TOWARD EFFICIENT SPREADSHEET COMPUTATION AND VISUALIZATION

Christopher De Leon
Aditya Parameswaran, Ed.
Dixin Tang, Ed.

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2022-67
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2022/EECS-2022-67.html

May 11, 2022

Acknowledgement

I'd like to give a special thanks to Professor Aditya Parameswaran for advising me during my candidacy. I couldn't have asked for a more dedicated, thoughtful, and committed advisor to guide me through my time in the program.

I'd like to give a special thank you to Dr. Dixin Tang for supporting me immensely throughout my candidacy. His advice and guidance have made me feel well supported and have allowed me to successfully fulfill my research goals.

I would like to thank Tana Wattanawaroon and Richard Lin for acting as a great source of advice and inspiration during this project. Finally, I would like to thank the whole DataSpread team, who I've happily worked with over the past couple years. It's been an incredible journey and I wouldn't have

made it this far without their support!

TOWARD EFFICIENT SPREADSHEET COMPUTATION AND VISUALIZATION

A THESIS SUBMITTED TO
THE UNIVERSITY OF CALIFORNIA BERKELEY GRADUATE SCHOOL OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
IN
ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

By
Chris De Leon

May 2022

Toward Efficient Spreadsheet Computation and Visualization

by Chris De Leon

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Professor Aditya Parameswaran
Research Advisor

5/11/2022
(Date)

****

Dixin Tang

Dr. Dixin Tang
Second Reader

5/11/2022
(Date)

ABSTRACT

TOWARD EFFICIENT SPREADSHEET COMPUTATION AND VISUALIZATION

Chris De Leon
M.S. in Electrical Engineering and Computer Science
Advisor: Professor Aditya Parameswaran
May 2022

Spreadsheets are ubiquitous tools that offer users an intuitive interface for interacting with complex data. However, there are two major problems with modern spreadsheet systems: scalability and usability. Scalability broadly refers to a spreadsheet's ability to remain responsive when dealing with a large dataset, and usability measures how well users can make sense of a spreadsheet. Currently, spreadsheet systems have trouble supporting large datasets that are increasingly common interactively. They also do not provide users with sufficient tooling to better understand the structural layout of their spreadsheets. These limitations can make it much more difficult for most users to identify errors, keep the spreadsheet organized for other users, and for the system to optimize certain computations. To address these issues, we present TACO (Tabular Locality-Based Compression), a framework for performing more efficient spreadsheet computation, and Sherlock, an interface that leverages the TACO framework to support effective sense making of spreadsheets.

Keywords: spreadsheet, visualization, automation.

Acknowledgement

I would like to give a special thanks to Professor Aditya Parameswaran for advising me during my candidacy. I could not have asked for a more dedicated, thoughtful, and committed advisor to guide me through my time in the program.

I would like to give a special thank you to Dr. Dixin Tang for supporting me immensely throughout my candidacy. His advice and guidance have made me feel well supported and have allowed me to successfully fulfill my research goals.

I would like to thank Tana Wattanawaroon and Richard Lin for acting as a great source of advice and inspiration during this project. Finally, I would like to thank the whole DataSpread team, who I have happily worked with over the past couple years. It's been an incredible journey and I wouldn't have made it this far without their support!

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Spreadsheets are a popular choice for users looking to analyze and make sense of their data. In particular, they provide a powerful and intuitive UI that allows users who may not be familiar with traditional relational database systems to perform similar operations with ease. While modern spreadsheet systems provide users with a great deal of power in wrangling their datasets, they face two over-arching challenges: scalability and usability [1, 2, 3]. Many modern spreadsheet systems have trouble scaling to the size of modern datasets and oftentimes crash or freeze for simple computations involving no more than 100,000 rows. Furthermore, current spreadsheet systems do not provide users with sufficient tooling to better understand the structural layout of their spreadsheets, which can make it much more difficult to identify errors and keep the spreadsheet clean and organized for other users.

1.1 Spreadsheet Scalability Issues

It is no question that we live in an age of big data and the need to analyze this data has become a more important need than ever. However, many popular spreadsheet systems have not scaled appropriately to accommodate today's data analysis needs [2]. In particular, modern spreadsheet systems like Excel end up hanging or freezing for extended periods of time when users perform a single cell update on spreadsheets with as few as 50,000 rows [3]. Some spreadsheet systems have increased the memory size limitations of their software to address this problem [4, 5]. However, the computation involved in processing even a small subset of these cells still takes a substantial amount of time [3], [1], which renders the software often unusable for today's demanding data analysis requirements.

We refer to scalability as a spreadsheet system's ability to remain responsive when dealing with a large dataset. One of the major factors that contributes to a spreadsheet system's responsiveness is its internal formula graph implementation discussed more next. A formula graph keeps track of the dependencies between cells, and is used internally by many modern spreadsheet systems today [5, 6, 7]. When a user edits a cell on a spreadsheet, the spreadsheet system queries its formula graph to determine the cells that require recalculation. As we will see later, real-world formula graphs are typically very large and complex. Thus, the action of querying dependencies becomes a severe performance bottleneck leading to much slower recalculations and potentially even UI crashes.

1.2 Structural Complexities of Real-World Spreadsheets

As previously mentioned, spreadsheets allow users to express their computations as formulae, which opens the door to more complex data analysis. One formula cell can reference one or more other formulae creating what is known as a formula graph. A formula graph captures the dependencies between cells and can be used to determine how an update to one formula or data cell affects others. One challenge with spreadsheet systems is that they provide very minimal tooling for users who wish to deeply understand a formula graph and the dependencies between cells in their spreadsheet in a more human-readable way. For example, modern spreadsheet systems such as Excel have a built-in dependency tracker that can find the precedents and dependents of a cell. While this can provide the user with clear insights for very simple spreadsheets, most real-world sheets can have hundreds of thousands of dependencies. As a result, the tracker becomes much less helpful and users have a more challenging time conceptualizing their computations.

Take the spreadsheet in Figure 1.1 for example. Here we see that cell $B7$ has a significant number of dependent cells with more than half the sheet covered by overlapping dependency arrows. A closer look into the spreadsheet computation reveals that many of B7's dependents perform roughly the same operation (e.g., $A1+B7$, $A2+B7$, ...). This type of situation appears very frequently in real-world spreadsheets because many users tend to take advantage of spreadsheet shortcuts such as copy-paste and autofill. These shortcuts generate similar copies of the same formula over large portions of the sheet, and as a result, the use of these functionalities leads to very dense graphs with repeated computation. This makes it nearly impossible for ordinary users to use Excel's dependency tracker to obtain any sort of useful information regarding the dependents of the cell, let alone how their computations are arranged.

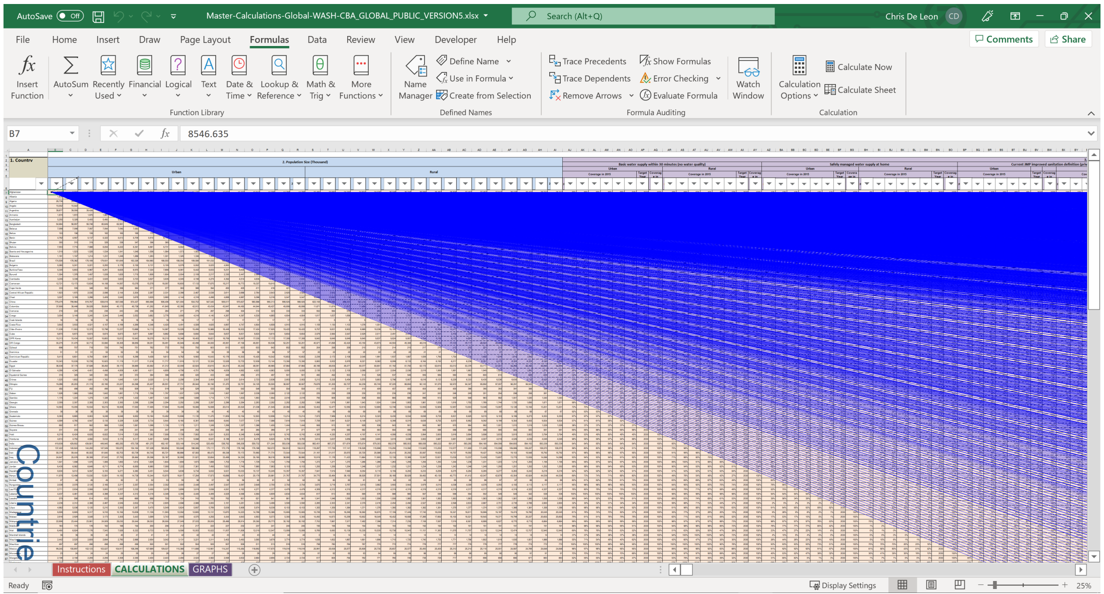


Figure 1.1: Excel Dependency Tracker for a Single Cell

To understand how frequently these complex spreadsheets appear in the real-world, we scraped a sample of 7.8K xlsx sheets from Github and analyzed the number of formulae in each spreadsheet as well as the number of dependencies of each cell. The results are displayed in Table 1.1. In the table, we list the name of the spreadsheet, the number of formulae in the spreadsheet, the number of vertices and edges in the uncompressed version of the spreadsheet's formula graph, and the largest number of direct dependents for a given cell in the spreadsheet. Note that real world spreadsheets can contain cells with up to 300,000 dependencies and over half a million formulae making it infeasible for a user to understand the relationships between cells and the consequences of updating a single value. We also see that these graphs can have millions of edges, so if we were to use Excel's existing formula dependency tracker to visualize the entire graph of cell dependencies, the graph could span several workbook frames making it much more difficult for the average user to catch computational errors, optimize their computations, or explain their work to others.

File name	Formulae	Vertices	Edges	Max Direct Dependents
Master-Calculati....xlsx	518662	618902	2353656	304082
Tio Cash Master....xlsx	258764	333341	580650	189106
xx-30000.xlsx	539997	540051	539997	179999
beerDB05.02_10.11.xlsx	234586	320608	547305	156385
WM5603_LT_tim....xlsx	175200	245281	350400	140160
aero_test.xlsx	131532	131533	131532	131532
InputData.xlsx	219120	350595	569712	131455
191001....xlsx	329897	406051	1116394	126820
datalog1.xlsx	124534	124535	124534	124534
Detroit data (stata...	258660	268240	258660	120827

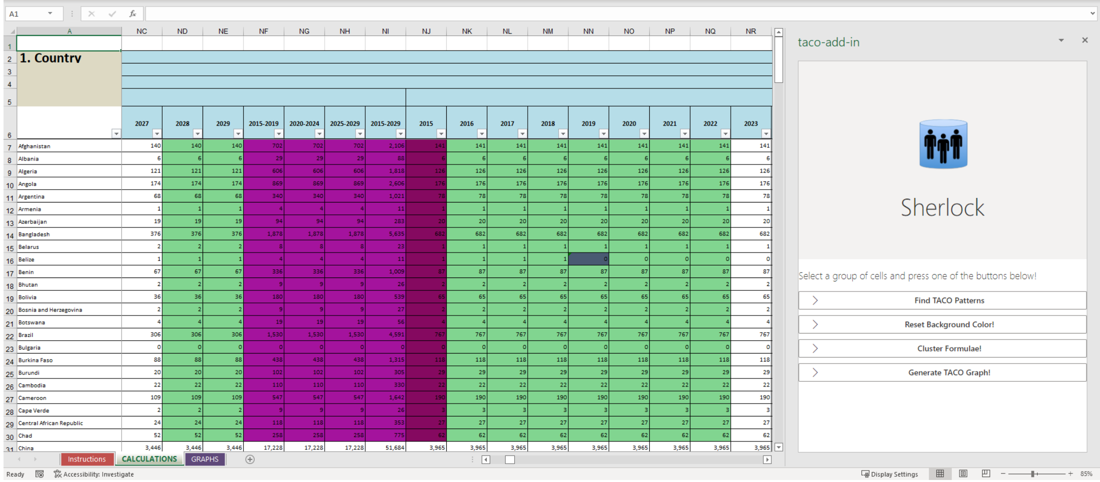
Table 1.1: Github data analysis results

1.3 TACO and Sherlock

To address the issue of spreadsheet scalability, we introduce the Tabular Locality-Based Compression framework or TACO. At a high level, TACO takes a spreadsheet as input and outputs a compressed version of its formula graph. The compressed version of the graph can then be used to quickly identify the cells that need to be re-computed whenever a cell on the sheet is updated. In the traditional formula graph representation, identifying cells that need re-computation can be a very time-consuming and CPU-intensive process especially for spreadsheets with more complex formula graphs. However, our framework makes use of a property known as tabular locality to achieve much faster dependency identification times. This in turn can allow spreadsheet systems to offer much better interactivity for users analyzing large datasets.

To help users make better sense of the organization and computation in their spreadsheet, we introduce Sherlock, an Excel add-in that users can use to visualize the spreadsheet's formula graph. Sherlock uses the TACO framework to identify useful patterns in spreadsheets and presents them to the user in a human-readable way. By providing users with a high-level overview of the patterns in their spreadsheet, users can more easily audit their computations and arrange their sheets in much cleaner ways.

Take Figure 1.2 for example. In the figure, we have used Sherlock to analyze a different portion of the same dense spreadsheet from Figure 1.1. Observe that Sherlock has not only identified four primary patterns in the selected range, but has also highlighted that cell $NN16$ may have a potential error. In simple terms, cells with the same color perform essentially the same computation and only differ by constant values or referenced cells. With that in mind, these four primary patterns correspond to four general types of computation. This representation is not only much clearer for the end user to interpret, but also greatly minimizes the effort involved in tracking errors in dense spreadsheets, which as we have seen above can be a very cumbersome process with existing tools.



Select a group of cells and press one of the buttons below!

