Anti-Freeze for Large and Complex Spreadsheets: Asynchronous Formula Computation

Mangesh Bendre, Tana Wattanawaroon, Kelly Mack
Kevin Chen-Chuan Chang, Aditya Parameswaran
University of Illinois at Urbana-Champaign (UIUC)
(bendre1|wattana2|knmack2|kcchang|adityagp)@illinois.edu

ABSTRACT

Spreadsheet systems used by nearly 1/10th of the world's population allow users to store and analyze data in an intuitive and flexible interface. Yet the scale of data and the complexity of formula computation often leads to spreadsheets hanging and freezing on small changes. We propose a new asynchronous approach to formula computation: instead of freezing the interface until all of the formulae are computed, we return control to the user quickly, while computing the formulae in the background, ensuring interactivity. For formulae that are being computed in the background, we indicate the formula computation progress on the spreadsheet frontend via visual cues. To ensure interactivity without introducing inconsistencies, we develop a flexible and efficient formula computation framework that (a) identifies cell dependencies in bounded time, and (b) schedules computation to maximize the number of cells available to the user over time. We bound the dependency identification time by compressing the formula dependency graph lossily, a problem we demonstrate to be NP-Hard, and propose techniques for compression and maintenance. Finding an optimal computation schedule to maximize cell availability is also NP-Hard, and even merely obtaining a schedule can be expensive—we propose an on-the-fly scheduling technique to address this. Overall, we have incorporated our approach in a scalable spreadsheet system that we are developing, targeted at operating on arbitrarily large datasets on a spreadsheet frontend.

1 INTRODUCTION

Spreadsheets are one of the most popular systems for ad-hoc storage and analysis of data, with a user base of roughly 10% of the world's population [31]. From personal bookkeeping to complex financial reports to scientific data analysis, the ubiquity of spreadsheets as a computing system is unparalleled. Nardi and Miller [24] identify two reasons for their success: an intuitive tabular *presentation*, and in-situ formula *computation*. In particular, formula computation enables end-users with little programming experience to be able to interrogate their data, and compute derived statistics.

However, the sheer volume of data available for analysis in a host of domains exposes the limitations of traditional formula computation. A recent study exploring Microsoft Excel forum posts on Reddit describes several instances of Excel becoming unresponsive while computing formulae [21]. One user posted[1] that complex calculations on Excel can take as long as four hours to finish, during which time the user interface is unresponsive:

> *"... approximately 90% of the time I spend with the spreadsheet is waiting for it to recalculate ..."*

Another user reported using spreadsheets to track their entire life, and periodically cull data to keep the size manageable, but they still have trouble with formula computation:

> *"... the spreadsheet locks up during basic calculations—the entire screen freezes ..."*

The chief culprit for this unresponsiveness is that in traditional spreadsheet systems, every change, be it changing values or formulae, triggers a sequence of computation of dependent formulae. This sequence could take minutes to complete, depending on the size of the data and complexity of the formulae. Since these systems aim to present a "consistent" view after any update, *i.e.*, one with no stale values, they forbid users from interacting with the spreadsheet while the computation is being performed, limiting interactivity. They *only return control to the user after the computation is complete*: the only indication to the user is a bar at the bottom, as in Figure 1(c), with no viewing, scrolling, or edits allowed. Recent studies have shown that even delays of 0.5s can lead to fewer hypotheses explored and insights generated [19], so this *synchronous computation* approach is not desirable.

One workaround that traditional spreadsheet systems provide is a *manual computation* approach, wherein computation of dependent formulae is performed only when triggered manually by users. This method breaks consistency, as stale values are visible to the users, as in Figure 1(b), potentially leading to users drawing incorrect conclusions.

Towards Interactivity and Consistency

We introduce an *asynchronous computation approach that preserves both interactivity and consistency*. After updates, we *return control to the user almost immediately*, "blur out" cells that are not yet up-to-date or consistent, and compute them in the background, incrementally making them available once computed. Users are able to continue working on the rest of the spreadsheet. We show an example in Figure 1(a)

[1] All Reddit quotes are paraphrased to preserve anonymity.

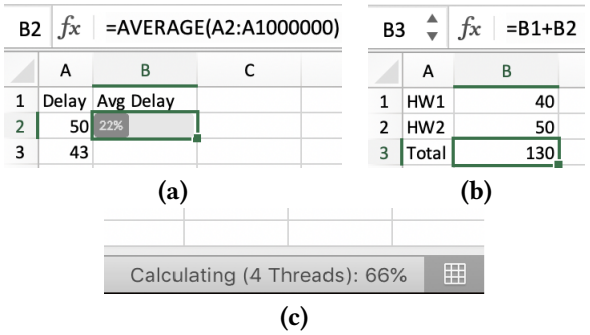


Figure 1: (a) Our proposed asynchronous computation maintains interactivity and consistency by showing computation status instead of a stale value. (b) Manual computation mode in traditional spreadsheets achieve interactivity but violate consistency. (c) Automatic calculation mode in traditional spreadsheets achieve consistency but keep user interface non-responsive for the duration of the computation.

where the formula in B2 summing up one million values is "blurred out", with a progress bar indicating the computation progress, while users can still interact with the rest of the sheet. For example, a user can add a new formula to cell B3, after which both formulae are computed in the background.

We can quantify the benefit of this approach using a new metric we developed, called *unavailability*, *i.e.*, the number of cells that are not available for the user to operate on, at any given time. Synchronous computation has the highest unavailability, since all of the sheet is inaccessible while computation is being performed. In contrast, asynchronous computation allows users to interact with most of the sheet while computation happens in the background, leading to low unavailability, while still respecting consistency.

While the asynchronous computation approach is appealing and natural, and dramatically minimizes the time during which users cannot interact with the spreadsheet, and consequently the unavailability, it requires a fundamental redesign of the formula computation engine, thanks to two primary challenges: *dependencies*, and *scheduling*. Next, we describe these challenges along with our approaches to address them.

