Benchmarking Spreadsheet Systems

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

Kelly Mack*
University of Washington
knmack3@uw.edu

Mangesh Bendre*
VISA Research
mbendre@visa.com

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

CCS CONCEPTS

• **Information systems → Data management systems**.

KEYWORDS

Spreadsheet systems; Scalability; Use cases

ACM Reference Format:
Sajjadur Rahman, Kelly Mack, Mangesh Bendre, Ruilin Zhang, Karrie Karahalios, and Aditya Parameswaran. 2020. Benchmarking Spreadsheet Systems. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD'20), June 14–19, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3318464.3389782

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

SIGMOD'20, June 14–19, 2020, Portland, OR, USA

ACM ISBN 978-1-4503-6735-6/20/06…$15.00
https://doi.org/10.1145/3318464.3389782

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 [18]. 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 [22]. Even web-based Google Sheets now supports five million cells [10], 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 [18]. Further, importing a spreadsheet of 100,000 rows in Excel (10% of the size limit of one million rows) can take over 10 minutes [1].

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 [14], data analysis [26], and cloud computing [3].

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)*. 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 [17].

Optimization Opportunities Testing (OOT). Spreadsheet systems have continued to increase their size limits over the past few decades [10, 22]. 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 [21]—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

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

violate the interactivity bound on a spreadsheet with 110k, and 10k rows, respectively.

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 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

2 BENCHMARK SETUP

We first provide a brief overview of the spreadsheet systems that we are benchmarking, namely, Excel, Calc, and Google Sheets. 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.

2.1 Spreadsheet Systems Overview

We evaluate three popular systems: Excel and Calc, which are both desktop-based (and operate on MacOS and Windows, unlike Numbers, which only operates on MacOS), and Google Sheets, which is web-based. Excel, part of the Office 365 suite [25], is the most popular desktop-based spreadsheet system, boasting about 700M registered users [24]. Excel can support up to 1M rows and 17,000 columns in a given spreadsheet [22]. Calc is an open-source spreadsheet system used by two office software suites, OpenOffice and LibreOffice [32], and can support up to one million rows per spreadsheet [31]. Google Sheets, part of G suite [8], is the most popular web-based spreadsheet system, with users numbering in the 100s of millions [30]. Google Sheets supports up to five million cells per spreadsheet [10]. A detailed overview of these systems can be found in the techreport [29]. We also assume that readers are familiar with basic spreadsheet concepts, such as cells and formulae, and the fact that spreadsheet computation is *synchronous*, leading to performance issues as documented in recent work [2, 18]—for a primer, see [29].

2.2 Taxonomy of Spreadsheet Operations

We first group spreadsheet operations into three categories: data load, update, and query 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.

2.3 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 [29] 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.

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

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	$O(m_x n_x m_y n_y)$

$N_i = 10000 + (i-3) \times 10000$, where $i = 3, 4, 5, \ldots, 51$. We provide further details on dataset creation in [29].

2.4 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 [23]. However, the default setting is to evaluate a formula on the main thread of Excel.

For each experiment in Excel, we first created an Excel Macro-Enabled Workbook (*xlsm*) [34] 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*) [33]. We created the Google App Scripts in G Suite Developer Hub [9]. 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.

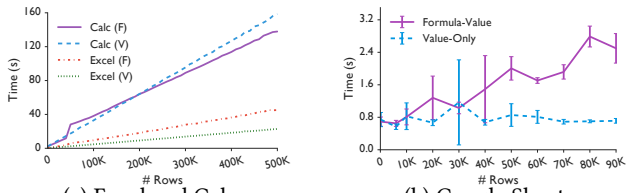
For each experiment, we ran ten trials and measured the running time. We report the average run time of eight trials while removing the maximum and minimum reported time. Note that for experiments with Google Sheets, we restricted the maximum size of the data to 90k rows to fit in the experiment trials for different test cases within the allocated daily quotas imposed by Google Apps Script services. 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 our techreport [29].

3 BCT BENCHMARK

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 [17]. 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. See technical report [29] for more details.

