadsheets for interactive data analysis. In *ACM SIGMOD Workshop on DMKD*, 1999.

[37] V. Raman, B. Raman, and J. M. Hellerstein. Online dynamic reordering. *The VLDB Journal*, 9(3):247–260, 2000.

[38] J. Zobel and A. Moffat. Inverted files for text search engines. *ACM computing surveys (CSUR)*, 38(2):6, 2006.