Dependencies: Challenges and Approach

Since we need to preserve both interactivity and consistency, once a change is made, we need to quickly identify cells dependent on that change, and therefore must be "blurred out", or made unavailable, as in B2 in Figure 1(a). One simple approach is to traverse the network or graph of formula dependencies to find all dependent cells, and then make them unavailable. However, during this period, the entire spreadsheet is unavailable, so we aim to minimize the time spent in identifying dependent cells. Unfortunately, for computationally heavy spreadsheets, a traditional dependency graph that captures formula dependencies at the cell level [32] can be quite large, so identifying dependencies can be computationally intensive and cannot be done in a bounded time. Ideally, we would like to do this within interactive timescales (less than 500 ms [19]), without sacrificing consistency.

To enable fast lookups of dependencies, we introduce the idea of *compression*. Dependency graphs can tolerate false positives, *i.e.*, identifying a cell as being impacted by an update, even when it is not. However, false negatives are not permitted, since they violate consistency. The goal of compression is to represent the dependencies of each cell by using a bounded number of regions. Using this representation, we can quickly identify the impacted cells after a user updates a cell, ensuring interactivity and consistency.

When compressing our representation of the dependency graph, we trade off the size of the representation and the number of false positives. The size impacts the dependency lookup time, and the false positives impact the formula computation time, and thus both impact the unavailability. We show that graph compression is NP-Hard. Thus, formally, our challenge is to find an optimal way to compress the dependencies such that the unavailability metric is optimized.

We propose techniques and data structures for compressing the dependency graph and its maintenance.

Scheduling: Challenges and Approach

Once we have identified cells that are dependent on the change that was made (with possibly a few false positives), we then need to compute them efficiently, so that we can decrease unavailability as much and as quickly as possible. In the asynchronous computation model, we incrementally return the values of the dependent cells to users as soon as they are computed, as opposed to waiting for all cells to be computed, as is done in a synchronous computation model. When adhering to a schedule, or an order in which the cells are computed, the time that a dependent cell is unavailable essentially comprises of *(i)* time waiting for prior cells in the schedule to complete, and *(ii)* computing of the cell itself. Therefore, choosing the schedule is crucial because it directly impacts the unavailability. For example, if we compute a cell that takes more time to compute early in the schedule, all other cells pay the penalty of being unavailable during this time. A computation schedule must respect dependencies: the computation of a cell must be scheduled only after all the cells that it depends on are computed.

We find that not only is determining an optimal schedule NP-Hard, merely obtaining a schedule can be prohibitively expensive as it requires traversal of the entire dependency graph—this is undesirable and can negate the benefits gained from incrementally returning the computed values within the asynchronous computation model. We propose an on-the-fly scheduling technique that reduces the up-front scheduling time by performing local optimization.

Putting It All Together

We are incorporating our asynchronous computation model in a scalable spreadsheet system that we're building DataSpread [7, 8], with the goal of holistically integrating spreadsheets with databases to address the scalability limitations of traditional spreadsheet systems. DataSpread achieves scalability by utilizing a two-tiered memory model, where data resides in an underlying relational database and is fetched on-demand into main-memory, which is limited in size. This introduces additional challenges that go beyond those found in traditional spreadsheets which are completely main-memory resident. (Note, however, that our techniques for decreasing unavailability apply equally well to traditional spreadsheets as well as our spreadsheet-database hybrid, DataSpread.) We additionally discuss how we support this two-tiered memory model in this paper. For the two-tiered memory model, the computation schedule impacts not only the unavailability metric but also the total computation time significantly.

Contributions. The following list describes our contributions and also serves as the outline of the paper.

1. Asynchronous Computation. In Section 2, we introduce the asynchronous computation model ensuring interactivity and consistency. Additionally, we propose a novel unavailability metric to quantitatively evaluate our model.

2. Fast Dependency Identification. In Section 3, we propose the idea of lossily compressing the dependency graph to identify dependencies in a bounded time. We show that the problem is NP-Hard, and develop techniques for compression and maintenance of this graph.

3. Computation Scheduling. In Section 4, we discuss the importance of finding an efficient schedule for computing formulae. Since, not only is finding the optimal schedule NP-Hard but also obtaining a schedule expensive, we propose an on-the-fly scheduling technique.

4. Incorporation in the Prototype of DataSpread. In Section 5, we describe DataSpread, a scalable spreadsheet system we built that incorporates the ideas discussed in this paper, operating on very large spreadsheets.

5. Experimental Evaluation. Throughout the paper, we provide illustrative experiments to demonstrate individual ideas. In Section 6, we discuss our experimental setup and provide evaluation with real-world spreadsheets.

2 ASYNCHRONOUS COMPUTATION

We propose an asynchronous computation model to address the interactivity issues of traditional spreadsheet systems when operating on complex spreadsheets. We first define key spreadsheet terminology. We then introduce two principles that influence the design of our model, and conclude with new concepts for our proposed model. We also define *unavailability* to formally quantify spreadsheet usability and evaluate the performance of our computation models.

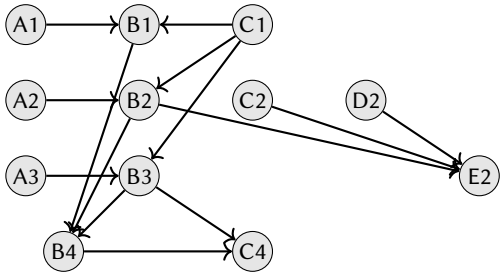


Figure 2: A dependency graph that captures the dependencies of Example 1 at the granularity of cells.

For simplicity, we explain the concepts and techniques in the context of standard spreadsheet tools, which are main-memory-based, where once loaded the cost of data retrieval is negligible compared to the cost of formula evaluation. The techniques, as described for main-memory systems, are beneficial even if used in systems with different memory settings. In Appendix A, we extend our techniques to two-tier memory systems wherein data retrieval cost is significant.

While the techniques discussed in this paper extend to normal usage of spreadsheets where multiple update events happen throughout the timeline, for ease of exposition, we focus on changes resulting from a single update to a cell u.

2.1 Standard Spreadsheet Terminology

We now formally introduce spreadsheet terminology that we utilize throughout the paper.

Spreadsheet Components. A *spreadsheet* consists of a collection of *cells*. A cell is referenced by its *column* and its *row*. Columns are identified using letters A, . . ., Z, AA, . . . in order, while rows are identified using numbers 1, 2, . . . in order. A *range* is a collection of cells that form a contiguous rectangular region, identified by the top-left and bottom-right cells of the region. For instance, A1:C2 is the range containing the six cells A1, A2, B1, B2, C1, C2.