	A	NC	ND	NE	NF	NG	NH	NI	NJ	NK	NL	NM	NN	NO	NP	NQ	NR
1																	
2	1. Countrv																
3																	
4																	
5																	
6		2027	2028	2029	2015-2019	2020-2024	2025-2029	2015-2029	2015	2016	2017	2018	2019	2020	2021	2022	2023
7	Afghanistan	140	140	140	702	702	702	2,106	141	141	141	141	141	141	141	141	141
8	Albania	6	6	6	29	29	29	88	6	6	6	6	6	6	6	6	6
9	Algeria	121	121	121	606	606	606	1,818	126	126	126	126	126	126	126	126	126
10	Angola	174	174	174	869	869	869	2,606	176	176	176	176	176	176	176	176	176
11	Argentina	68	68	68	340	340	340	1,021	78	78	78	78	78	78	78	78	78
12	Armenia	1	1	1	4	4	4	11	1	1	1	1	1	1	1	1	1
13	Azerbaijan	19	19	19	94	94	94	283	20	20	20	20	20	20	20	20	20
14	Bangladesh	376	376	376	1,878	1,878	1,878	5,635	682	682	682	682	682	682	682	682	682
15	Belarus	2	2	2	8	8	8	23	1	1	1	1	1	1	1	1	1
16	Belize	1	1	1	4	4	4	11	1	1	1	1	0	0	0	0	0
17	Benin	67	67	67	336	336	336	1,009	87	87	87	87	87	87	87	87	87
18	Bhutan	2	2	2	9	9	9	26	2	2	2	2	2	2	2	2	2
19	Bolivia	36	36	36	180	180	180	539	65	65	65	65	65	65	65	65	65
20	Bosnia and Herzegovina	2	2	2	9	9	9	27	2	2	2	2	2	2	2	2	2
21	Botswana	4	4	4	19	19	19	56	4	4	4	4	4	4	4	4	4
22	Brazil	306	306	306	1,530	1,530	1,530	4,591	767	767	767	767	767	767	767	767	767
23	Bulgaria	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
24	Burkina Faso	88	88	88	438	438	438	1,315	118	118	118	118	118	118	118	118	118
25	Burundi	20	20	20	102	102	102	305	29	29	29	29	29	29	29	29	29
26	Cambodia	22	22	22	110	110	110	330	22	22	22	22	22	22	22	22	22
27	Cameroon	109	109	109	547	547	547	1,642	190	190	190	190	190	190	190	190	190
28	Cape Verde	2	2	2	9	9	9	26	3	3	3	3	3	3	3	3	3
29	Central African Republic	24	24	24	118	118	118	353	27	27	27	27	27	27	27	27	27
30	Chad	52	52	52	258	258	258	775	62	62	62	62	62	62	62	62	62
31	China	3,446	3,446	3,446	17,228	17,228	17,228	51,684	3,965	3,965	3,965	3,965	3,965	3,965	3,965	3,965	3,965

Figure 1.2: Example of Sherlock identifying a potential error in a dense spreadsheet

The main contributions of this thesis are as follows. First, we introduce a new compression framework that can be used to greatly reduce the memory usage of many modern spreadsheet systems. Second, we present a complementary visualization tool that can be used to automate the process of spreadsheet auditing and analysis.

We divide the rest of this thesis into the following chapters. In Chapter 2, we provide some technical background on how spreadsheet systems track dependencies between cells, explain different execution modes for formula computation, and dive deeper into the intuition for formula patterns. In Chapter 3, we discuss related work in spreadsheet compression and graph compression. We also examine several existing visualization tools for modern spreadsheet systems. In Chapter 4, we introduce TACO and provide the lower level details regarding its compression algorithm and system design. In Chapter 5, we provide a performance evaluation of TACO and compare its efficiency to a modern spreadsheet system such as Excel. Finally, in Chapter 6 we introduce Sherlock and examine some use-cases for its supported functionalities.

Chapter 2

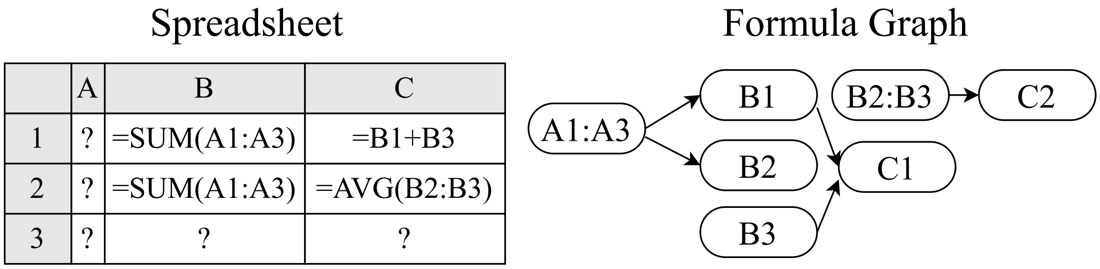
Technical Background

We now define a few pieces of useful terminology including formula graphs, execution models, and formula patterns. We also further discuss why current spreadsheet systems are limited in terms of scalability and usability tooling.

2.1 Formula Graphs

Spreadhseets provide the notion of formulae for users to perform programmatic transformations on their input data. A formula takes zero or more arguments as input and outputs a value. Formulas are therefore functions that perform some sort of operation on zero or more cells of the spreadsheet. Formulae can be composed together creating a formula graph, a directed acyclic graph (DAG) that captures the computational dependencies between cells. Each vertex in the graph represents a cell or range of cells. There exists an edge from vertex v_i to v_j if v_j depends on v_i. Figure 2.1 shows an example of such a formula graph.

Many modern spreadsheet systems such as Excel [5], Libre Office Calc [6], and ZK Spreadsheet [7] implement these formula graphs internally to identify cells that require re-computation when a cell is updated. When a cell is updated, two main operations must be performed to keep the spreadsheet UI consistent:



	A	B	C
1	?	=SUM(A1:A3)	=B1+B3
2	?	=SUM(A1:A3)	=AVG(B2:B3)
3	?	?	?

Figure 2.1: A sample spreadsheet and its corresponding formula graph

1. the dependents of the updated cell must be identified
2. the formula in each dependent cell must be recomputed

In other words, updating one formula cell requires a formula graph traversal to determine the set of affected formula cells followed by a re-calculation of all the affected cells. As we will show later, finding the dependents of an updated cell in a formula graph tends to be a time-consuming process due to the sheer size of the graph. Furthermore, if the size of the formula graph is too large to fit in memory or the updated cell has so many dependents that the re-computation becomes too computationally intensive, modern spreadsheet systems will begin to freeze or crash, which leads to scalability issues for large datasets. Therefore, querying the formula graph is a major bottleneck with most spreadsheet systems.

2.2 Execution Models

When a formula cell is updated, there are two primary approaches to dealing with the re-computation of dependent cells.

In a **synchronous execution model**, the spreadsheet system will freeze the entire UI until the result of the entire computation is completed. For simple spreadsheets, the work involved in updating a cell is small enough not to interfere with user interactivity. However, for larger scale spreadsheets, the traversal and re-computation time becomes nontrivial, and spreadsheet systems like Excel, which utilize a synchronous computation model, will freeze or become unresponsive until all cells are updated [3]. During this time, the spreadsheet is rendered unusable to the end user, and performing data analysis becomes a slow and inefficient process.

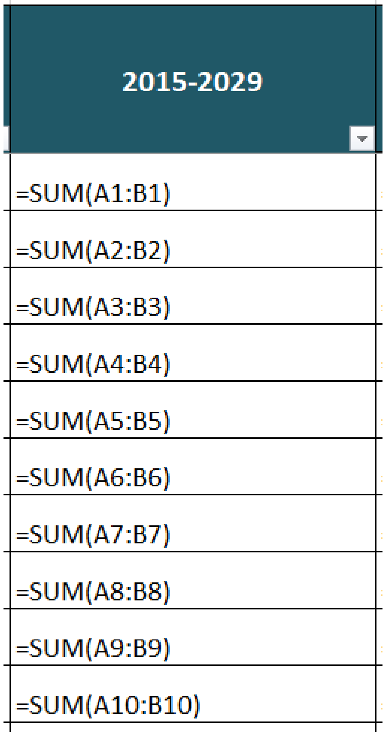
In an **asynchronous execution model**, the spreadsheet system returns control to the user after it has identified the dependents of the updated cell. The computation of the affected cells is then performed separately by another thread and the user can interact with the spreadsheet as normal. All cells that need to be updated will not be available to the user until the spreadsheet system computes its value. Unlike the synchronous approach, the user can receive control back much earlier and perform other data analysis tasks while their previous computation completes. As the system computes the value of each cell, they are displayed to the user one by one instead of being pushed to the UI all at once.

For either approach, dependency identification is a major component that directly affects the usability of the spreadsheet. An inefficient dependency identification algorithm can lead to severe interactivity issues and cause spreadsheets to remain non-responsive for an unacceptable amount of time. Our TACO framework can not only be used for either execution model but it can also provide more optimized support for dependency identification, leading to much more scalable and usable spreadsheet systems.

2.3 Formula Patterns

To address the computational inefficiencies of spreadsheets, we can apply compression techniques to the spreadsheet's formula graph thereby shrinking the size of the graph and achieving faster graph traversal times for large computations. In order to compress a formula graph, we need to identify some form of redundancy in the graph that we can take advantage of.

In our analysis, we found that many real-world spreadsheets contain patches of cells with formulae that follow a similar structure. Take Figure 2.2 for example, which is a portion of a real-world spreadsheet taken from the Github corpus described previously. In the figure, we have a column of formulae each of which references a range in another column. Each referenced range involves the same columns (e.g., A and B) and the row numbers follow a predictable sequence. More generally, we see that the column can be represented by a single pattern: the formula in row i can be derived by taking the formula in row $i-1$ and incrementing the row numbers of each cell referenced in the formula by 1. In the traditional formula graph representation, each of these cells would be represented using a single node and collection of edges in the graph despite their structures being the same. This leads to a dense formula graph that is computationally expensive to traverse. However, by taking advantage of the pattern described above, we can instead represent this redundant patch of cells as one node in the graph allowing for much smaller and robust formula graphs.



2015-2029
=SUM(A1:B1)
=SUM(A2:B2)
=SUM(A3:B3)
=SUM(A4:B4)
=SUM(A5:B5)
=SUM(A6:B6)
=SUM(A7:B7)
=SUM(A8:B8)
=SUM(A9:B9)
=SUM(A10:B10)

Figure 2.2: A patch of similar formulae observed in a real-world spreadsheet

Formula patterns such as the ones in Figure 2.2 tend to comprise large portions of typical spreadsheets and formulae in very close proximity to each other. As a result, it is possible to condense graphs with hundreds of thousands of nodes into much smaller ones by identifying these fundamental formula patterns. We use the term **tabular locality** to describe the phenomenon where cells appearing very close to each other tend to follow the same formula structure. Spreadsheets that exhibit a high degree of tabular locality tend to be more compressible since their formula patterns span large portions of the sheet. We will dive deeper into these formula patterns in an upcoming section.

Chapter 3

Related Work

In this section, we include a brief survey of solutions that have attempted to solve scalability and usability issues with spreadsheet systems.

3.1 Spreadsheet Computation

There are several existing solutions that aim to help users understand formula graphs by tracking dependencies [5, 8, 6, 9]. As mentioned in the introduction, Excel has a built-in dependency tracker that can allow users to find the precedents and dependents of a cell [10]. This can be useful for spreadsheets with simple formula graphs, but as illustrated in the introduction, most real-world spreadsheets have complex graphs with hundreds of thousands of connections making Excel's tracker less of a viable option if the user desires a concise and high-level overview of the spreadsheet's formula graph.

3.2 Graph Compression

As far as we know, there are currently no spreadsheet systems that utilize some form of graph compression on the internal formula graph. While graph compression is by no means a new topic, much research on graph compression has been devoted to other applications such as biological networks, Web graphs, and social networks [11, 12, 13]. Our research also suggests that existing solutions do not make any clever use of tabular locality or compression methods to optimize formula computation in any way even if a majority of the spreadsheet has repetitive structures.

3.3 Spreadsheet Usability

There are many existing tools aimed at helping users make sense of their spreadsheets. For example, Excelint is an Excel add-in that identifies errors in spreadsheets [14]. The tool helps users perform much more efficient auditing on spreadsheet systems, and in this regard it performs its intended function well. However, in terms of scalability, the tool does not focus on optimizing spreadsheet calculations, so it is subject to the same re-computation issues as Excel. Furthermore, while the tool does help users debug their spreadsheets, it is not capable of giving users a human-friendly view of the dependencies between cells. As a result, the tool only tackles a small subset of the spreadsheet visualization problem.

Another tool aimed at helping users make sense of their spreadsheets is Perquimans [15]. Perquimans is tool that can help users visualize spreadsheet function combinations. Unlike Excelint, Perquimans is capable of breaking down formula computations, which can be useful in understanding how computations are arranged on the sheet. However, it does not help identify the dependents or precedents of a cell, so it is much less useful in providing users with a high-level overview of what needs to be re-computed when a cell is updated. Perquimans does not perform any sort of compression on their formula visualizations either, so in terms of scalability the tool also falls short in handling larger-scale datasets.

3.4 Conclusion

As shown above, there are very few frameworks or tools that attempt to:

1. optimize spreadsheet computations
2. help users visualize formula graphs in a human-readable way

To this end, we attempt to extend the state of the art by offering a practical framework that is capable of scaling up to the size of modern datasets while fulfilling the needs of breaking down complex spreadsheets in a more visually interpretive way for users.

Chapter 4

System Overview

TACO is a framework for compressing formula graphs and quickly querying the dependencies of a particular cell. Below we describe the different aspects that comprise the TACO framework.

4.1 Fundamental Patterns

As alluded to in Chapter 2, spreadsheets can be decomposed into a series of fundamental patterns and these patterns can be used to compress the formula graph of a spreadsheet. In this section, we review the fundamental patterns that we discovered from our data analysis. For simplicity, we only consider the case of adjacent formulae appearing in one column. The row-wise case can be derived symmetrically.

4.1.1 Fixed and Relative Relationships

In order to describe the following patterns more formally, we introduce the concept of fixed and relative relationships between cells. Before diving into these relationships, we first introduce the following definitions. Suppose we have a cell range A1:B1. We will refer to $A1$ as the *head cell* and $B1$ as the *tail cell* of the range. Furthermore, we will sometimes describe the positions of cells using an x-y coordinate system. In our framework, we can write $A1$ as the coordinate pair (1, 1) and $B1$ as (2, 1).

With that in mind, consider a collection of formula cells where each references one cell range. We call the relationship between the formula cell and its referenced head / tail cell a **fixed** relationship if any of the following are true:

1. each referenced head cell is the same
2. each referenced tail cell is the same
3. the referenced head and tail cell are the same

In a **relative** relationship, we can derive the referenced range of each formula cell by using a single x-y coordinate offset. In other words, given two cells' positions $\vec{u} = (x_1, y_1)$ and $\vec{v} = (x_2, y_2)$ and an offset (c, r), we say $\vec{u}$ is relative to $\vec{v}$ if $x_2 = x_1 + c$ and $y_2 = y_1 + r$.