3.1 Data Load Operations

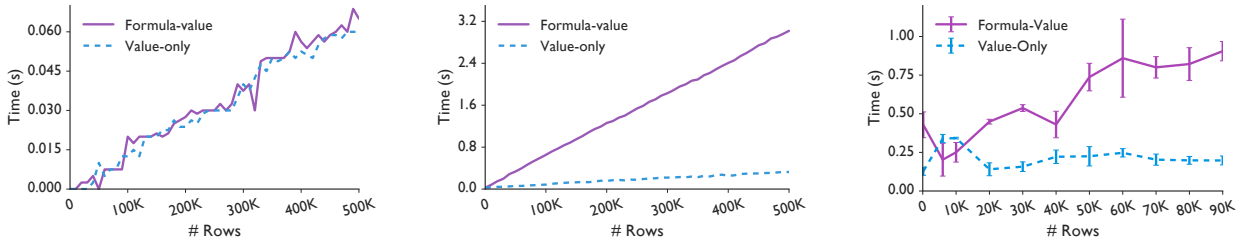
The open operation loads an existing spreadsheet from disk to memory. We document the time to open Formula-value (F) and Value-only (V) datasets, while varying row sizes m, where $m = 150, 6k, 10k, 20k, \ldots, 500k$. As we keep the number of columns fixed, the expected complexity is $O(m)$.



(a) Excel and Calc (b) Google Sheets

Figure 1: *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 1a shows that the time taken by the desktop-based spreadsheets is linear in m for both datasets. 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 1b). When opening a spreadsheet for the first time, *Google Sheets appears to*



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

Figure 2: 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.

load the first m rows visible within the screen, and then load the rest on-demand as the user scrolls. However, Google Sheets breaks the interactivity threshold of 500ms to load even a screenful of data, possibly due to network or web rendering delays [11]. 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. 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 [20], but we expect other systems are similar), and as the number of embedded formulae increases, the latency of open increases as well.

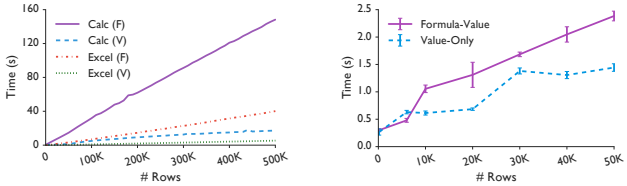
Thus, beyond prioritizing loading the first "window" of the spreadsheet, there are additional opportunities to reduce the latency of data load by prioritizing formula computation for the first window, done by no systems currently.

3.2 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 4.

3.2.1 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. 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 2 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. 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.



(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.

3.2.2 Sort. 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 3.

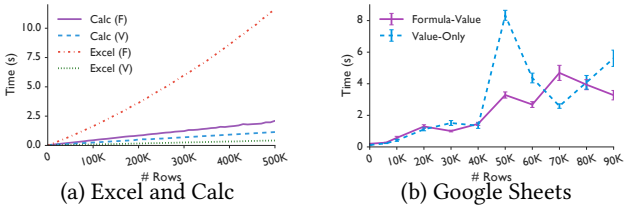
Observations. The deceptively linear trend for sorting for all systems is due to the size of the datasets used in our experiments—even row size m = 500k is not large enough for the logarithmic factor to be pronounced for the $O(m \log m)$ trend. Similar to data load operations, Excel, Calc, and Google 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). 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.

3.3 Query Operations

We now discuss the results for four query operation categories. 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.

3.3.1 Select (Filter). In this experiment, we filter a given spreadsheet by state *SD* (South Dakota). We vary the row count, m, and expect the run time to be linear in m.

Observations. As can be seen in Figure 4, 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* 40k

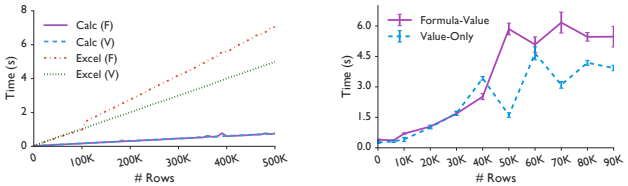


(a) Excel and Calc (b) Google Sheets

Figure 4: *Filter* on Formula-value in Excel does unnecessary recomputation. Google Sheets is slower than the other two.

rows (Figure 4a). Filtering likely triggers unnecessary formula recalculation in Excel [20]. 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.

3.3.2 Report (Pivot Table). The pivot table operation [6] creates a table with summary statistics (similar to a SQL GROUP BY). In this experiment, we create a pivot table that shows the *sum of storms* per state in a new worksheet.



(a) Excel and Calc (b) Google Sheets

Figure 5: Calc is faster than the other two for Pivot Tables

Observations. Figure 5 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.

3.3.3 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. Therefore, the aggregate operation is a special case of pivot tables. Due to the similarity in experimental outcomes, we discuss these results in our technical report [29].

3.3.4 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 searching for an integer X and return the corresponding US state for the row i such that $A_i = X$, where $X = 200000$. 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.

Observations. Figure 6 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. 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. 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.