A cell may contain *content* that is either a *value* or a *formula*. A value is a constant belonging to some fixed type. For example, in Figure 1(b), cell A1 (column A, row 1) contains the value HW1. In contrast, a formula is a mathematical expression that contains values and/or cell/range references as arguments to be manipulated by operators or functions. A formula has an *evaluated value*, which is the result of evaluating the expression, with cell references substituted by their values or evaluated values. For the rest of the paper, we shall use the term "value" to refer to either the value or the evaluated value of a cell, depending on what the cell contains. In addition to a value or a formula, a cell could also additionally have formatting associated with it, *e.g.*, width, or font. For the purpose of this paper, we focus only on computation.

Dependencies. In spreadsheets, cell contents may change, and maintaining the correct evaluated values of formulae is necessary for consistency. Consider the following example.

Example 1. *A spreadsheet with the following formulae: (i) B1=A1*C1, (ii) B2=A2*C1, (iii) B3=A3*C1, (iv) B4=SUM(B1:B3), (v) C4=B3+B4, and (vi) E2=SUM(B2:D2).*

The cell B4 has a formula SUM(B1:B3), which indicates that B4's value depends on B1:B3's value. Any time a cell is updated, the spreadsheet system must check to see whether other cells must have their values recalculated. For example, if B2's value is changed, B4's value must be recalculated using the updated value of B2. We formalize the notion of dependencies as follows.

Definition 1 (Direct Dependency). *For two cells u and v, $u \rightarrow v$ is a* direct dependency *if the formula in cell v references cell u or a range containing cell u. Here, u is called a* direct precedent *of v, and v is called a* direct dependent *of u.*

Definition 2 (Dependency). *For two cells u and v, $u \Rightarrow v$ is a* dependency *if there is a sequence $w_0, w_1, \ldots, w_n$ of cells where $w_0 = u$, $w_n = v$, and for all $i \in [n]$, $w_{i-1} \rightarrow w_i$ is a direct dependency. Here, u is called a* precedent *of v, and v is called a* dependent *of u. We denote the set of dependents of a cell u as Δ_u.*

One can construct a conventional *dependency graph* of direct dependencies [32]. Figure 2 depicts the graph for the formulae in Example 1 at the granularity of cells. Here, each vertex corresponds to a single cell, *e.g.*, A1. The edges in the graph indicate direct dependencies. For example, the directed edge from A1 to B1 indicates a direct dependency due to formula A1*C1 in cell B1. The dependencies of a cell u are therefore the vertices that are reachable from u in the dependency graph. For example, cell B1 has B4 and C4 as dependents. As this dependency graph captures dependencies at the granularity of cells, this graph grows quickly when the ranges mentioned in the formulae are large [32]. For example, a formula SUM(A1:A1000) in cell F2 will require 1,001 vertices and 1,000 edges to capture the dependencies.

2.2 Design Principles

We introduce consistency and interactivity as two fundamental principles that any system should maintain during formula computation. Spreadsheets should be *consistent*, *i.e.*, they should not display stale values. For example, if a cell B2 contains the formula SUM(A1:A225500) and the user updates the value in cell A1, the user should not see the stale value in B2 until the corresponding formula is recomputed. Along with consistency, spreadsheet systems must ensure *interactivity*, meaning they should react to user events, such as cell updates, rapidly, and provide users with results as soon as possible—this is crucial for the usability of any interactive exploration systems [19]. Thus, we introduce the following two design principles by which our solution must abide.

Principle 1 (Consistency). *Never display an outdated or incorrect value on the user interface.*

Principle 2 (Interactivity). *Return control to users within a bounded time after any cell update user event.*

In relation to these two principles, we describe the computation model adopted by traditional spreadsheet systems, and then discuss our proposed model.

Synchronous Computation Model. Traditional spreadsheet systems adopt a *synchronous computation model*, where, upon updating u, the entire spreadsheet becomes unavailable during the evaluation of cells that are dependent on u. The spreadsheet system waits for all of the computation to complete before providing updated values and returning control back to the user—thereby adhering to the consistency principle. However, the waiting time can be substantial for computationally intensive spreadsheets. According to our recent Reddit study [21], waiting for formula computation is one of the primary sources of poor interactivity. In other words, when the number of cells dependent on u is large, this model sacrifices interactivity, with often minutes to hours of unresponsiveness.

Asynchronous Computation Model. To adhere to interactivity in addition to consistency, we propose an asynchronous computation model. Here, upon updating u, the cells dependent on u are computed asynchronously in the background without blocking the user interface.

One naïve asynchronous approach could be to merely modify the synchronous model to return control to the user immediately after an update, even before all the dependent cells are computed. However, similar to the *manual computation* option found in traditional spreadsheet systems, this approach violates consistency. Consider Figure 1(b), where even after updating the value of cell B1 from 80 to 40, the cell B3 is not updated to the correct value of 90 unless a full computation of the spreadsheet is triggered manually—violating consistency, cell B3 shows 130, a stale value.

To satisfy the consistency principle within the asynchronous computation model, we instead provide users with the cells that the system can ensure to have correct values in a short time, while notifying users of cells that have stale values—see Figure 1(a), where upon updating A1 the computation of cell B2 is performed in the background and the computation progress is depicted by a progress bar. Our solution is to add a "dependency identification" step before computation of any dependent formulae. The goal of this step is to efficiently identify the cells that do not depend on an updated cell, so that they can be quickly marked clean and "control" of them can be returned to the user.

2.3 New Concepts

We now introduce new concepts that help us describe and quantify the benefits of the asynchronous computation model.

Partial Results. Within the asynchronous computation model, we introduce the notion of *partial results*: providing users with the cells that the system can ensure to have correct (or consistent) values and notifying users of cells that have stale values. Thus, within these *partial results*, each cell on the spreadsheet is determined by the computation model to be in the "clean" or the "dirty" state, defined as follows.