4.1.2 Representing Fixed and Relative Relationships

In a compressed formula graph, we can store information about fixed and relative relationships as part of the metadata for an edge. More specifically, we use four variables to capture the fixed and relative relationships between cells. We again consider a collection of formula cells that each reference one cell range.

1. If each formula cell in the collection references one range, and each range has the same **head** cell, we use $hFix$ to denote the x-y coordinate of the head cell.

2. If each formula cell in the collection references one range, and each range has the same **tail** cell, we use $tFix$ to denote the x-y coordinate of the tail cell.

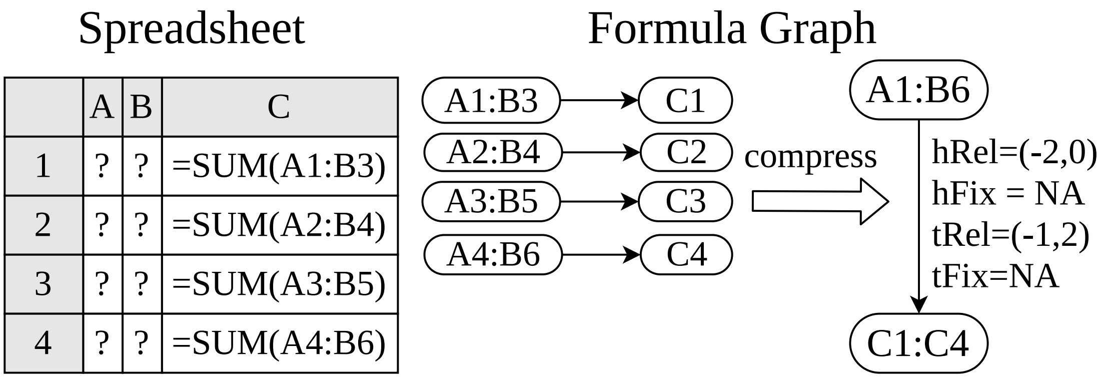
3. If each formula cell in the collection references one range, and the **head** cell of each range is relative the location of the formula cell by (c, r), then we use $hRel$ to denote the (c, r) offset.

4. If each formula cell in the collection references one range, and the **tail** cell of each range is relative the location of the formula cell by (c, r), then we use $tRel$ to denote the (c, r) offset.

Now that we've defined fixed and relative relationships and further described how to encode these in the edge metadata for a compressed formula graph, we now review the fundamental patterns that can be derived from these relationships.

4.1.3 Relative plus Relative (RR)

We classify a collection of formula cells and their referenced ranges as an RR pattern if for each referenced range, the head cell and tail cell of the range are relative to the formula cell by the same offsets $hRel$ and $tRel$ respectively. For example, in Figure 4.1 observe that each formula cell in column C is relative to the head cell of its referenced range by $(-2, 0)$ (i.e., to the left by two columns) and relative to the tail cell by $(-1, 2)$. The edge metadata for this example is $meta = (hRel = (-2, 0), hFix = NA, tRel = (-1, 2), tFix = NA)$ where NA means that the particular parameter is not applicable to the given pattern. With the metadata defined, we can now represent this graph with one compressed edge: $(prec = A1 : B6, dep = C1 : C4, pattern = RR, meta)$.

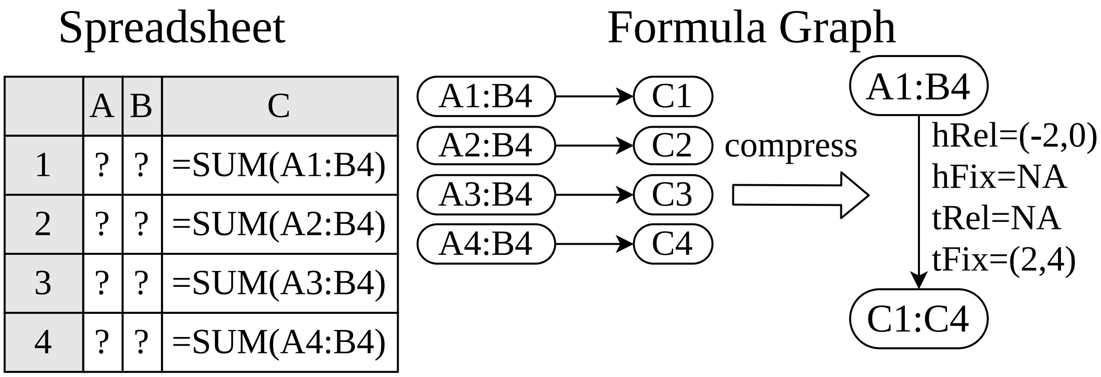


	A	B	C
1	?	?	=SUM(A1:B3)
2	?	?	=SUM(A2:B4)
3	?	?	=SUM(A3:B5)
4	?	?	=SUM(A4:B6)

Figure 4.1: An example RR pattern

4.1.4 Relative plus Fixed (RF)

A collection of formula cells and their referenced ranges is classified as an RF pattern if all tail cells of the referenced range are the same, and the head cell of each referenced range is relative to the formula cell by the same offset $hRel$. For example, in Figure 4.2, note that each formula cell in column C is relative to the head cell of its referenced range by $(-2, 0)$ (i.e., to the left by two columns) and points to a fixed tail cell $B4 = (2, 4)$. The edge metadata for this example is $meta = (hRel = (-2, 0), hFix = NA, tRel = NA, tFix = (2, 4))$ and we can represent this graph with one compressed edge: $(prec = A1 : B4, dep = C1 : C4, pattern = RF, meta)$.

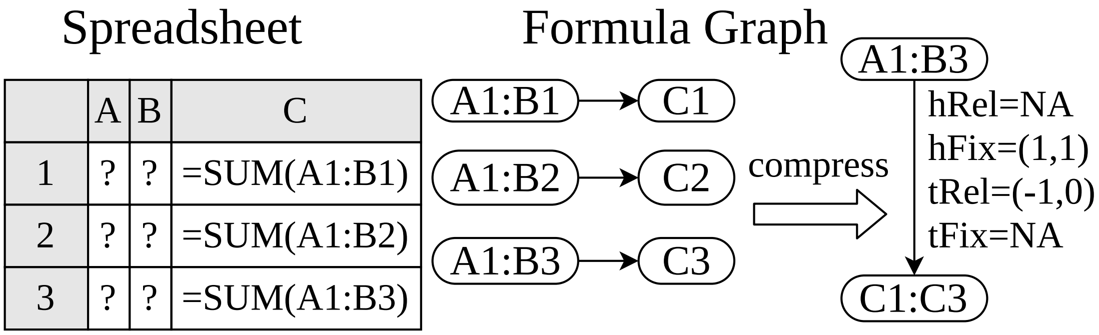


	A	B	C
1	?	?	=SUM(A1:B4)
2	?	?	=SUM(A2:B4)
3	?	?	=SUM(A3:B4)
4	?	?	=SUM(A4:B4)

Figure 4.2: An example RF pattern

4.1.5 Fixed plus Relative (FR)

The FR pattern is symmetric to the RF pattern. That is, in an FR pattern, all head cells of the referenced range are the same, and the tail cell of each referenced range is relative to the formula cell by the same offset $tRel$. Using the same reasoning in the preceding sections, we see that for Figure 4.3 the metadata of the compressed edge is $(hRel = NA, hFix = (1, 1), tRel = (-1, 0), tFix = NA)$.

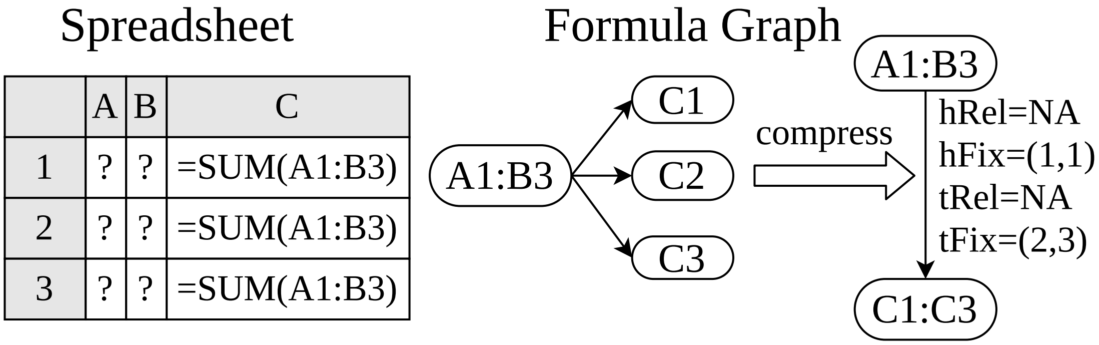


	A	B	C
1	?	?	=SUM(A1:B1)
2	?	?	=SUM(A1:B2)
3	?	?	=SUM(A1:B3)

Figure 4.3: An example FR pattern

4.1.6 Fixed plus Fixed (FF)

The FF pattern describes the situation where for each referenced range, the head cell and tail cell of the range are the same. Using the same reasoning in the preceding sections, we see that for Figure 4.4 the metadata of the compressed edge is $(hRel = NA, hFix = (1, 1), tRel = NA, tFix = (2, 3))$.



	A	B	C
1	?	?	=SUM(A1:B3)
2	?	?	=SUM(A1:B3)
3	?	?	=SUM(A1:B3)

Figure 4.4: An example FF pattern

4.2 Formula Graph Compression Algorithm

Now that we've defined the patterns that TACO is capable of identifying, we can move onto describing the algorithm used to compress a formula graph. At a high level, the algorithm takes a list of dependencies between formula cells and their referenced ranges as input and greedily builds the dependency graph. On each iteration, the algorithm inserts and partitions the dependencies such that the number of edges in the output graph is minimized. It makes use of the compression patterns described previously and also applies several heuristics based on our analysis of real-world spreadsheets to decide on the optimal insertion.

It's important to note that the formula patterns mentioned above do not necessarily need to be adjacent to be compressed. If there exists multiple pattern types that are separated by a fixed number of cells, then our algorithm can still perform compression. In the event that multiple pattern types are separated by a fixed number of cells, we will refer to the number of cells between each pattern type as the **gap size**.

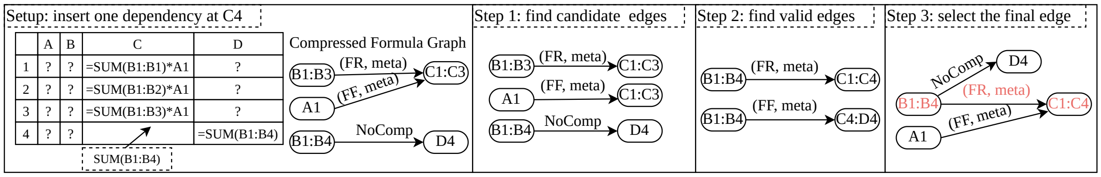
The gap size between patterns can be used as a constraint to quickly find candidate edges that the input dependency can be compressed into. For instance, if the gap size is 0, we only consider adjacent formula cells. Observe that smaller gap sizes are much more efficient in compressing dependencies, so our algorithm prioritizes patterns with smaller gap sizes. If there are multiple edges a dependency can be compressed into, we use several heuristics based on our analysis of real-world spreadsheets to decide the edge that can best reduce graph sizes (e.g., we prioritize column-wise compression over row-wise since the former is more common).

With these intuitions in mind, Algorithm 1 shows our approach for compressing one dependency $e^{'} = (prec, dep)$ into a compressed formula graph G.

Algorithm 1: Compressing a dependency $e^{'}$ into $G(E, V)$

```
Algorithm compressDep($G(E, V)$, $e^{'}$)
    $isCompressed \leftarrow$ false
    for $g \in [0, MAX\_GAP]$ do
        $pSet \leftarrow$ find patterns for the selected gap $g$
        $eSet \leftarrow$ find all $e \in E$ whose $e.dep$ is $g$ cells
                away from $e^{'}.dep$ on column or row axis
        for $candE \in eSet$ do
            $edgePairs \leftarrow$ genCompEdges($candE$, $e^{'}$, $pSet$)
            $edgePairSet$.add($edgePairs$)
        end
        if $edgePairSet$ is not empty then
            $edgePair \leftarrow$ sort $edgePairSet$ by heuristics and take the first
            maintain $G$ using $edgePair$
            $isCompressed \leftarrow$ true
            break
        end
    end
    if $isCompressed$ is false then
        insert $e^{'}$ into $G$
    end
Procedure genCompEdges($candE$, $e^{'}$, $pSet$)
    if $candE.p$ ==NoComp then
        for $p \in pSet$ do
            $pair \leftarrow (p.$compressDep$(candE, e^{'}), candE)$
            $edgePairs$.addIfValid($pair$)
        end
    end
    else
        $pair \leftarrow (candE.p.$compressDep$(candE, e^{'}), candE)$
        $edgePairs$.addIfValid($pair$)
    end
    return $edgePairs$
```

Note that the algorithm iterates through patterns with a gap size varying from 0 to MAX_GAP, where MAX_GAP is the maximum gap across all predefined patterns. This parameter is configurable and for our experiments we have this set to 7. To further illustrate the algorithm, we will use Figure 4.5.



	A	B	C	D
1	?	?	=SUM(B1:B1)*A1	?
2	?	?	=SUM(B1:B2)*A1	?
3	?	?	=SUM(B1:B3)*A1	?
4	?	?		=SUM(B1:B4)

Figure 4.5: An example of the TACO compression algorithm

In the figure, each formula cell in column C references two ranges. The references to column B follow the FR pattern and the references to column A follow the FF pattern. Furthermore, we have an uncompressed edge where $B1 : B4$ references $D4$. Now suppose we insert $SUM(B1 : B4)$ at $C4$. In other words, say we want to compress the dependency $e^{'} = (B1 : B4, C4)$ and the gap size is $g = 0$. In this case, we have multiple patterns with the same gap size, and our algorithm will perform three steps to compress the graph.

Find candidate edges:

In this step, we use the gap constraint to find candidate edges that the dependency $e^{'}$ can be compressed into. More specifically, an edge e is a candidate edge if $e.dep$ is g cells away from $e^{'}.dep$ row-wise or column-wise. In step 1 of Figure 4.5, observe that all three edges qualify as candidate edges since $e^{'}.dep = C4$ is adjacent to both $C1 : C3$ and $D4$.

To find these edges, we first shift $e^{'}.dep$ by $g + 1$ cells in all four directions (i.e. up, down, left, and right) and use a spatial index on the vertices (e.g., an R-Tree [16]) to efficiently find ranges that overlap with the shifted $e^{'}.dep$. For each overlapping range (e.g. $D4$), we find its precedents (e.g. $B1 : B4$) and add this edge (e.g. $B1 : B4 \rightarrow D4$) into the candidate edge set.