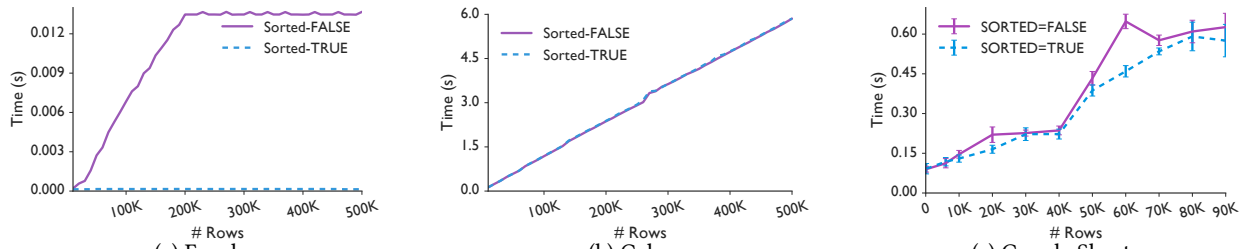
3.4 Discussion

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.

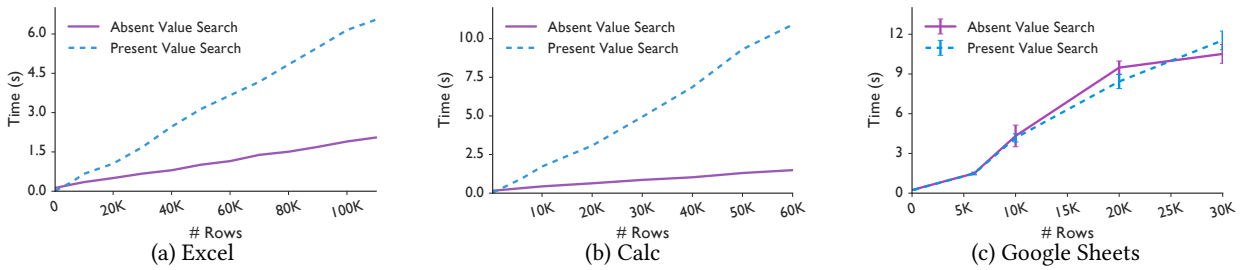
	Formula-value			Value-only		
	E (%)	**C** (%)	**G** (%)	**E** (%)	**C** (%)	**G** (%)
Open	**0.6**	0.015	0.05	**0.6**	0.015	0.05
Sort	1	0.6	**3.4**	**7**	1	2.04
Conditional Formatting	**100**	8	17	**100**	**100**	**100**
Filter	4	**12**	3.4	**100**	20	6.8
Pivot Table	5	**34**	3.4	5	**33**	6.8
COUNTIF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



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

Figure 6: 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 7: A linear trend for *Find and Replace* indicates the absence of an index.

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?

4 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.

4.1 Indexing

We now explore whether spreadsheets maintain indexes on the columns to facilitate faster computation for find-and-replace. We have already seen earlier (Figure 6) that none of the systems employ indexes for VLOOKUP, leading to linear scaling with size; likewise, no system uses indexes for COUNTIF [29]. For find-and-replace, we wanted to see if spreadsheet systems perform inverted indexing [35]. 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.

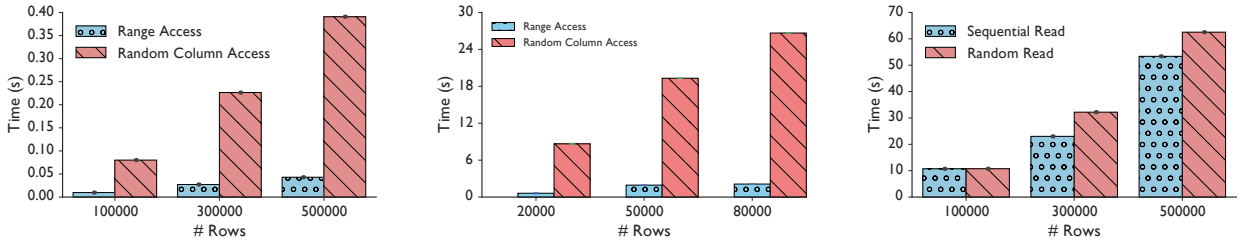
Observations. For all three systems, we see a linear trend that violates interactivity at 10k, indicating the absence of indexes. Even when searching a non-existent value, the search time scales linearly. As the value doesn't exist, replace is skipped, leading to faster completion for a non-existent value. Surprisingly, Google Sheets takes the same time in both cases.

4.2 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.