Definition 3 (Clean Cell). *We consider a cell u to be* clean *if and only if (i) all of u's precedents are clean and (ii) u's evaluated value is determined to be up-to-date.*

Definition 4 (Dirty Cell). *We consider a cell u to be* dirty *if and only if (i) at least one of u's precedents are dirty or (ii) the u's evaluated value is determined to be not up-to-date.*

Adhering to the consistency principle, *(i)* for clean cells, the evaluated value is displayed on the user interface (like in existing spreadsheet systems), and *(ii)* for dirty cells, the cell displays a progress bar depicting the status of its computation, thus preventing users from acting on stale values. Note that a dirty cell is one that is *determined* by the computation model to be dirty, and therefore requires recomputation. As we will see later, a dirty cell may be a false positive, but we will treat both false positives and true positives equivalently since they will both be recomputed—and are therefore both dirty from the perspective of the computation model.

Finally, we introduce one last term to describe the state of a cell: the *unavailable* state. A cell is unavailable if it cannot be used by the user for various reasons, defined as follows.

Definition 5 (Unavailable Cell). *We consider a cell c to be* unavailable *if and only if a user cannot act on c either because (i) c is determined to be dirty or (ii) the system has not yet determined if c is in the clean or dirty state or (iii) the user interface is unresponsive.*

Utilizing the idea of partial results within the asynchronous computation model, we propose to provide users with the cells that are being computed as soon as they are ready (moving them from the dirty to the clean state), without waiting for all of the cells to be computed. This idea of incrementally computing and marking cells as clean allows the number of unavailable cells to gradually decrease over time.

Unavailable and Dirty Time. Quantifying the time a cell is unavailable to the user to act upon is an important factor for understanding the usability of the spreadsheet. Similarly, the dirty time is the time a cell spends in the dirty state. We formalize the notion of unavailable and dirty time as below.

Definition 6 (Unavailable Time). *The unavailable time of a cell c, denoted as* unavailable(c)*, is the amount of time that c remains in the unavailable state after an update.*

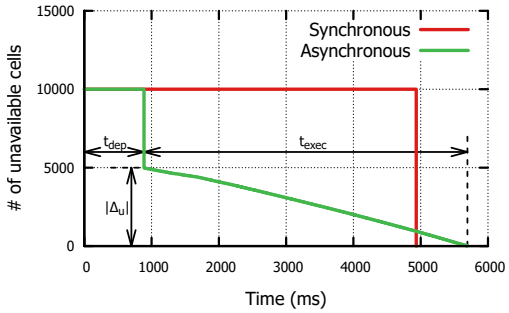


Figure 3: Unavailability comparing synchronous and asynchronous models. For the asynchronous model, t_{dep} denotes dependency identification time, Δ_u is the set of cells that are determined to be dependent on u and therefore need computation, and t_{exec} denotes computation time for these cells.

Definition 7 (Dirty Time). *The dirty time of a cell c, denoted as* dirty(c)*, is the amount of time that c remains in the dirty state after an update.*

Unavailability. To quantitatively evaluate different computation models, we introduce the metric of *unavailability*, which we define as the area under the curve that, for a computation model, plots the number of unavailable cells in a spreadsheet with respect to time.

Definition 8 (Unavailability). *The unavailability U_M for a computation model M is given by $U_M = \int_0^t D(t)\,dt = \sum_{c \in S}$ unavailable(c), where $D(t)$ denotes the number of unavailable cells at time t and S is the set of all spreadsheet cells.*

Simply put, unavailability measures the effectiveness of a computation model by quantifying the number of cells that a user cannot act upon over time. Therefore, a computation model with lower unavailability is more *usable* than a model with a higher value. For the synchronous computation model, for the entire time the user interface is unresponsive, all of the cells within the spreadsheet are unavailable. On the other hand, by incrementally returning results in the asynchronous computation model, for a cell c, unavailable(c) = t_{dep} + dirty(c), where t_{dep} is the time taken by the system to determine if c is in the clean or dirty state.

Illustrative Experiment 1: Asynchronous vs. Synchronous Computation. The goal of this experiment is to quantitatively compare the asynchronous and synchronous computation models using unavailability. We describe the experimental setup later in Section 6. Here, we adopt a conventional dependency identification mechanism as described in Section 2.1 and a naïve schedule for computing cells—we will build on this and develop better variants later. We use a synthetic spreadsheet with a total of 10,000 cells out of which 5,000 cells are formulae dependent on a cell u. We update the value of u and plot the number of unavailable cells on

the y-axis with respect to time on the x-axis for both computation models—see Figure 3. The synchronous computation model (in red) performs poorly under unavailability, since it keeps the interface unresponsive for the entire duration of computation of all of the dependent cells. The asynchronous computation model (in green) performs better in terms of unavailability, since it allows users to interact with most of the spreadsheet cells while performing calculations asynchronously in the background, with the cells incrementally returned to the user interface as they are complete.

We now describe how the computations proceed with respect to time for both models—refer to Figure 3. Upon updating u (at $time = 0$), the asynchronous model identifies dependents of u, as is marked by t_{dep} on the graph. For both models, all 10,000 cells in the sheet are unavailable for the first 890 ms, as the sheet is unresponsive. After this point, the asynchronous model has determined which cells are clean and which cells are dirty, and it returns the clean cells to the user. Thus, the number of unavailable cells drops down to 5,000 from 10,000 after 890 ms. However, under the synchronous model, control has not been returned to the user, and thus all cells are still unavailable. Under the asynchronous model, at the 5,700 ms mark, all of the cells have been computed and marked clean—this is slightly after the 4,900 ms mark, which is when the synchronous model returns control of all of the cells to the user. This time difference is due to the fact that the asynchronous model takes some time to identify dependent cells in a separate step from computing them, while the synchronous model does not have to have this separate step. Note that the area under the green curve is greater than that under the red curve, and therefore the asynchronous model performs better under unavailability.

Takeaway: The asynchronous computation model improves usability of spreadsheets, without forgoing correctness, by (i) quickly returning control to the user and (ii) incrementally making cells available.

Thus, while this experiment shows that the asynchronous computation model already has a lower unavailability than the synchronous one, it can be reduced even further; in the remainder of this paper, we discuss approaches for doing so.

3 FAST DEPENDENCY IDENTIFICATION

In this section, we propose our first technique for decreasing unavailability: *identifying dependencies in a bounded time*. Upon updating u, our strategy is to quickly identify u's dependencies, ideally, within a bounded time—this enables us to promptly identify the cells that do not depend on u as clean and return their control to the user. Our strategy, the *dependency table*, aims to reduce t_{dep} in Figure 3, which is the time during which the user interface is non-responsive for the asynchronous computation model. Reducing t_{dep} is particularly crucial when the update affects a small number of cells relative to the size of the spreadsheet. We propose compression to accelerate dependency identification by grouping a large number of dependent cells into a smaller number of regions. We then discuss construction and maintenance of the compressed dependency table.