Find valid candidates:

Once we collect a list of candidate edges, we need to verify if $e^{'}$ can be compressed into each candidate edge using the compressDep$(e, e^{'})$ routine. There are two cases to consider depending on whether the candidate edge is already compressed. If the candidate edge *candE* is not compressed, then we check whether $e^{'}$ and *candE* can be compressed into a new edge *newEdge* using the predefined patterns. If this is possible, we store *newEdge* as a valid candidate edge. If *candE* is a compressed edge, we check whether $e^{'}$ can be compressed into *candE* and if this is possible, we generate a valid edge. Step 2 in Figure 4.5 shows two valid compressed edges because the edge $B1 : B4 \rightarrow C4$ can be compressed into $B1 : B3 \rightarrow C1 : C3$ or $B1 : B4 \rightarrow D4$.

Select the final edge:
In the final step, we select an optimal valid candidate from the preceding step. The algorithm's selection is based on two heuristics. For the first heuristic, valid candidate edges that lead to column-wise compression are prioritized over those which lead to row-wise compression. If this heuristic does not return a single edge, then we further compare the priority of each remaining edge's pattern based on the probability they appear in real-world datasets (these probabilities are computed offline). In step 3 of Figure 4.5, we choose the compressed edge ($B1 : B4 \rightarrow C1 : C4$) over ($B1 : B4 \rightarrow C4 : D4$) because the former one uses column-wise compression. Finally, we delete the old edge from the original formula graph and insert the newly compressed edge.

4.3 Querying the Compressed Formula Graph

Once TACO creates a compressed version of the graph, the framework can provide more optimized usage for finding the dependents of a particular cell, which is a fundamental operation for maintaining the interactivity of the spreadsheet. In this section, we describe in more detail the algorithm that TACO uses to quickly query the dependents of a cell.

In order to find the dependents of a range r in our compressed dependency graph G, we use a slightly modified version of the Breadth-First-Search (BFS) algorithm. In the traditional algorithm, we only need to query the direct dependents or

Algorithm 2: Find deps of a column/row of cells r in $G(E, V)$

```
initialize $Queue$ as a queue containing only $r$
initialize $explored$ as an empty set
while $Queue$ is not empty do
    $precU \leftarrow$ remove the first element in $Queue$
    $precRanges \leftarrow$ overlapping ranges of $precU$
    for $prec \in precRanges$ do
        $edges \leftarrow \{e : e \in E$ and $e.prec = prec\}$
        for $e \in edges$ do
            $depUSet \leftarrow e.p.findDep(e, precU)$
            if explored does not contain depUSet then
                add $depUSet$ to $explored$
                add $depUSet$ to $Queue$
            end
        end
    end
end
return $explored$
```

neighbors of a vertex v in the graph. However, in our setting we need to include all the vertices in G that intersect r in order to find all the dependents of r. Another slight variation on the traditional algorithm is that edges may either be compressed or uncompressed. If an edge e is compressed, and want to find the dependents of a range r in e, then we need to ensure that we're retrieving all dependents of r not just a subset of them. Algorithm 17 shows the pseudo code for the modified BFS algorithm. The algorithm takes a column or row of cells r as input and returns the set of ranges that depend on r.

To further break down this algorithm we consider the more concrete example provided in Figure 4.6

	A	B	C	D
1	?	?	=SUM(B1:B1)+A1	?
2	?	?	=SUM(B1:B2)+A1	?
3	?	?	=SUM(B1:B3)+A1	?
4	?	?	=SUM(B1:B4)	=SUM(B1:B4)

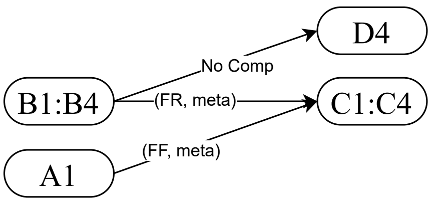


Figure 4.6: An example of finding dependents in a TACO graph

Suppose we'd like to query the dependents of cell B2. First, our algorithm stores B2 in a queue, which will store the ranges to be visited. For each range $precU$ in this queue, we find its direct dependents, and as mentioned earlier, we need to consider all the ranges that overlap with $precU$, which in this example is B1:B4. Next, we find the direct dependents of each overlapping range and the corresponding edges like in the original BFS algorithm. In this case, we have $B1 : B4 \rightarrow C1 : C4$ and $B1 : B4 \rightarrow D4$. Note that the edge $B1 : B4 \rightarrow C1 : C4$ can be compressed, so we need to uncover the actual direct dependents within this edge. This functionality is provided by the function $findDep(e, precU)$ function that we will describe shortly. For the inputs $B1 : B4 \rightarrow C1 : C4$ and B2, we return C2:C4 for the edge since C1 does not depend $B2$. We add the real dependents, C2:C4, to the queue if they are not yet visited and repeat the process until the queue is empty.

Next we move onto how the $findDep$ procedure works. At a high level, $findDep(e, r)$ finds the dependents d of a range r that is contained in e. The algorithm for $findDep(e, r)$ varies slightly between the RR pattern and the RF, FR, and FF patterns. We cover the RR pattern first. Suppose we have a compressed edge $e = (prec, dep, pattern = RR, meta)$ and we'd like to find the dependents d of a range of cells r that are contained in e. To determine d, we need to find its head cell and tail cell which we denote d_h and d_t respectively. Since the edge follows an RR pattern, each cell's precedent in d forms a sliding window on $e.prec$ like the one shown in Figure 4.7.

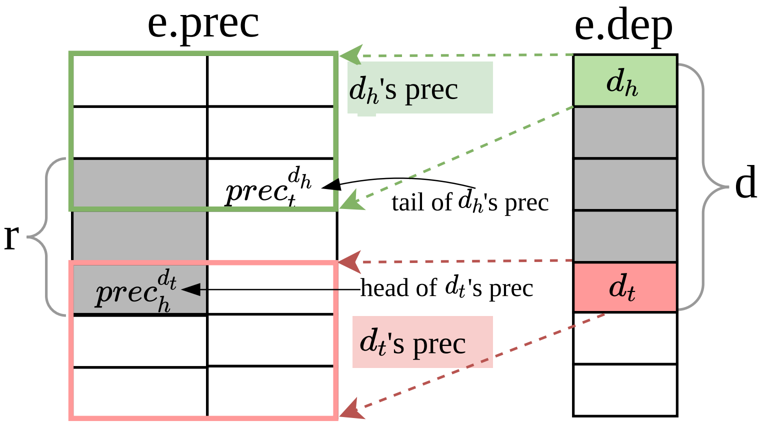


Figure 4.7: An example of finding dependents for the RR pattern

To compute d_h, observe that the top row of r must intersect with the bottom row of d_h's precedent. Similarly, the bottom row of r must intersect the top row of d_t's precedents. As a result, we can back calculate d_h and d_t based on r. More specifically, we use the following invariant to compute d_h,

$$d_h = prec_t^{d_h} - tRel$$

where $prec_t^{d_h}$ is d_h's precedent's tail cell and $tRel$ is the relative position of d_h with respect to $prec_t^{d_h}$. Recall that $tRel$ is known since it is encoded in the metadata of a compressed edge. Thus, the remaining task is to compute $prec_t^{d_h}$. We know that $prec_t^{d_h}$ is in the bottom row of d_h's precedent since it is a tail cell and that the bottom row of d_h's precedent intersects the top row of r. Therefore, $prec_t^{d_h}$ is in the top row of r and its row index is the same as the row index of r's head cell (i.e., $r.head.row$). Since $prec_t^{d_h}$ is a tail cell, it is in the right-most column of $e.prec$, so its column index is $e.prec.tail.col$. Hence, $prec_t^{d_h} = (e.prec.tail.col, r.head.row)$ and we have d_h. To find d_t we follow a similar procedure. In this case, we have the invariant

$$d_t = prec_h^{d_t} - hRel$$

where $prec_h^{d_t}$ is d_t's precedent's head cell and $hRel$ is the relative position of d_t with respect to $prec_h^{d_t}$. As shown in Figure 4.7, $prec_h^{d_t}$ should be in the last row of r and in the left-most column of $e.prec$. Therefore, we have $prec_h^{d_t} = (e.prec.head.col, r.tail.row)$. Note this procedure can output a range d that is beyond $e.dep$. In this case, we take the intersection between d and $e.dep$ to return a valid range.

For the RF, FR, and FF patterns, $findDep(e, r)$ works similarly. As with the RR pattern, we need to find d's head cell d_h and tail cell d_t. To compute d_h, we use the intuition shown in Figure 4.8

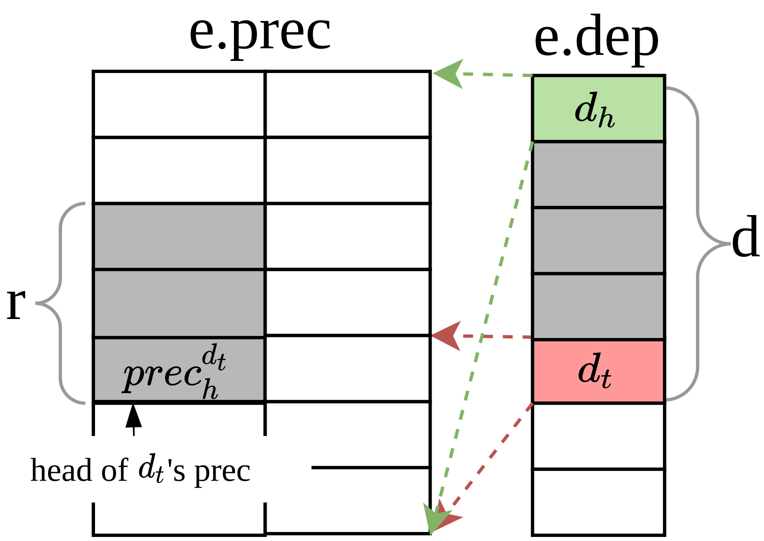


Figure 4.8: An example of finding dependents for the RF, FR, and FF patterns

For the RF, FR, and FF patterns, $e.dep.head$ references the entire range of $e.prec$ and is the dependent of any r contained in $e.prec$. Consequently, $e.dep.head = d_h$. To compute d_t, we use the fact that

$$d_t = prec_h^{d_t} - hRel$$

where $prec_h^{d_t}$ is the head cell of d_t's precedents. Recall that $hRel$ is known since we are given e, so we only need to compute $prec_h^{d_t}$. Using the same intuition as the RR case, we note that $prec_h^{d_t}$ is in the bottom row of r, so the row index of $prec_h^{d_t}$ is $r.tail.row$ and the column index is $e.prec.head.col$ since $prec_h^{d_t}$ is a head cell.

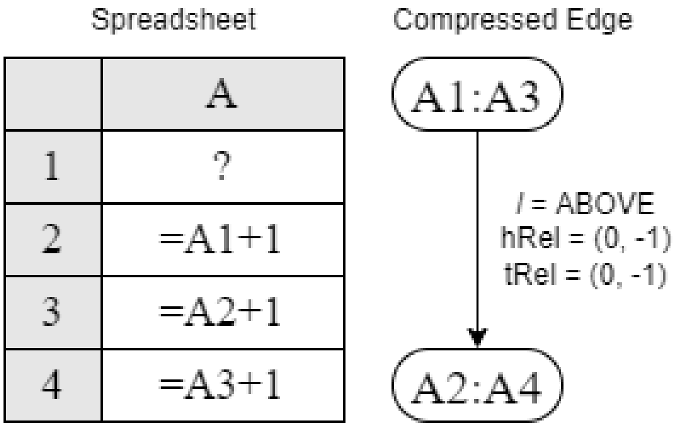
4.4 Extensions

In addition to the fundamental patterns described previously, there is another special case pattern that is typically found in real-world spreadsheets. Oftentimes, users wish to compute some sort of commutative total on the input data. In this scenario, we have a column of formula cells where each formula in the column either references the cell directly above or below it creating a chain of dependencies.

While we could group this structure under the RR pattern, the compressed edge we would obtain is not optimal for dependency identification. To understand why, consider Figure 4.9. In the figure, each formula cell starting from A2 increments the value of the above formula cell by one. Suppose A1 is updated and we now need to find its dependents. If we were to use the $findDep(e, r)$ functioned we defined previously for RR, we would only return A2 as the dependent of A1. Our algorithm would then repeat this process for A2, then A3, and so on until we reach the end of the chain. Instead, it would be much more efficient to simply return a range that encompasses all the cells in the chain. Fortunately, the RR-chain pattern can help accomplish this.

We introduce a new pattern called the RR-chain which is a special case of the RR pattern. Along with standard edge metadata that the four fundamental patterns posses, the RR-chain edge metadata consists of a variable l to indicate the dependency direction of the chain. For example, l is ABOVE in 4.9 since each formula cell references the adjacent cell above it. The following discussion focuses on the case where $l = ABOVE$. The case for $l = BELOW$ is symmetric.

To compress a dependency $e^{'}$ into e for RR-chain, we first check if the edge follows an RR pattern. If it does then we further check whether $e^{'}.prec$ is adjacent to and above $e^{'}.dep$. If this is the case, then to find the dependents of a range r, we return a range d between $r.head$'s direct dependent and the tail cell of $e.dep$. For example, if we wanted to find the dependents of A2 in Figure 4.9, we can now return the range between A3 (i.e., A2's direct dependent) and the tail cell of $e.dep$ (i.e. A4) rather than iteratively collect dependents for the entire chain one by one.



	A
1	?
2	=A1+1
3	=A2+1
4	=A3+1

Figure 4.9: An example RR-chain pattern

Chapter 5

Performance Evaluation of TACO

At this point, we have covered the basic TACO patterns used to compress formula graphs, the types of operations that can be performed on the compressed formula graph, and the compression algorithm used to transform a traditional formula graph graph to a compressed version. In this chapter, we evaluate TACO by examining several metrics. In particular, we will compare the TACO framework's performance with Excel, another popular spreadsheet system and we will also examine how TACO helps alleviate issues with interactivity and memory.

5.1 Background

The following evaluations are performed using DataSpread [17], an open-source spreadsheet system. DataSpread uses an asynchronous computation model whereby control is returned to users once dependencies of an updated cell are identified. It also follows a frontend-backend architecture with ZK Spreadsheet [8] as its formula engine. Users can interact with Dataspread using a web-based spreadsheet UI and computations make use of PostgreSQL on the backend side. For the formula graph implementation, we have two versions: TACO and the traditional, uncompressed formula graph implemented as an adjacency list which we denote **NoComp**.