4.2.1 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 8a and Figure 8b, range access is orders of magnitude faster than random column access for both Excel ($\approx 10X$) and Google Sheets ($\approx 11X$),



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

Figure 8: While Both Excel and Google Sheets employ a row-oriented data layout (indicated by faster range access than random column access), Calc employs a columnar data layout (indicated by faster sequential access than random access).

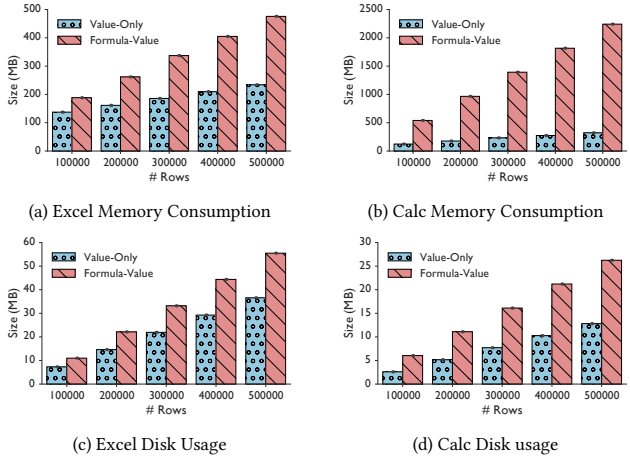
respectively, indicating a row-oriented data layout. However, the execution time for both types of data access is similar in Calc (see technical report [29]). To further verify that Calc uses a column-oriented data layout, we employ a second set of experiments, discussed next.

4.2.2 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. 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 technical report [29]), re-affirming a row-oriented data layout. However, as shown in Figure 8c, 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 [19] which explains better sequential data access performance. However, the improvement in sequential access is not proportional to the number of rows accessed.

4.2.3 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 the cell-dependency graph and formula calculation chain—both these data structures are loaded in memory along with the spreadsheet impacting Formula-value datasets [20]; so we compare Formula-value and Value-only datasets for this experiment, and focus on desktop-based systems.

Observations. Figure 9 shows the disk and memory size of the desktop-based systems for Value-only and Formula-value. We discuss how we measured these quantities in our technical report [29]. Excel's in-memory representation is



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

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

Figure 9: 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.

up to 10× that of the datasets on disk, while Calc's representation is up to 87× for Formula-value, and 33× 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.

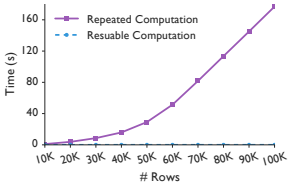
4.3 Shared Computation

In Section 3, 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.

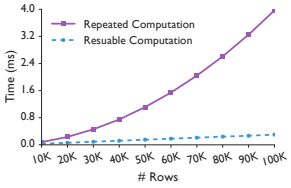
We conduct an experiment where we insert a formula within each cell i of a column that computes the following: $\sum_{j=1}^{i} Aj$, *i.e.*, the cumulative sum of cells of column A up to row i (see Figure 10a) where $10k \leq i \leq 100k$. One way to compute this cumulative sum is the *repeated* computation approach, using the SUM formula (see column B in Figure 10a) 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

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)

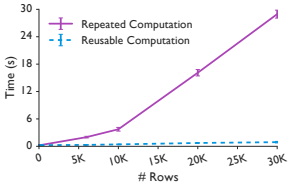
(a) Sample Data



(b) Excel

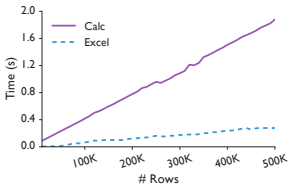


(c) Libre

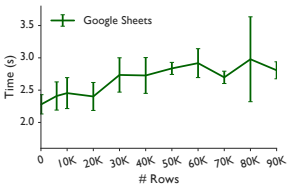


(d) GS

Figure 10: 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 11: All three systems recompute the results of a COUNTIF formula from scratch after a single cell update.

approaches (computing the same final result) to scale linearly with the number of formulae (see column C in Figure 10a).

Observations. Figure 10 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.*, $\sum_{i=1}^{m} i = O(m^2)$. On the other hand, reusable computation 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.

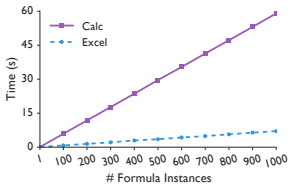
Redundant Computation. Our previous experiment revealed a setting where shared computation was not used by spreadsheet systems; the systems were not able to detect sharing opportunities and use them to reduce computation. We wanted to test an extreme (and very obvious to detect) version of shared computation—one where the formulae being computed were *exactly* the same. Unfortunately k identical instances of the same COUNTIF formula took k times as much time as one—thus even entirely redundant computation is not eliminated by spreadsheet systems; see [29] for details.