3.1 Motivation and Problem Statement

After a user updates a cell u in a spreadsheet, to minimize the number of unavailable cells over time, we need to quickly identify the cells that depend on u. Until we can determine that a cell c is independent of u or not, we cannot designate c as clean and return its control to the user. For example, within the asynchronous computation model in Figure 3, we return the control to the user in 890 ms, which corresponds to the time it takes dependency identification to finish.

A naïve approach to identify the cells that depend on u is to individually check whether each cell is reachable from u in the dependency graph. However, this strategy is time consuming for large and complex spreadsheets, since all cells will remain in the unavailable state for a long period of time.

Our goal is to efficiently identify the cells that do not depend on an updated cell, so that they can be quickly marked clean and their control can be returned to the user. Thus, we formalize our problem as follows:

Problem 1 (Dependency Identification). *Design a data structure that, upon updating u, quickly (preferably in bounded time) identifies u's dependencies. Additionally, modifications to the data structure, i.e., inserts and deletes, should be quick (again, preferably in bounded time).*

Our proposed method of capturing dependencies is to maintain a *dependency graph*. Rather than the conventional method of recording dependencies between individual cells (Figure 2), we capture dependencies between regions—this substantially reduces the size of the dependency graph. Figure 4 shows the dependency graph for Example 1. Our dependency graph has the following four components. *(i)* A *cell vertex* corresponding to each cell, in gray, *e.g.*, A1, B1. *(ii)* A *range vertex* corresponding to each range that appears in at least one formula, in red, *e.g.*, B1:B3. *(iii)* A *formula edge* from u to v if u is an operand in the formula of cell v, *e.g.*, the edge from A1 to B1. *(iv)* An *inherent edge* from u to v if cell u is contained in range v, *e.g.*, the edge from B1 to B1:B3.

In the dependency graph, the cells that depend on a cell u are those represented by vertices reachable from the vertex representing u. For example, the dependencies of the cell C1 are B1, B2, B3, B4, C4, and E2.

We can persist the formula edges in the dependency graph as adjacency lists. Thus, the number of dependent regions within formulae is a good proxy for storage cost. For example, we can represent the formula A1*C1 within cell B1 using two directed edges: *(i)* from A1 to B1 and *(ii)* from C1 to

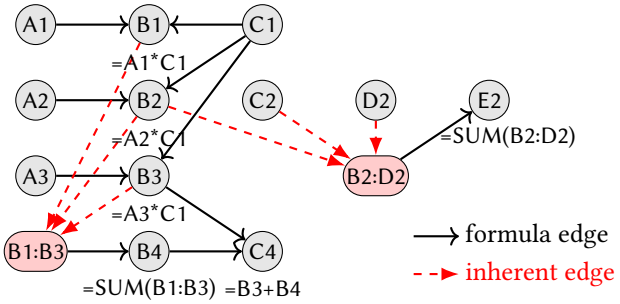


Figure 4: Dependency graph capturing dependencies between regions thus reducing the graph size.

B1. Rather than storing the inherent edges explicitly, which can be expensive, we can infer these edges from the cell and ranges they represent. To enable efficient lookups for inherent edges, we can use a spatial index, such as R-tree [15]. To find outgoing edges from a cell c, we can issue a query to the R-tree to find all ranges containing c. For example, to infer the outgoing edges from B2, we can search for all the nodes that overlap with B2—for B2 we have B1:B3 and B2:D2.

Challenges With Dependency Traversal. The lookup of dependencies by traversing a full dependency graph takes time proportional to the number of dependencies, which is inefficient when the number of dependencies is large. Consider the scenario depicted in Figure 5—looking up dependencies of A1 takes $\Omega(n)$ time, where n is the number of dependencies. For example, the t_{dep} of 890 ms in Figure 3 will increase linearly with the number of dependencies. Therefore, to perform the dependency identification in a bounded time, we cannot traverse the dependency graph on-the-fly.

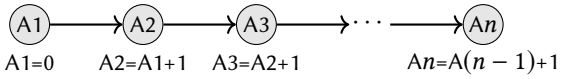


Figure 5: Long Dependency Chain

3.2 Compressed Dependency Table (CDT)

To overcome the aforementioned challenge, we propose an alternate manner to capture dependencies. In addition to the dependency graph, we maintain a "cache" of dependents for each cell, in a *dependency table*—see Figure 7(a). The dependency table stores key-value pairs of cells and their dependents, and thus allows us to query a cell u and quickly identify all of the cells that depend on u. We can construct the dependency table from scratch by traversing the dependency graph multiple times, starting from every vertex.

As discussed, the number of dependencies of a cell is $\Theta(n)$ in the worst case, where n is the number of cells on the spreadsheet, and thus even recording each dependency at a cell level could take too long and be expensive to store. Therefore, we propose *compression* to reduce both the dependency identification time and the dependency output size.

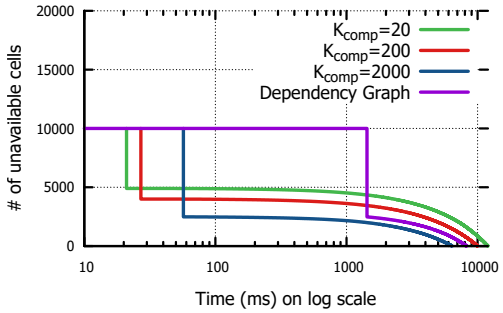


Figure 6: Comparing unavailability for by using dependency graph vs dependency table with varying K_{comp}.

cell	dependents
A1	B1, B4, C4
A2	B2, B4, C4, E2
A3	B3, B4, C4
B3	B4, C4
C1	B1, B2, B3, B4, C4, E2
⋮	⋮

(a)

cell	dependents
A1	B1, B4:C4
A2	B2:C4, E2
A3	B3, B4:C4
B3	B4, C4
C1	B1:C4, E2
⋮	⋮

(b)

Figure 7: Compressing dependency table to bound the number of dependents: (a) original before compression (b) after compression with $K_{\text{comp}} = 2$.