The following experiments are run on a 12 GB RAM machine with an Intel i7-10750H CPU with 6 physical cores. For all tests excluding the one involving the comparison with Excel, we use Ubuntu 20.04. For the comparison between TACO and Excel, we use a Windows 10 machine. Each experiment with Excel is performed using a single thread and we report the average of three test re-runs.

Our evaluation is performed on two real-world spreadsheet datasets. The first one is the Enron dataset [18] with 17K xls files. In our pre-processing, we removed spreadsheets that caused exceptions (e.g. requiring password) or were too small (i.e. fewer than 10 dependencies). This left us with 7.4K xls files. The second dataset includes a collection of 7.8K xlsx files webscraped from Github. We filtered the original dataset such that workbooks with less than than 10 KB (or caused exceptions) were removed leaving 5.4K xlsx Github files. In total, our evaluation encompasses a collection of 12.8K xls and xlsx files.

5.2 Storage Savings

In this section, we compare the memory consumption of TACO with that of the traditional formula graph. We first analyze the reduction of the total number of vertices and edges for all formula graph across the Enron and Github datasets. he results are summarized in Table 5.1. As shown in the table, TACO can significantly decrease the size of a traditional formula graph. For example, the total number of edges across the Github dataset decreased by about 98%.

	Enron		Github	
	Vertices	Edges	Vertices	Edges
No Compression	27.1M	36.1M	176.3M	234.6M
TACO	3.0M	2.4M	4.7M	3.8M

Table 5.1: Formula graph sizes after TACO compression

Next, we analyzed all sheets individually. Table 5.2 shows that there exists a spreadsheet in the Enron dataset that TACO can reduce by 875K edges and one in the Github dataset that TACO can reduce by 3.1M edges. The average edge reduction by TACO is 4.5K and 42K for the Enron and Github datasets respectively.

	Max	75th per.	Median	Mean
Enron	875,286	1,680	258	4,544
Github	3,141,054	26,627	4,579	42,460

Table 5.2: Number of edges reduced by TACO (high is better)

Now we examine memory savings by pattern type. The results are summarized in Table 5.3. In the table, note that show that the RR and FF patterns compress the most edges. The RR pattern is capable of reducing the total number of edges by more than 22M and 148M for the Enron and Github datasets respectively. The FF pattern reduces more than 4M and 24M edges in total for the two respective datasets. Other patterns also reduce a significant number of edges in some spreadsheets. The RF and FR patterns, even though not as common, can reduce up to around 10K and 39K edges for a single spreadsheet in the Enron and Github datasets respectively. These results show that TACO's patterns are not only commonly used in real-world spreadsheets but can also significantly reduce the graph sizes leading to much more robust spreadsheet systems.

Pattern	Enron Total	Enron Max	Github Total	Github Max
RR	22,014,883	525,026	148,392,334	2,094,936
RF	2,820	1,413	13,514	9,999
FR	158,207	13,815	204,107	39,008
FF	4,258,701	174,948	24,291,075	1,043,702
RR-Chain	717,788	24,596	6,106,323	399,996

Table 5.3: Number of edges reduced by each pattern (high is better)

Finally, we examine the TACO compression benefits for a subset of the most complex spreadsheets in the Enron and Github datasets. For each spreadsheet in the corpus, we found the cell that has the greatest number of direct dependents and the cell that has the longest chain of dependents. Once these were identified, we extracted the top 3 spreadsheets with the longest dependency chain and the top 5 spreadsheets that have the max number of direct dependents for each dataset. We also ensured that the group of 5 spreadsheets with the greatest number of direct dependents did not contain any spreadsheets that were also included in the group of 3 spreadsheets with the longest dependency chains for both the Enron and Github datasets. The group of 5 spreadsheets with the greatest number dependents includes more diverse patterns, so we included more sheets than the longest dependents group. The compression results are shown in Table 5.4. Note that we renamed the original spreadsheet files to show the dataset and group that the file belongs to.

Filename	Original Edges	TACO Edges	Source Cell	Num of Deps	Longest path
enron_max1.xls	876,020	734 (0.084%)	Z1	175,496	184
enron_max2.xls	124,629	1,574 (1.263%)	J5	34,481	4
enron_max3.xls	114,101	1,534 (1.344%)	J5	31,846	4
enron_max4.xls	205,218	1,152 (0.561%)	D4	29,029	17
enron_max5.xls	50,540	1,342 (2.655%)	B42	23,749	5
enron_long1.xls	312,642	35 (0.011%)	F2	20,846	6,952
enron_long2.xls	4,290	1 (0.023%)	A6	4,290	4,290
enron_long3.xls	17,525	12 (0.068%)	I7	4,381	2,191
github_max1.xlsx	2,353,646	14,204 (0.603%)	S82	304,082	22
github_max2.xlsx	441,242	14 (0.003%)	C6	189,106	2
github_max3.xlsx	547,305	73 (0.013%)	R5820	156,385	3
github_max4.xlsx	350,400	8 (0.002%)	D21917	140,156	4
github_max5.xlsx	569,712	12 (0.002%)	A8	131,455	43,819
github_long1.xlsx	400,006	4 (0.001%)	B2	199,999	199,999
github_long2.xlsx	131,532	1 (0.001%)	A8	131,532	131,532
github_long3.xlsx	124,534	1 (0.001%)	A2	124,534	124,534

Table 5.4: TACO compression results for most complex spreadsheets

For each spreadsheet s, the *Source Cell* column shows the cell that has the max number of dependents or the longest dependency path. The last two columns show the number of direct dependents and the length of the longest dependency chain for the source cell.

Observe that many of the spreadsheets in Table 5.4 have very complex dependency structures. For example, the source cell of enron_max1 has more than 175K dependents and has a chain of dependents spanning 184 cells. github_max1 has a cell with 304K direct dependents making it the spreadsheet with the most direct dependents, and github_long1 has a cell with a dependency chain of 200K cells long making it the spreadsheet with the longest dependency chain. Regardless of these complexities, we find that TACO can significantly reduce the size of formula graphs. The most notable decrease occurs for enron_max1, which was reduced from 876K edges to 734 edges (99.916% decrease).

5.3 Interactivity

Next, we examine the extent to which TACO is capable of reducing the time of controlling control to users. We use the spreadsheets from Table 5.4 and update the corresponding source cell. After the source cell is updated, we test the time for returning control to users. The time for returning control mainly depends on the time for finding the dependents of the source cell in the formula graph. If a test does not finish within 30 mins, we mark it with *DNF*. The results are shown below in Table 5.1.

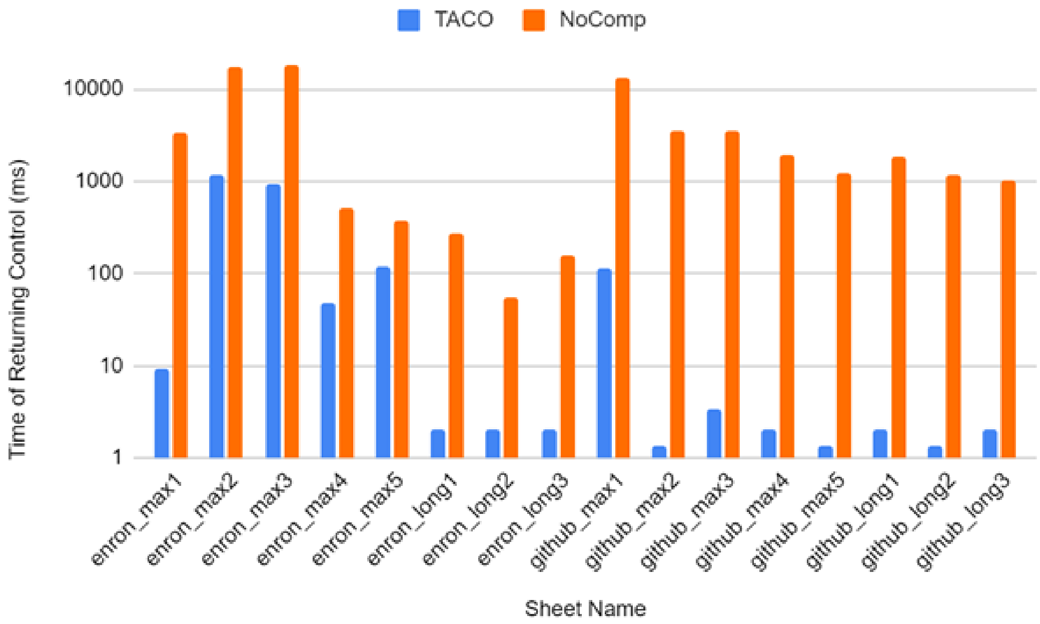


Figure 5.1: Time of Returning Control to Users

5.4 Comparison with Excel

To wrap up our evaluation, we compare TACO with a popular commercial spreadsheet system, Excel. For Excel, we use VBA macros for finding the dependents of a cell [19]. We use the spreadsheet files in Table 5.4 and report the time for finding the dependents of the source cell. The results are shown in figure 5.2. Observe that Excel does not finish for github_max5, github_long1, and github_long3 whereas TACO can finish for all spreadsheets within 1000 ms. While TACO is slower than Excel for two spreadsheets (enron_max2 and enron_max3), it is up to around 850K times faster than Excel for a majority of the other spreadsheets (e.g. github_long2).

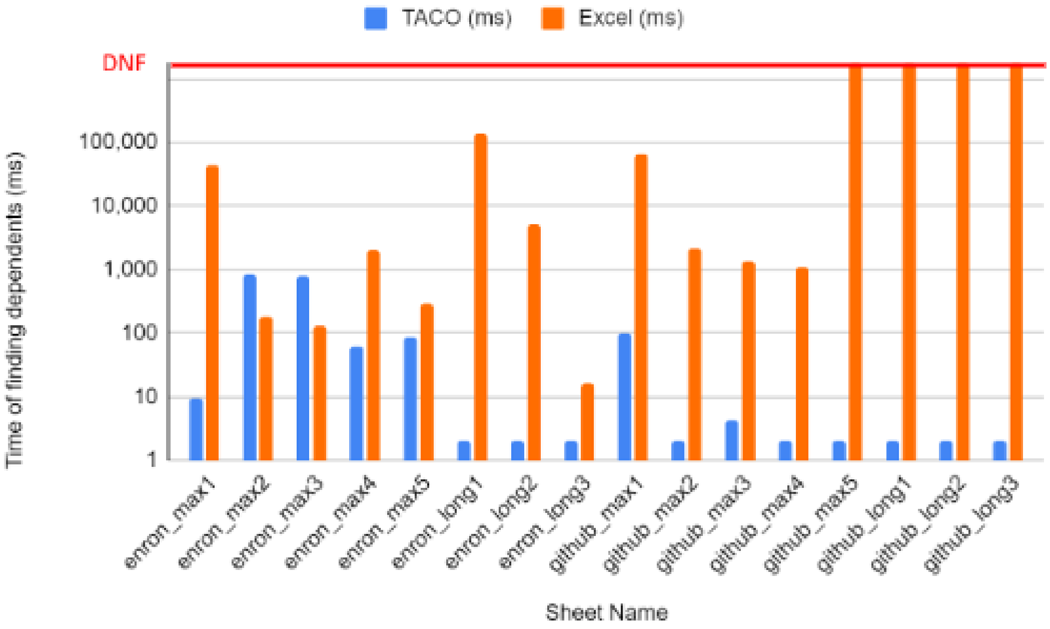


Figure 5.2: Performance comparison between TACO and Excel

Chapter 6

Visualizing Formula Graphs with Sherlock

In Chapter 4, we introduced the TACO framework and discussed its dependency identification and compression algorithms, which are crucial to tackling scalability issues with current spreadsheet systems. In this section, we introduce Sherlock, an Excel add-in that utilizes the functionalities of TACO to help users understand formula graphs more effectively.

6.1 System Overview

Sherlock follows a frontend-backend design scheme as shown in Figure 6.1.

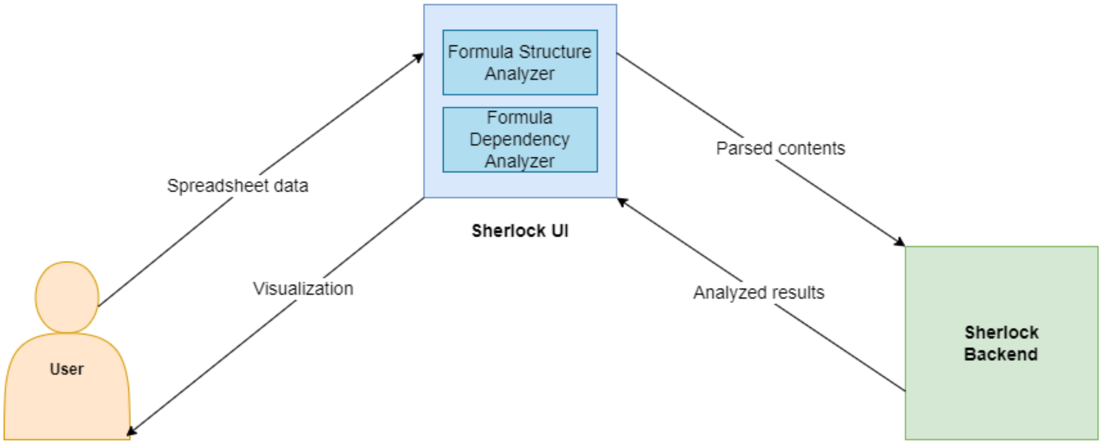


Figure 6.1: A high level overview of Sherlock

The tool is capable of performing two main types of analyses. On one hand, Sherlock can generate visualizations based purely on the structure or the template of a formula. This type of analysis does not take cell references into account, but rather groups formulae together based on their superficial template. On the other hand, Sherlock can generate visualizations by taking cell references into account while ignoring any information regarding the structure of the formula. The combination of these two types of analysis can be considered an area for potential future work.

In order to support the two types of analyses above, Sherlock contains several core components, which we briefly introduce next. For analyzing formula structure, Sherlock has an analyzer that can cluster similar formulae based on a formula's string representation, excluding any of its references or constants. For formula dependency analysis, Sherlock contains a separate analyzer that can identify TACO patterns in the selected range using algorithms described in Chapter 4. To complement dependency analysis, Sherlock also comes with a tool for displaying the TACO graph representation of the selected range. As we will see in the next sections, these visualizations provide a much more concise overview of large repetitive ranges of a spreadsheet, which in turn allows users to examine traditional formula graphs in a much more efficient and automated way.