4.4 Incremental Updates

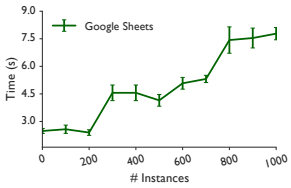
Next, we wanted to see whether spreadsheet formulae can efficiently handle updates to cells that the formulae operate on. One approach can be to materialize the formula result, compute the difference between the old and new value of a cell and then update the results, analogous to incremental view updates. We run this experiment on the following formula "=COUNTIF($J2:Jm$,"1")". 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, formula recomputation would take near constant time.

Observations. Figure 11 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 12: While recomputing a mere 100 instances of a COUNTIF formula following a single cell update, all systems violate the interactivity bound.

Observations. Figure 12 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 [18].

5 CONCLUSION AND NEXT STEPS

We now summarize the findings from our experiments and discuss ways to improve spreadsheet systems. We found that even though spreadsheet systems operate on in-memory data, they remain interactive for only a few operations through bespoke optimizations. 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 4.1, 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 [1]. As Figure 9 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.

Shared computation. It is clear from Section 4.3 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 [2].

Incremental updates. 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.

Detecting what needs recomputation. The Formula-value datasets often performed much worse than Value-only datasets due to poor detection of what needs recomputation (see Section 3). 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. One can employ asynchrony to increase interactivity, covering up in-progress computation with a progress bar [2]. Asynchrony can be adapted to other operations like open and sort, targeting the visible window, as is done in Google Sheets, and proposed in prior work [27, 28]. We can also use a database backend for efficient execution by translating formulae into SQL queries [1, 5, 16], *e.g.*, a join instead of a collection of VLOOKUPs. Efficient execution can also happen via *approximation* [7, 13], enabling early termination.

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 4.1): Calc has a columnar MDDS data store for optimized data access, with SSE optimization [4] for columnar SUMS. The Calc team acknowledged the trade-off between performance and storage (Section 4.2.3), 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 4.3), 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 [12]. Calc benchmarks performance using the open-source Callgrind Test Suite [15].

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] 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.

[2] 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.

[3] 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.

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

[5] 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.

[6] 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.

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

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

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

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

[11] 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

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

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

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

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

[16] 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.

[17] 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.

[18] 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.

[19] MDDS. 2020. Multi-dimensional data structures. (2020). Retrieved April 8, 2020 from https://gitlab.com/mdds/mdds

[20] Microsoft. 2020. Excel calculation engine. (2020). Retrieved April 8, 2020 from https://docs.microsoft.com/en-us/office/client-developer/excel/excel-recalculation/

[21] 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

[22] Microsoft. 2020. Excel limit. (2020). https://support.office.com/en-us/article/excel-specifications-and-limits-1672b34d-7043-467e-8e27-269d656771c3

[23] Microsoft. 2020. Excel threading. (2020). Retrieved April 8, 2020 from https://docs.microsoft.com/en-us/office/client-developer/excel/multithreaded-recalculation-in-excel

[24] 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/

[25] Microsoft. 2020. Office 365. (2020). Retrieved April 8, 2020 from https://www.office.com/

[26] 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.

[27] Vijayshankar Raman et al. 1999. Scalable Spreadsheets for Interactive Data Analysis. In *ACM SIGMOD Workshop on DMKD*.

[28] Vijayshankar Raman, Bhaskaran Raman, and Joseph M Hellerstein. 2000. Online dynamic reordering. *The VLDB Journal* 9, 3 (2000), 247–260.

[29] Technical Report. 2020. Benchmarking Spreadsheet Systems. (2020). Retrieved April 8, 2020 from http://srahman7.web.engr.illinois.edu/papers/benchmarking_spreadsheets_techreport.pdf

[30] 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/

[31] Wikipedia. 2020. LibreOffice Calc. (2020). Retrieved April 8, 2020 from https://en.wikipedia.org/wiki/LibreOffice_Calc/

[32] Wikipedia. 2020. List of spreadsheet softwares. (2020). Retrieved April 8, 2020 from https://en.wikipedia.org/wiki/List_of_spreadsheet_software

[33] Wikipedia. 2020. OpenDocument format. (2020). Retrieved April 8, 2020 from https://en.wikipedia.org/wiki/OpenDocument

[34] Wikipedia. 2020. XLSM file. (2020). Retrieved April 8, 2020 from https://en.wikipedia.org/wiki/List_of_Microsoft_Office_filename_extensions

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