Recall that to ensure consistency, we must recalculate all the dependent cells on a cell update. If the dependency table includes a "false positive", *i.e.*, a cell c_{FP} that is not an actual dependency of u, the system will trigger a recalculation of c_{FP}, whose value will remain the same. In other words, the dependency table is *false positive tolerant*—the presence of a false positive does not affect correctness, but can cause unnecessary calculations. On the other hand, a "false negative", a cell c_{FN} that is an actual dependency of u but is missing from the table, is unacceptable, because a update to u would not trigger a recalculation of c_{FN}, leading to a possibly incorrect value for c_{FN}.

A *compressed dependency table*, or CDT for short, is a variation of a dependency table that enables identifying dependencies in $O(1)$ time—see Figure 7(b). As ranges naturally represent a group of cells, we express the dependents in a compressed dependency table as ranges. For example, dependents of C1 can be expressed as B1:B3, B4:C4, E2 with no false positives, or as B1:C4, E2 with three false positives (C1, C2, C3). For a set of cells C to be expressed as a set of regions R, we require that the regions in R can collectively "cover" the set C. We formalize the notion of a cover as follows.

Definition 9 (Cover). *For a set C of cells, a set $R = \{R_1, \ldots, R_m\}$ of ranges is a* cover *of C if $C \subseteq R^{\cup}$, where $R^{\cup}$ denotes the set of cells that are in at least one of the ranges $R_1, \ldots, R_m$. The size of the cover R, denoted by* size(R), *is $|R|$. The cost of the cover R, denoted by* cost(R), *is $\left|R^{\cup}\right|$.*

To ensure that dependents of a cell u can be retrieved and reported in constant time, we limit the size of the cover to a constant K_{comp}. In Figure 7(b), the K_{comp} is 2. Varying the value of K_{comp} can significantly impact the unavailability, due to the following: there is a trade-off between the time it takes to perform dependency identification (t_{dep} in Figure 3) and the number of cells that remain when dependency identification is complete (Δ_u in Figure 3). This trade-off is because the less time we spend identifying dependencies, the smaller the K_{comp} and the more false positives we introduce into the dependency table. This increase in false positives causes the cost of the cover, and therefore the total number of dirty cells at the time immediately following dependency identification, to increase. Ultimately, we need a value of K_{comp} that minimizes unavailability.

Illustrative Experiment 2: Impact of K_{comp}. In this experiment we quantitatively demonstrate the benefit of using a dependency table instead of a traditional dependency graph using unavailability. Additionally, we also demonstrate the impact of varying K_{comp} for the dependency table. We consider a synthetic spreadsheet having 10,000 cells out of which 5,000 cells contain formulae. Out of the 5,000 formulae cells, 50% of the cells are dependent on a cell u, which we intersperse with cells that are independent of u. For this synthetic spreadsheet, we update the value of u and plot the number of unavailable cells on the y-axis with respect to time on the x-axis for the asynchronous computation model—see Figure 6. Due to a large number of dependencies, dependency identification using the dependency graph (in purple) takes a significant time of 1.4 seconds. The three remaining curves show the benefit of using a dependency table—here, we vary K_{comp} and observe its impact on the time for identifying dependencies. At one extreme, we have the blue curve where K_{comp} is 2,000—the dependency identification takes around 60 ms. On the other hand, the green curve, when K_{comp} is 20, remains in the dependency identification step for very little time (20 ms). However, to compress all of the dependents of a cell into 20 regions, the number of false positives grow to 2,400 cells. Therefore, even though the green curve returns control to the user in a few milliseconds, it takes more time to clean all the dirty cells. In this example, $K_{\text{comp}} = 2{,}000$ (in blue) performs the best under the unavailability metric, as its curve encloses the least area.

Takeaway: Dependency table with lossy compression of dependencies bounds the time for which user interface is unresponsive.

3.3 Construction of the CDT

When constructing the compressed dependency table, our goal is to group dependents of each cell into K_{comp} groups while allowing for the fewest false positives and no false negatives. We formalize the problem as follows:

Problem 2 (Dependents Compression). *Given a set C of cells and a size parameter k, find the cover of C whose size does not exceed k with the smallest cost.*

Grouping the dependents of a cell u into K_{comp} regions amounts to solving Problem 2 with a set Δ_u of cells and a size parameter K_{comp}, where Δ_u is the set of cells dependent on u. For a cover R, the number of false positives is $|R^{\cup}| - |\Delta_u|$. Thus, minimizing the number of false positives is equivalent to minimizing the cost of the cover. It turns out that the aforementioned problem is NP-Hard—see Theorem 1. The proof of the theorem is given in Appendix B.

Theorem 1. *The decision version of Dependents Compression is* NP-Hard.

Greedy Heuristic. Since efficiently finding the best compression is hard, we propose a greedy algorithm for graph compression: while the number of ranges representing dependents of a cell exceeds K_{comp}, two of those ranges are selected and replaced by the smallest range enclosing them; repeat until the number of ranges reduce to K_{comp}. We can use various heuristics for selecting the two ranges to combine. One such simple heuristic is to select two ranges such that replacing them with their enclosing range introduces the fewest false positives, which, as we will see, does well in practice. Note that due to the incremental nature of our compression algorithm, we can use it for the maintenance of the dependency table when we add a new dependency, as we will see next. The pseudocode for the greedy compression algorithm is given as Algorithm 1 in Appendix B.3.

3.4 Maintenance of the CDT

We now discuss how to update the compressed dependency table when formulae are changed. Adhering to the interactivity principle, our goal is to return control to the user quickly after an update. Therefore, the time taken to modify the dependency table must be small. Finding all dependents of a cell by traversing the dependency graph again, for example, is infeasible. We now introduce techniques for inserting into and deleting elements from the dependency table.

Deleting dependencies. Deleting a dependent from the dependency table can potentially introduce false negatives. To illustrate this, consider the example provided in Figures 4 and 7. Here, C4 is a dependent of B3. Suppose the formula in C4 is changed to =B4+3, and thus the direct dependency B3 $\rightarrow$ C4 is deleted. However, we cannot remove C4 from the dependent list of B3, because C4 remains a dependent of B3, albeit no longer a direct one. In other words, the dependency between C4 and B3 is due to more than one formula. Another issue is deleting a single dependent cell from one represented by a range, which is difficult to do efficiently without leading to a highly fragmented, inefficient R-tree.