Each of the functionalities mentioned previously is aimed at addressing common usability needs with spreadsheet systems such as formula auditing [20], managing the sheer volume of cells [21], and much more. Below we provide a more in depth explanation of what each functionality does and how they help resolve the usability issues described in the introduction.

6.2 Formula Template Visualization

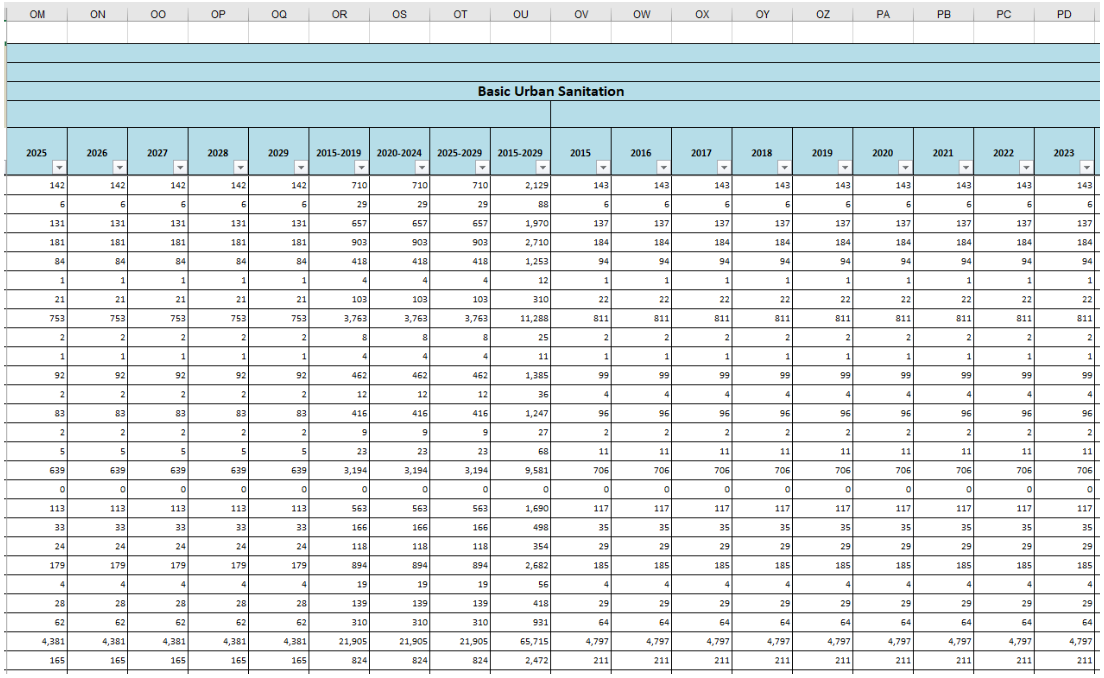
6.2.1 Formula Clustering

When auditing spreadsheets, users need a quick and clear way to identify any discrepancies in their formulae. As shown earlier, many real world spreadsheets exhibit a high degree of tabular locality, which means that many formulae that look alike tend to appear close to each other in large groups. Thus, if we were given a range of formulae which may or may not have errors, we should expect that most of the formulae should follow a similar structure. Any formula that does not follow the same structure as a majority of its neighbors should be flagged as a potential discrepancy.

With this observation in mind, we can help users identify cells that have potential errors by examining what we call the **formula template** of each cell. Given a cell with a formula f, we can derive the formula template by removing all of f's cell references and constants. For example, the formula $SUM(A1:B1)+4$ would have a formula template of $SUM(:)+$.

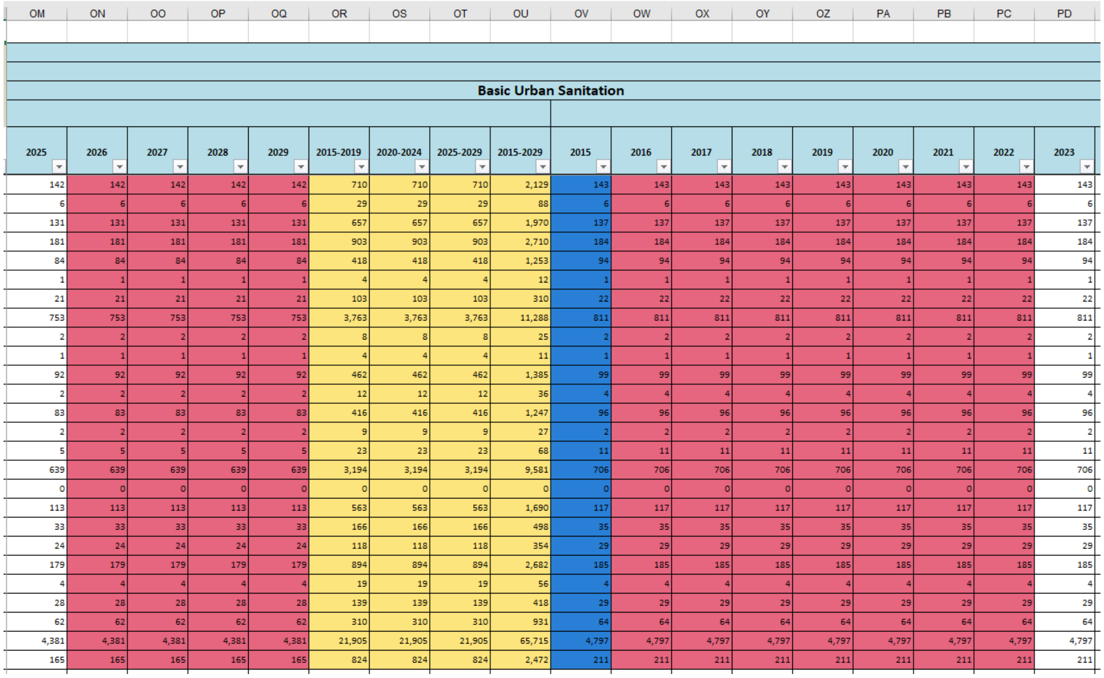
A formula template can be hashed to obtain what we call a cluster ID. Performing this operation on each cell of the spreadsheet creates a clustering, and we can color each cell based on its cluster ID. With this clustering, the user can more easily spot any discrepancies in a large range of cells and more deeply understand how their spreadsheet is arranged.

We chose the lossy mapping scheme above as opposed to something more accurate such as $SUM(-)+4$ or $SUM((0,0),(0,1))+4$ for several key reasons. One of the more important reasons was that, based on our data analysis, the scheme described in the preceding paragraph leads to fewer partitionings of the spreadsheet since it ignores any variations with the constant values of the formula. As a result, the coloring schemes are much easier for the end user to understand and there is a much lower risk of Sherlock devolving into the Excel dependency tracker for larger sheets.



OM	ON	OO	OP	OQ	OR	OS	OT	OU	OV	OW	OX	OY	OZ	PA	PB	PC	PD
								Basic Urban Sanitation									
2025	2026	2027	2028	2029	2015-2019	2020-2024	2025-2029	2015-2029	2015	2016	2017	2018	2019	2020	2021	2022	2023
142	142	142	142	142	710	710	710	2,129	143	143	143	143	143	143	143	143	143
6	6	6	6	6	29	29	29	88	6	6	6	6	6	6	6	6	6
131	131	131	131	131	657	657	657	1,970	137	137	137	137	137	137	137	137	137
181	181	181	181	181	903	903	903	2,710	184	184	184	184	184	184	184	184	184
84	84	84	84	84	418	418	418	1,253	94	94	94	94	94	94	94	94	94
1	1	1	1	1	4	4	4	12	1	1	1	1	1	1	1	1	1
21	21	21	21	21	103	103	103	310	22	22	22	22	22	22	22	22	22
753	753	753	753	753	3,763	3,763	3,763	11,288	811	811	811	811	811	811	811	811	811
2	2	2	2	2	8	8	8	25	2	2	2	2	2	2	2	2	2
1	1	1	1	1	4	4	4	11	1	1	1	1	1	1	1	1	1
92	92	92	92	92	462	462	462	1,385	99	99	99	99	99	99	99	99	99
2	2	2	2	2	12	12	12	36	4	4	4	4	4	4	4	4	4
83	83	83	83	83	416	416	416	1,247	96	96	96	96	96	96	96	96	96
2	2	2	2	2	9	9	9	27	2	2	2	2	2	2	2	2	2
5	5	5	5	5	23	23	23	68	11	11	11	11	11	11	11	11	11
639	639	639	639	639	3,194	3,194	3,194	9,581	706	706	706	706	706	706	706	706	706
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
113	113	113	113	113	563	563	563	1,690	117	117	117	117	117	117	117	117	117
33	33	33	33	33	166	166	166	498	35	35	35	35	35	35	35	35	35
24	24	24	24	24	118	118	118	354	29	29	29	29	29	29	29	29	29
179	179	179	179	179	894	894	894	2,682	185	185	185	185	185	185	185	185	185
4	4	4	4	4	19	19	19	56	4	4	4	4	4	4	4	4	4
28	28	28	28	28	139	139	139	418	29	29	29	29	29	29	29	29	29
62	62	62	62	62	310	310	310	931	64	64	64	64	64	64	64	64	64
4,381	4,381	4,381	4,381	4,381	21,905	21,905	21,905	65,715	4,797	4,797	4,797	4,797	4,797	4,797	4,797	4,797	4,797
165	165	165	165	165	824	824	824	2,472	211	211	211	211	211	211	211	211	211

Figure 6.2: A dense spreadsheet with no formula template highlighting



Basic Urban Sanitation

OM	ON	OO	OP	OQ	OR	OS	OT	OU	OV	OW	OX	OY	OZ	PA	PB	PC	PD
2025	2026	2027	2028	2029	2015-2019	2020-2024	2025-2029	2015-2029	2015	2016	2017	2018	2019	2020	2021	2022	2023
142	142	142	142	142	710	710	710	2,129	143	143	143	143	143	143	143	143	143
6	6	6	6	6	29	29	29	88	6	6	6	6	6	6	6	6	6
131	131	131	131	131	657	657	657	1,970	137	137	137	137	137	137	137	137	137
181	181	181	181	181	903	903	903	2,710	184	184	184	184	184	184	184	184	184
84	84	84	84	84	418	418	418	1,253	94	94	94	94	94	94	94	94	94
1	1	1	1	1	4	4	4	12	1	1	1	1	1	1	1	1	1
21	21	21	21	21	103	103	103	310	22	22	22	22	22	22	22	22	22
753	753	753	753	753	3,763	3,763	3,763	11,288	811	811	811	811	811	811	811	811	811
2	2	2	2	2	8	8	8	25	2	2	2	2	2	2	2	2	2
1	1	1	1	1	4	4	4	11	1	1	1	1	1	1	1	1	1
92	92	92	92	92	462	462	462	1,385	99	99	99	99	99	99	99	99	99
2	2	2	2	2	12	12	12	36	4	4	4	4	4	4	4	4	4
83	83	83	83	83	416	416	416	1,247	96	96	96	96	96	96	96	96	96
2	2	2	2	2	9	9	9	27	2	2	2	2	2	2	2	2	2
5	5	5	5	5	23	23	23	68	11	11	11	11	11	11	11	11	11
639	639	639	639	639	3,194	3,194	3,194	9,581	706	706	706	706	706	706	706	706	706
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
113	113	113	113	113	563	563	563	1,690	117	117	117	117	117	117	117	117	117
33	33	33	33	33	166	166	166	498	35	35	35	35	35	35	35	35	35
24	24	24	24	24	118	118	118	354	29	29	29	29	29	29	29	29	29
179	179	179	179	179	894	894	894	2,682	185	185	185	185	185	185	185	185	185
4	4	4	4	4	19	19	19	56	4	4	4	4	4	4	4	4	4
28	28	28	28	28	139	139	139	418	29	29	29	29	29	29	29	29	29
62	62	62	62	62	310	310	310	931	64	64	64	64	64	64	64	64	64
4,381	4,381	4,381	4,381	4,381	21,905	21,905	21,905	65,715	4,797	4,797	4,797	4,797	4,797	4,797	4,797	4,797	4,797
165	165	165	165	165	824	824	824	2,472	211	211	211	211	211	211	211	211	211

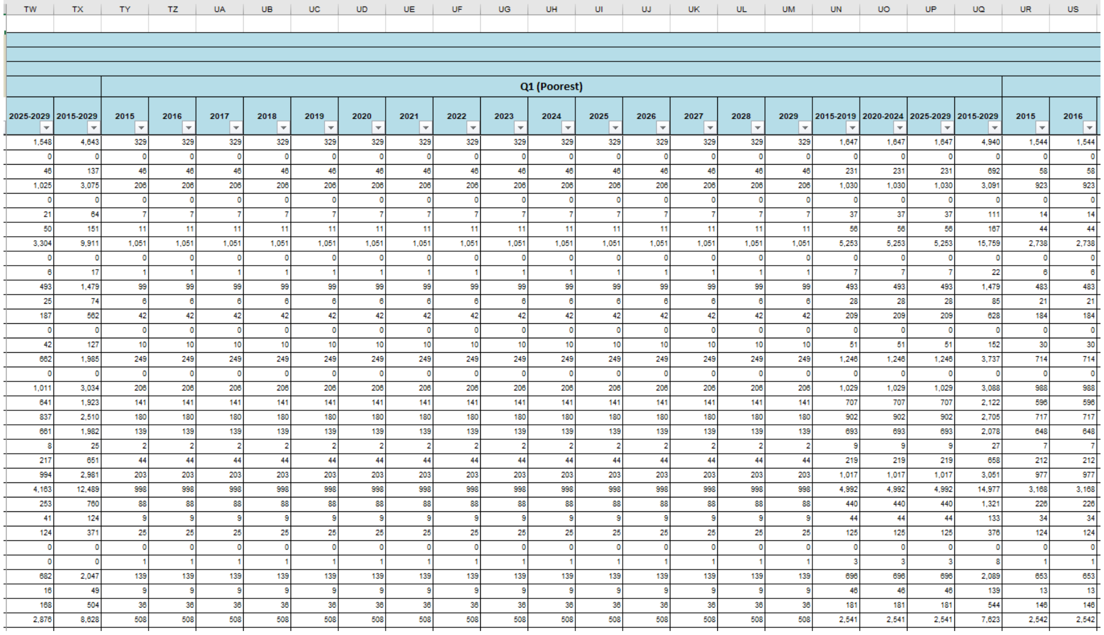
Figure 6.3: An example of formula template clustering

To further illustrate the motivation for formula templates, we include Figures 6.2 and 6.3, which show a sample spreadsheet before and after a formula template clustering respectively. In Figure 6.2, we see that it can be very difficult for the end user to find potential discrepancies in the range, obtain a sense of how formulae are arranged on the sheet, or determine columns of interest. Formula template clustering provides a solution to these problems. In particular, the user can now see that the colors are mostly uniform across each group indicating a low chance of error. Similarly, the user can see that this range of formulae consists of three different clusters, which suggests that there's only three types of fundamental computation being performed in the range. Finally, we note that the column for 2016 is much more narrow than the others, which informs the user that the computation differs from that of other years. Overall, the formula template coloring scheme can provide a much more straightforward summary of the user's selected range.