A simple way to circumvent deletion issues in the compressed dependency table is to make no changes to the table upon direct dependency deletion. If d is a dependent that is supposed to be deleted but is instead ignored and kept, then d becomes a false positive, which, as previously discussed, does not affect correctness but adds to computation time. Over time, however, false positives resulting from deletion accumulate. We combat this issue by periodically reconstructing the compressed dependency table from scratch, particularly during spreadsheet idle time. Such a method is also beneficial because the dependents of a cell can change drastically over the lifetime of a spreadsheet, and an entirely new grouping of cells into ranges may lead to a significant decrease in the number of false positives.

Adding dependencies. Adding a direct dependency can be quite time-consuming in the worst case. Consider the example in Figure 8, where the formula of B1 is changed from =0 to =A3+1, and thus a new direct dependency A3 ⟶ B1 is added. Because of this change, A3 and its precedents must have their entries changed in the dependency table by adding B1, B2, B3 as their dependents, which is quite time consuming.

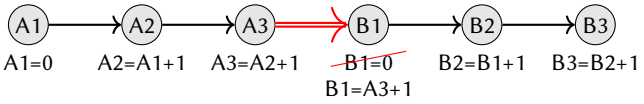


Figure 8: Adding a new direct dependency in a dependency chain.

To get around the aforementioned issue, we introduce *lazy dependency propagation*. The idea is to only add the direct dependency (A3 ⟶ B1) to the dependency table. Such direct dependencies have a *must-expand* status (indicated as a single bit), indicating that the dependency is recently added and not fully processed. Also, the dependency table is put into a special *unstable* state (another bit), indicating that at least one dependency has the *must-expand* status, because we can no longer perform the dependency lookup in the dependency table in the same manner. We propagate the must-expand dependencies in the background, say, during idle time. More precisely, for a must-expand dependency $u \longrightarrow v$, dependents of v are added as dependents of u and all its precedents (in the example above, adding B1, B2, and B3 as dependents to cells A1, A2 and A3). The dependency table leaves the unstable state once we are done propagating all must-expand dependencies.

To identify dependents of u in an unstable dependency table, one must look up dependents recursively, similar to traversing a dependency graph. However, a lookup requires no further recursive steps if none of its dependents have a must-expand dependent.

For example, instead of updating all entries as in Figure 9(a), B1 is added as a must-expand dependent of A3, as in Figure 9(b). At this unstable state, to identify dependents of A1, it is insufficient to just report A2, A3 as dependents, even if neither of the cells *are* must-expand dependents. Since A3 *has* a must-expand dependent B1, the recursive lookup continues, to include B2 and B3. Since neither of the dependents of B1 (which are B2 and B3) has a must-expand dependent, recursion can stop there. Eventually, the must-expand dependent is resolved by a background thread and the dependency table becomes similar to that shown in Figure 9(a).

cell	dependents
A1	A2, A3, B1, B2, B3
A2	A3, B1, B2, B3
A3	B1, B2, B3
B1	B2, B3
B2	B3
B3	

(a)

cell	dependents
A1	A2, A3
A2	A3
A3*	B1*
B1	B2, B3
B2	B3
B3	

(b)

Figure 9: Adding dependencies to dependency table: (a) naïve method (b) lazy dependency propagation (must-expand dependencies are marked by asterisks)

The downside of this approach is that dependency identification does not have a constant time guarantee until all must-expand dependencies are propagated and the table leaves the unstable state. However, this approach quickly returns control to the user and allows users to perform other spreadsheet operations while we update the dependency table, potentially at the expense of speed of subsequent operations, if they come in rapid succession.

Note that adding dependents to a cell can push the number of dependents beyond the K_{comp} limit. To ensure constant lookup time when the dependency table leaves the unstable state, we reduce the number of ranges representing the dependents down to K_{comp} using the method of repeated merging of ranges described in Section 3.3.

4 COMPUTATION SCHEDULING

In this section, we propose our second technique for decreasing unavailability: *computation scheduling*. After updating a cell u, we need to find an efficient schedule for the computation of the cells that depend on u to reduce the amount of time they spend being unavailable. We explain the significance of scheduling, discuss how obtaining a complete scheduling up front can be prohibitively expensive, and provide a solution, *on-the-fly scheduling*. We discuss the extension, weighted computation scheduling, that prioritizes computation based on what users are currently interacting.

Recall that, for asynchronous computation, we incrementally provide users with cell values as soon as they are computed, without waiting for the formula engine to compute the remaining dirty cells. We motivate scheduling by experimentally demonstrating its impact on unavailability.

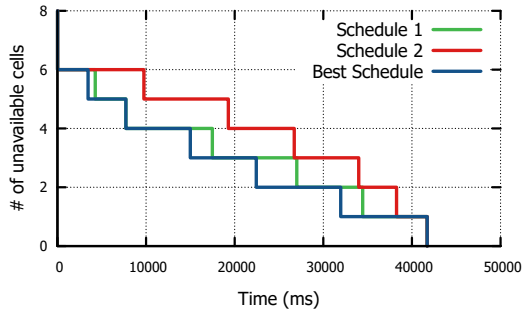


Figure 10: Unavailability varying the computation schedule (dependency identification time of 20 ms is not pictured on the graph)

Illustrative Experiment 3: Computation Schedule. The goal of this experiment is to demonstrate the impact of scheduling. Here, we consider a synthetic spreadsheet with six formula cells. The formulae perform summation using the SUM function of varying sized ranges to simulate varying complexities. In this case, as the complexity of a formula in increases, the time to compute it increases as well. These formula cells are independent of each other but dependent on a cell u. For this sample spreadsheet, we update the value of u and plot the number of formula cells that are unavailable on the y-axis with respect to time on the x-axis—see Figure 10. Even though the total time required to complete cleaning all the cells is the same across all possible schedules (around 40,000 ms), the time spent by each cell in the dirty state varies, which impacts unavailability. Schedule 1 and 2 adopt a random schedule, and thus differ in terms of unavailability. The best schedule computes the cells in the increasing order of complexity, thereby minimizing unavailability.