6.3 Formula Reference Visualization

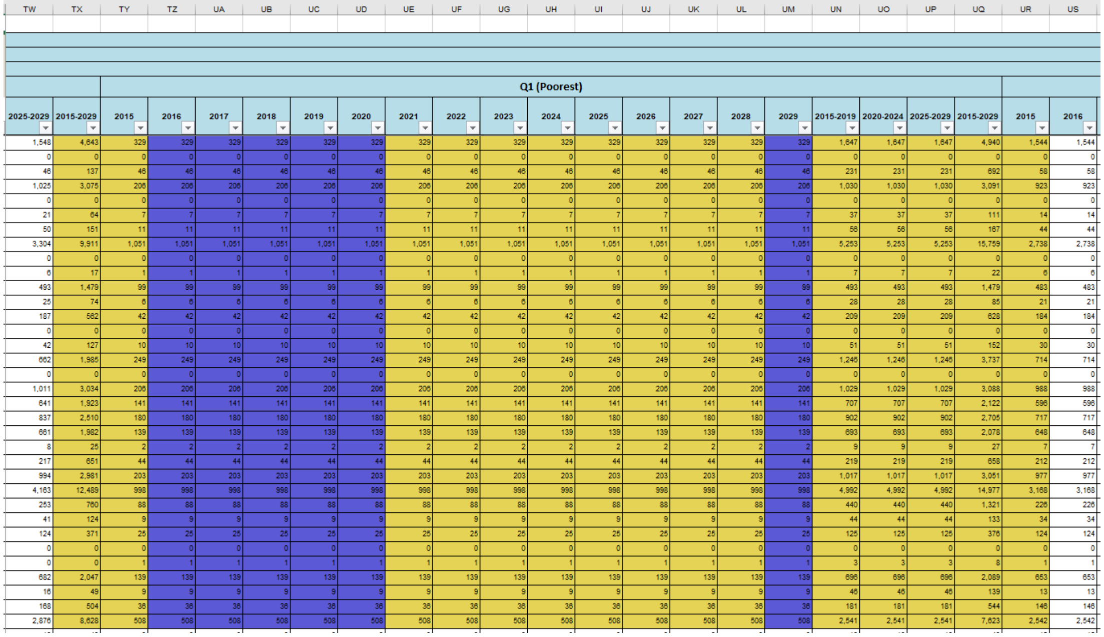
6.3.1 Highlighting TACO Patterns

While formula template clustering can be useful for identifying potential errors in a given range, it can also be useful to see what types of TACO patterns exist on a particular portion of a spreadsheet. Using the same principles outlined previously, Sherlock can identify any RR, FR, RF, FF, and RR-chain patterns on a spreadsheet. It is also capable of analyzing row-wise and column-wise patterns and can analyze patterns for formulae that reference more than one cell range as well.



TW	TX	TY	TZ	UA	UB	UC	UD	UE	UF	UG	UH	UI	UJ	UK	UL	UM	UN	UO	UP	UQ	UR	US
		Q1 (Poorest)																				
2025-2029	2015-2029	2015	2016	2017	2018	2019	2020	2021	2022	2023	2024	2025	2026	2027	2028	2029	2015-2019	2020-2024	2025-2029	2015-2029	2015	2016
1,548	4,643	329	329	329	329	329	329	329	329	329	329	329	329	329	329	329	1,647	1,647	1,647	4,940	1,544	1,544
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
46	137	46	46	46	46	46	46	46	46	46	46	46	46	46	46	46	231	231	231	692	58	58
1,025	3,075	206	206	206	206	206	206	206	206	206	206	206	206	206	206	206	1,030	1,030	1,030	3,091	923	923
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
21	64	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	37	37	37	111	14	14
50	151	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	56	56	56	167	44	44
3,304	9,911	1,051	1,051	1,051	1,051	1,051	1,051	1,051	1,051	1,051	1,051	1,051	1,051	1,051	1,051	1,051	5,253	5,253	5,253	15,759	2,738	2,738
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	17	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	7	7	7	22	6	6
493	1,479	99	99	99	99	99	99	99	99	99	99	99	99	99	99	99	493	493	493	1,479	483	483
25	74	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	28	28	28	85	21	21
187	562	42	42	42	42	42	42	42	42	42	42	42	42	42	42	42	209	209	209	628	184	184
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
42	127	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	51	51	51	152	30	30
662	1,985	249	249	249	249	249	249	249	249	249	249	249	249	249	249	249	1,246	1,246	1,246	3,737	714	714
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1,011	3,034	206	206	206	206	206	206	206	206	206	206	206	206	206	206	206	1,029	1,029	1,029	3,088	988	988
641	1,923	141	141	141	141	141	141	141	141	141	141	141	141	141	141	141	707	707	707	2,122	596	596
837	2,510	180	180	180	180	180	180	180	180	180	180	180	180	180	180	180	902	902	902	2,705	717	717
661	1,982	139	139	139	139	139	139	139	139	139	139	139	139	139	139	139	693	693	693	2,078	648	648
8	25	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	9	9	9	27	7	7
217	651	44	44	44	44	44	44	44	44	44	44	44	44	44	44	44	219	219	219	658	212	212
994	2,981	203	203	203	203	203	203	203	203	203	203	203	203	203	203	203	1,017	1,017	1,017	3,051	977	977
4,163	12,489	998	998	998	998	998	998	998	998	998	998	998	998	998	998	998	4,992	4,992	4,992	14,977	3,168	3,168
253	760	88	88	88	88	88	88	88	88	88	88	88	88	88	88	88	440	440	440	1,321	226	226
41	124	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	44	44	44	133	34	34
124	371	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	125	125	125	376	124	124
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	3	3	3	8	1	1
682	2,047	139	139	139	139	139	139	139	139	139	139	139	139	139	139	139	696	696	696	2,089	653	653
16	49	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	46	46	46	139	13	13
168	504	36	36	36	36	36	36	36	36	36	36	36	36	36	36	36	181	181	181	544	146	146
2,876	8,628	508	508	508	508	508	508	508	508	508	508	508	508	508	508	508	2,541	2,541	2,541	7,623	2,542	2,542

Figure 6.4: A dense spreadsheet with no TACO highlighting

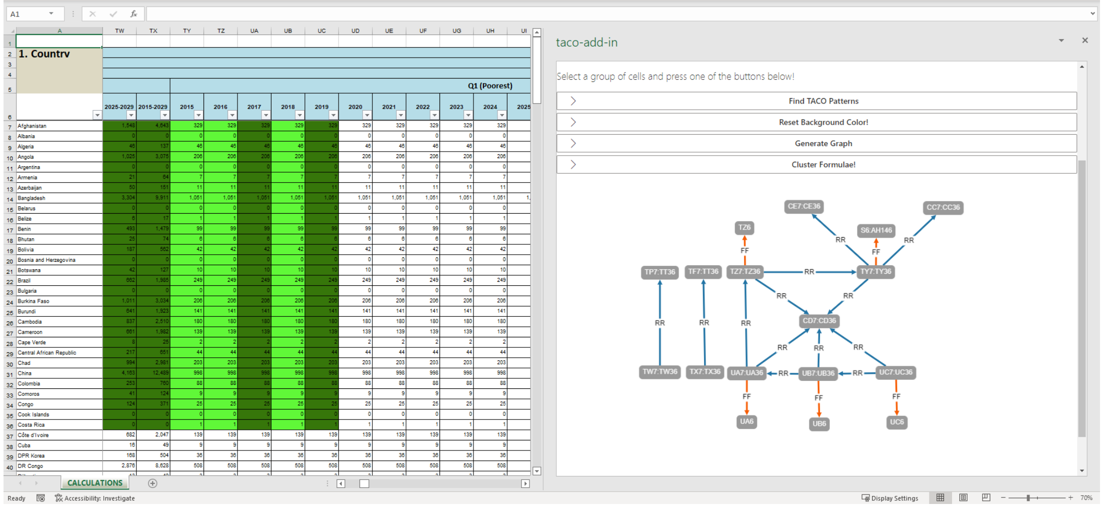


TW	TX	TY	TZ	UA	UB	UC	UD	UE	UF	UG	UH	UI	UJ	UK	UL	UM	UN	UO	UP	UQ	UR	US
		Q1 (Poorest)																				
2025-2029	2015-2029	2015	2016	2017	2018	2019	2020	2021	2022	2023	2024	2025	2026	2027	2028	2029	2015-2019	2020-2024	2025-2029	2015-2029	2015	2016
1,548	4,643	329	329	329	329	329	329	329	329	329	329	329	329	329	329	329	1,647	1,647	1,647	4,940	1,544	1,544
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
46	137	46	46	46	46	46	46	46	46	46	46	46	46	46	46	46	231	231	231	692	58	58
1,025	3,075	206	206	206	206	206	206	206	206	206	206	206	206	206	206	206	1,030	1,030	1,030	3,091	923	923
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
21	64	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	37	37	37	111	14	14
50	151	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	56	56	56	167	44	44
3,304	9,911	1,051	1,051	1,051	1,051	1,051	1,051	1,051	1,051	1,051	1,051	1,051	1,051	1,051	1,051	1,051	5,253	5,253	5,253	15,759	2,738	2,738
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	17	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	7	7	7	22	6	6
493	1,479	99	99	99	99	99	99	99	99	99	99	99	99	99	99	99	493	493	493	1,479	483	483
25	74	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	28	28	28	85	21	21
187	562	42	42	42	42	42	42	42	42	42	42	42	42	42	42	42	209	209	209	628	184	184
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
42	127	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	51	51	51	152	30	30
662	1,985	249	249	249	249	249	249	249	249	249	249	249	249	249	249	249	1,246	1,246	1,246	3,737	714	714
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1,011	3,034	206	206	206	206	206	206	206	206	206	206	206	206	206	206	206	1,029	1,029	1,029	3,088	988	988
641	1,923	141	141	141	141	141	141	141	141	141	141	141	141	141	141	141	707	707	707	2,122	596	596
837	2,510	180	180	180	180	180	180	180	180	180	180	180	180	180	180	180	902	902	902	2,705	717	717
661	1,982	139	139	139	139	139	139	139	139	139	139	139	139	139	139	139	693	693	693	2,078	648	648
8	25	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	9	9	9	27	7	7
217	651	44	44	44	44	44	44	44	44	44	44	44	44	44	44	44	219	219	219	658	212	212
994	2,981	203	203	203	203	203	203	203	203	203	203	203	203	203	203	203	1,017	1,017	1,017	3,051	977	977
4,163	12,489	998	998	998	998	998	998	998	998	998	998	998	998	998	998	998	4,992	4,992	4,992	14,977	3,168	3,168
253	760	88	88	88	88	88	88	88	88	88	88	88	88	88	88	88	440	440	440	1,321	226	226
41	124	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	44	44	44	133	34	34
124	371	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	125	125	125	376	124	124
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	3	3	3	8	1	1
682	2,047	139	139	139	139	139	139	139	139	139	139	139	139	139	139	139	696	696	696	2,089	653	653
16	49	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	46	46	46	139	13	13
168	504	36	36	36	36	36	36	36	36	36	36	36	36	36	36	36	181	181	181	544	146	146
2,876	8,628	508	508	508	508	508	508	508	508	508	508	508	508	508	508	508	2,541	2,541	2,541	7,623	2,542	2,542

Figure 6.5: An example of TACO highlighting

6.3.2 Visualizing TACO Graphs

Finally, Sherlock provides a utility for transforming a range of cells on the spreadsheet into its corresponding TACO graph visualization. In Figure 6.6, we demonstrate the usage of this feature on a random spreadsheet from the Github dataset. Here, we selected an arbitrary range from the spreadsheet and instructed Sherlock to generate the corresponding TACO graph from it. On the left side of the image, we see that Sherlock identified two types of TACO patterns in the selected range. On the right hand side, Sherlock provides a complete breakdown of the types of the identified patterns, the nodes of the compressed formula graph, and the dependencies between each of the nodes. In this case, we can see that a large portion of the selected cells depend on $CD7$ to $CD36$ and each of these dependencies fall under the RR pattern type. With this information, users can better estimate how long an update to a particular cell will take, spot outliers in the graph more easily, and can obtain a more localized view of the computations within a particular area of the spreadsheet.



A1

	A	TW	TX	TY	TZ	UA	UB	UC	UD	UE	UF	UG	UH	UI
1														
2	1. Countrv													
3														
4														
5													Q1 (Poorest)	
6		2025-2029	2015-2029	2015	2016	2017	2018	2019	2020	2021	2022	2023	2024	2025
7	Afghanistan	1,548	4,643	329	329	329	329	329	329	329	329	329	329	
8	Albania	0	0	0	0	0	0	0	0	0	0	0	0	
9	Algeria	46	137	46	46	46	46	46	46	46	46	46	46	
10	Angola	1,025	3,075	206	206	206	206	206	206	206	206	206	206	
11	Argentina	0	0	0	0	0	0	0	0	0	0	0	0	
12	Armenia	21	64	7	7	7	7	7	7	7	7	7	7	
13	Azerbaijan	50	151	11	11	11	11	11	11	11	11	11	11	
14	Bangladesh	3,304	9,911	1,051	1,051	1,051	1,051	1,051	1,051	1,051	1,051	1,051	1,051	1,
15	Belarus	0	0	0	0	0	0	0	0	0	0	0	0	
16	Belize	6	17	1	1	1	1	1	1	1	1	1	1	
17	Benin	493	1,479	99	99	99	99	99	99	99	99	99	99	
18	Bhutan	25	74	6	6	6	6	6	6	6	6	6	6	
19	Bolivia	187	562	42	42	42	42	42	42	42	42	42	42	
20	Bosnia and Herzegovina	0	0	0	0	0	0	0	0	0	0	0	0	
21	Botswana	42	127	10	10	10	10	10	10	10	10	10	10	
22	Brazil	662	1,985	249	249	249	249	249	249	249	249	249	249	
23	Bulgaria	0	0	0	0	0	0	0	0	0	0	0	0	
24	Burkina Faso	1,011	3,034	206	206	206	206	206	206	206	206	206	206	
25	Burundi	641	1,923	141	141	141	141	141	141	141	141	141	141	
26	Cambodia	837	2,510	180	180	180	180	180	180	180	180	180	180	
27	Cameroon	661	1,982	139	139	139	139	139	139	139	139	139	139	
28	Cape Verde	8	25	2	2	2	2	2	2	2	2	2	2	
29	Central African Republic	217	651	44	44	44	44	44	44	44	44	44	44	
30	Chad	994	2,981	203	203	203	203	203	203	203	203	203	203	
31	China	4,163	12,489	998	998	998	998	998	998	998	998	998	998	
32	Colombia	253	760	88	88	88	88	88	88	88	88	88	88	
33	Comoros	41	124	9	9	9	9	9	9	9	9	9	9	
34	Congo	124	371	25	25	25	25	25	25	25	25	25	25	
35	Cook Islands	0	0	0	0	0	0	0	0	0	0	0	0	
36	Costa Rica	0	0	1	1	1	1	1	1	1	1	1	1	
37	Côte d'Ivoire	682	2,047	139	139	139	139	139	139	139	139	139	139	
38	Cuba	16	49	9	9	9	9	9	9	9	9	9	9	
39	DPR Korea	168	504	36	36	36	36	36	36	36	36	36	36	
40	DR Congo	2,876	8,628	508	508	508	508	508	508	508	508	508	508	

taco-add-in

Select a group of cells and press one of the buttons below!

Figure 6.6: Sherlock TACO graph visualization

This type of visualization can not only scale to larger input sizes, but also provides a more holistic view of the user's spreadsheet and its computation graph. The conventional formula graph representation on the other hand would have dozens of redundant edges for the same range shown in the figure, which can make it challenging for the user to understand the dependencies between cells. In the next section, we move onto describing more of the implementation details for these functionalities.

6.4 System Implementation

6.4.1 Backend

Now that we've described the functionalities of Sherlock, we move more into the algorithms and architecture that comprise the add-in starting with the backend architecture. While the Excel API has plenty of rich functionalities for interacting with the user's spreadsheet, it does not provide utilities for parsing a formulae into tokens, which is crucial for identifying patterns in spreadsheets [22]. To resolve this issue, Sherlock was designed using a frontend-backend architecture scheme unlike traditional Excel add-ins. The backend follows a model-view-controller

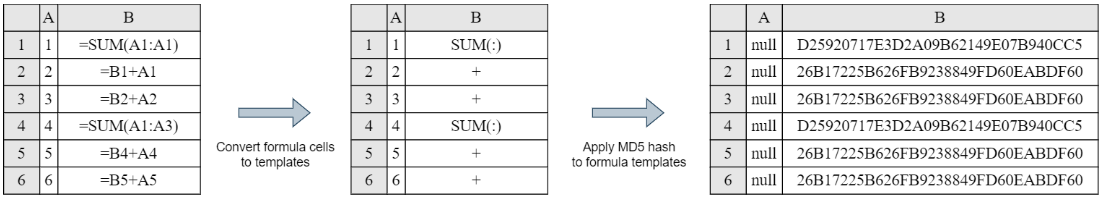
(MVC) architecture and is written in Java so that we can make use of the Apache POI library for parsing formulae [23]. TACO is implemented as a standalone Maven package.

The backend API consists of a couple endpoints for analyzing formulae, which we list below:

1. POST /formulas/cluster
2. POST /taco/patterns

Each endpoint accepts a 2D array of spreadsheet cell contents as input from the frontend and uses a combination of the POI and sheet analyzer packages to help automate the process of scanning through dense formula ranges manually. The response format differs for each of these endpoints and we will briefly discuss them here.

For the formula clustering endpoint, the backend receives the user's highlighted cell contents from the frontend as a 2D array. It then iterates over the 2D array and computes the cluster ID for each element using the formula template mapping process described previously. Once the formula templates have been computed, the backend sends the results of the clustering to the frontend as a 2D array. The returned 2D array has the same shape as the input 2D array, and each element in the returned 2D array now corresponds to a cluster ID rather than the contents of the original spreadsheet. An example of the formula template process is provided in Figure 6.7.



	A	B
1	1	=SUM(A1:A1)
2	2	=B1+A1
3	3	=B2+A2
4	4	=SUM(A1:A3)
5	5	=B4+A4
6	6	=B5+A5

Convert formula cells to templates

	A	B
1	1	SUM(:)
2	2	+
3	3	+
4	4	SUM(:)
5	5	+
6	6	+

Apply MD5 hash to formula templates

	A	B
1	null	D25920717E3D2A09B62149E07B940CC5
2	null	26B17225B626FB9238849FD60EABDF60
3	null	26B17225B626FB9238849FD60EABDF60
4	null	D25920717E3D2A09B62149E07B940CC5
5	null	26B17225B626FB9238849FD60EABDF60
6	null	26B17225B626FB9238849FD60EABDF60

Figure 6.7: An example formula clustering

For the TACO pattern endpoint, the backend utilizes the formula compression algorithm described in chapter 4 to analyze the input 2D array. The response object is a mapping from a cell range to an array of metadata for the range. Each metadata object in the array contains edge data and a cell reference data. In the edge data, sheet analyzer provides info about the starting and ending offsets as well as the pattern type, and in the reference data sheet analyzer returns info regarding the workbook that the range exists on (e.g., book name, sheet name) as well as data regarding the location of the range (e.g., column, last column, row, last row, sheet index). These pieces of information are sent back to the front end and TACO patterns can be highlighted based on the metadata returned. An example of a single metadata object is provided below in Figure 6.8 for clarity.

```json
{
  "taco": {
    "default-sheet-name": {
      "default:A4": [
        {
          "ref": {
            "bookName": "default",
            "sheetName": "default",
            "_type": "AREA",
            "_row": 3,
            "_column": 1,
            "_lastRow": 4,
            "_lastColumn": 1,
            "_sheetIdx": -1
          },
          "edgeMeta": {
            "patternType": "TYPEFOUR",
            "startOffset": {
              "rowOffset": 0,
              "colOffset": 0
            },
            "endOffset": {
              "rowOffset": 0,
              "colOffset": 0
            }
          }
        }
      ],
      ...
    }
  }
}
```

Figure 6.8: Example response for TACO patterns endpoint

6.4.2 Frontend

On the frontend, Sherlock presents a very simple UI consisting of four buttons for the user. The user can select a range of formulae from their spreadsheet and click one of the buttons on the side panel to interact with Sherlock's supported functionalities.

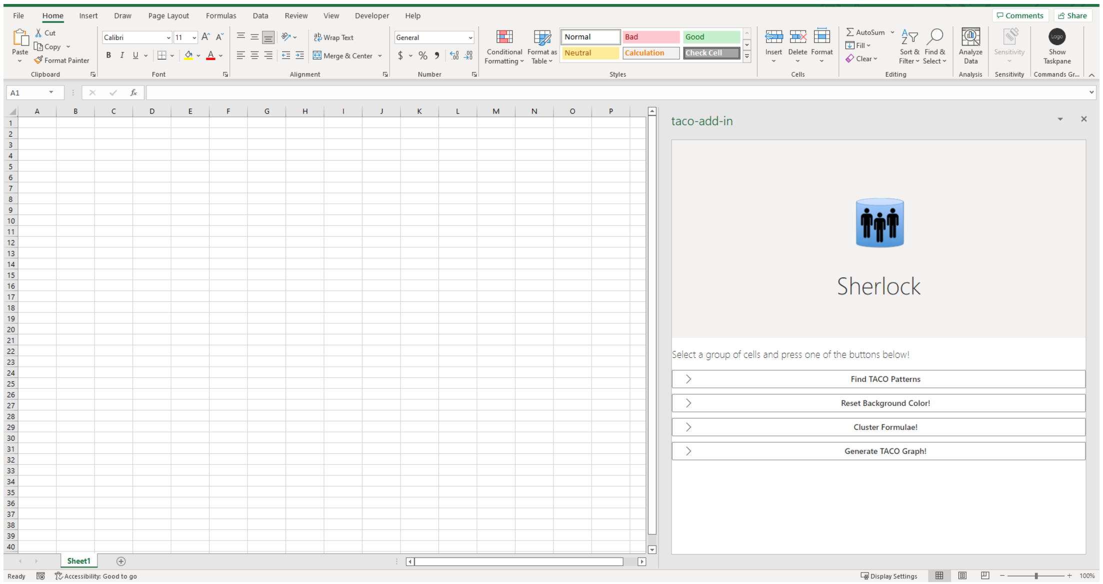

Select a group of cells and press one of the buttons below!

Figure 6.9: Sherlock UI

Once the user highlights a range and selects one of the buttons, the front end will send the contents of the selected range as a 2D array to the front end. The 2D array contains the raw cell contents that were extracted by the Excel API [24]. Once the contents have been sent to the backend, they are processed by the sheet analyzer library, and the front end uses the response object described in the preceding section to color the cells accordingly.

6.5 Conclusion

In this chapter we have introduced Sherlock, an Excel add-in that can help users make sense of their spreadsheets in a much more automated way. Sherlock is capable of performing formula template analysis and formula dependency analysis. For formula template analysis, Sherlock examines a formula's structure and generates a colorful visualization based on how similar each formula looks in comparison to others. Sherlock can also perform formula dependency analysis by examining the TACO patterns of the user's selected range. For this type of analysis, Sherlock can generate two types of visualizations. The first is a coloring of the TACO patterns on the spreadsheet and the second is an interactive graph which users can use to navigate the compressed formula graph.

While Sherlock provides many functionalities for analyzing spreadsheets at scale, the tool still has several limitations and hence many areas of potential improvement. For example, Sherlock is only capable of analyzing either a formula's template or its TACO dependencies but not both. Thus, one area of future work would involve using some algorithm to combine the information from both of these types of analyses to achieve stronger formula pattern recognition. Another limitation of the tool is that it is only capable of examining dependencies within the user's selected range of cells, so if there exists dependencies beyond the user's selected range, these will not be identified. To alleviate this, one could use the Excel API to track dependencies beyond the user's originally selected range and apply the same techniques from above to highlight the patterns in the full dependency chain.

Chapter 7

Conclusion

In this thesis, we presented TACO, a system for efficiently compressing formula graphs and optimizing spreadsheet computations. We also introduced Sherlock, an Excel add-in that allows users to visualize their formula graphs in new ways using the TACO framework. TACO addresses the scalability issue of spreadsheet systems by making use of tabular locality to compress formula graphs and Sherlock allows users to better understand these formula graphs by providing a UI for presenting them in a human-readable way.

Our experiments with TACO demonstrate that by leveraging the formula patterns discussed above, we can achieve much faster dependency identification times and thus much more efficient spreadsheet re-computations. Our Sherlock framework complements TACO by providing users with simple functionalities to explore formula graphs more intuitively and precisely. We believe these frameworks can provide a foundation for scaling spreadsheet systems to the level necessary to manage today's big data needs.

Bibliography

[1] "Anti-freeze for large and complex spreadsheets: Asynchronous formula computation." https://people.eecs.berkeley.edu/ adityagp/papers/dataspread-async.pdf.

[2] "Characterizing scalability issues in spreadsheet software using online forums." https://www.researchgate.net/publication/324669113_Characterizing-_Scalability_Issues_in_Spreadsheet_Software_using_Online_Forums.

[3] "Benchmarking spreadsheet systems." https://people.eecs.berkeley.edu/ adityagp-/papers/spreadsheet_bench.pdf.

[4] "Memory usage in the 32-bit edition of excel 2013 and 2016." https://docs.microsoft.com/en-us/office/troubleshoot/excel/memory-usage-32-bit-edition-of-excel.

[5] "Excel performance: Improving calculation performance." https://docs.microsoft.com/en-us/office/vba/excel/concepts/excel-performance/excel-improving-calculation-performance.

[6] "Calc/implementation/formula cell and cells dependence." https://wiki.openoffice.org/wiki/Calc/Implementation/Formula_cell_and_cells_dependence.

[7] "Zk spreadsheet." https://www.zkoss.org/product/sheet.

[8] "Zk spreadsheet." https://www.zkoss.org/product/sheet.

[9] P. Sestoft, *Spreadsheet Implementation Technology: Basics and Extensions.* The MIT Press, 2014.

[10] "Display the relationships between formulas and cells." https://support.microsoft.com/en-us/office/display-the-relationships-between-formulas-and-cells-a59bef2b-3701-46bf-8ff1-d3518771d507.

[11] "Survey and taxonomy of lossless graph compression and space-efficient graph representations." https://arxiv.org/abs/1806.01799.

[12] "Succinct data structures for assembling large genomes." https://pubmed.ncbi.nlm.nih.gov/21245053/.

[13] "On compressing social networks." https://dl.acm.org/doi/10.1145/1557019.1557049.

[14] "Excelint: Automatically finding spreadsheet formula errors." https://www.microsoft.com/en-us/research/publication/excelint-automatically-finding-spreadsheet-formula-errors/.

[15] "Perquimans: A tool for visualizing patterns of spreadsheet function combinations." https://people.engr.ncsu.edu/ermurph3/papers/vissoft16.pdf.

[16] A. Guttman, "R-trees: A dynamic index structure for spatial searching," 1984.

[17] M. Bendre *et al.*, "Dataspread: Unifying databases and spreadsheets," in *VLDB*, 2015.

[18] B. Klimt and Y. Yang, "Introducing the enron corpus.," in *CEAS*, 2004.

[19] "Range.Dependents." https://docs.microsoft.com/en-us/office/vba/api/excel.range.dependents.

[20] "Errors in operational spreadsheets." https://www.researchgate.net/publication/220068767_E

[21] "Spreadsheet practices and challenges in a large multinational conglomerate." https://jssmith1.github.io/assets/pdf/VLHCC17_Spreadsheets.pdf.

[22] "Excel api." https://docs.microsoft.com/en-us/office/dev/add-ins/reference/overview/excel-add-ins-reference-overview.

[23] "Apache poi." https://poi.apache.org/.

[24] “Excel javascript api.” https://docs.microsoft.com/en-us/office/dev/add-ins/reference/overview/excel-add-ins-reference-overview.