Takeaway: Computation scheduling is important within the asynchronous computation model and impacts the number of cells that are available to users over time.

4.1 Motivation and Problem Statement

The computation scheduling problem naturally arises from the idea of partial results (Section 2.3): if we are displaying the computed cell values to the user as we finish computing them, in what order should we compute cells? For our computation scheduling problem, we define cost(c) to quantify the time taken for computing a cell c. For now we assume a simple independent computation model where we ignore the impact of caching cells ; we will discuss its impact later and relax this assumption. Their formal definitions are as follows.

Definition 10 (Cost). *The cost of a cell c, denoted by* cost(c)*, is the amount of time needed to compute the evaluated value of c, assuming the values of its precedents are already computed.*

Assumption 1 (Independent Computation Model). *We assume that the cost of computing a cell c, i.e.,* cost(c)*, is independent of the computation schedule. In other words, c takes the same time to evaluate, regardless of when we compute c.*

Note that for a synchronous computation model, computation scheduling is unimportant. The total evaluation time for all cells dependent on u is $\sum_{c\in\Delta_u}$ cost(c), where Δ_u is the set of cells dependent on u. Therefore, in the synchronous model, since all cells in the sheet remain unavailable until all of the computations are completed, the unavailable time for every cell in the spreadsheet is equal to $t_{\text{dep}} + \sum_{c\in\Delta_u}$ cost(c), where t_{dep} is the dependency identification time, and thus unavailability is $U_{\text{sync}} = |S| \cdot \left(t_{\text{dep}} + \sum_{c\in\Delta_u} \text{cost}(c)\right)$, where S is the set of cells in the spreadsheet, regardless of the order in which the cells in Δ_u are computed.

On the other hand, when we incrementally return cells in the asynchronous model, dirty(c) is not the same across all cells because a cell becomes clean as soon as its value is evaluated. Therefore, choosing the order in which cells are computed is crucial because it affects unavailability. For example, one simple intuition is to avoid calculating cells with a high cost early in the schedule, since all other cells must incur this cost in their unavailable time. We will now formally define the computation scheduling problem.

Computation Scheduling Problem. Upon updating u, our goal is to decide the order of evaluation of dependents of u, *i.e.*, Δ_u, such that the order minimizes unavailability. The primary constraint for scheduling the computation of a cell c is that the cells that are precedents of c, if they are dirty, need to be become clean before c itself can be evaluated. Otherwise, the computation would rely on outdated values resulting in incorrect results. Note that because cyclic dependencies are forbidden in spreadsheet systems, there is always at least one order that follows the dependency constraint of the problem: *a topological order*. Formally, we define the *dependency constraint* as follows.

Definition 11 (Dependency Constraint). *A computation order $c_1, \ldots, c_n$ of cells is valid only if the following holds: if $i < j$, then c_i is not a dependent of c_j.*

Recall that the dirty time of a cell c is the amount of time until its value is computed, which includes the time waiting for the earlier elements in the scheduled order to be computed as well as the cost of computing c itself, as follows.

Definition 12 (Dirty Time with respect to a Schedule). *In a computation order $c_1, \ldots, c_n$, the dirty time for the cell c_i is* dirty(c_i) = $\sum_{j=1}^{i}$ cost(c_j) = dirty(c_{i-1}) + cost(c_i).

We formalize our scheduling problem as follows, which is shown as NP-Hard by Lawler [17].

Problem 3 (Computation Scheduling). *Given a set of dirty cells (Δ) along with the dependencies among them, determine a computation order $c_1, \ldots, c_n$ of all the cells in Δ that minimizes unavailability, i.e., $\sum_{c_i \in \Delta} \text{dirty}(c_i)$, under the dependency constraint.*

4.2 On-the-fly Scheduling

In addition to the fact that Computation Scheduling is NP-Hard, upon updating u, merely obtaining a schedule can be prohibitively expensive. The dirty time defined in the previous subsection (Definition 12) only takes into account computation time, but not the time to perform the scheduling itself. If there are n dirty cells in Δ_u, then the amount of time to obtain any complete schedule satisfying the dependency constraints is $\Omega(n)$, as each of the n cells must be examined at least once to determine dependency and the cost of computation. If the scheduling algorithm takes time t_s, then performing scheduling up front increases the dirty time of each cell in Δ_u by t_s, and no progress towards their computation is made during that time. Such an effect is undesirable and potentially negates any gains from incrementally computing and showing results to the users.

To overcome this issue, upon updating u, we do not determine the complete order of all dependents of u up front—instead, we utilize the heuristic of performing scheduling "on-the-fly" by prioritizing a small sample of cells at a time based on their costs. A cell's exact computation cost can be difficult to determine exactly; the number of precedents provides a good approximation. We can easily determine the number of precedents by looking at a cell's formula.

Upon updating u, we perform on-the-fly scheduling as follows. We draw k cells from Δ_u and put them in the pool P. In each step, we choose m cells from P, where $m \ll k$, whose costs are the smallest among those in the pool. The system schedules computation for the chosen m cells. Then, we replenish P by drawing cells from Δ_u that still requires computation until P has k cells again (or until no cells remain). We repeat the steps until all cells in Δ_u are computed.

To properly schedule the chosen m cells for computation obeying the dependency constraint, precedents of each of the m cells must be computed before the cell itself can be computed. Thus, the precedents of the m cells must also be scheduled for computation, in topological order.

The on-the-fly scheduling heuristic attempts to postpone computing high cost cells for as long as possible, because computing low cost cells first allows for more results to be quickly shown to the user. In fact, without dependency requirements, scheduling computation in increasing order of cost yields the optimal schedule [13]. Our heuristic is based on the same principle, but adapted to obey the dependency constraint and to make decisions without looking at the entire workload. The pseudocode for the on-the-fly scheduling procedure is given as Algorithm 2 in Appendix B.3.

4.3 Weighted Computation Scheduling

Due to limited screen real estate, users often do not see all the cells of a spreadsheet at the same time. Typically, spreadsheet systems allow users to interact with spreadsheets through a *viewport*, which we define as a rectangular range of cells that a user can interact with, *i.e.*, read values or update cell content. The user can change the viewport either by scrolling or jumping to the desired part of the spreadsheet.

Since users can only view the cells that are within the viewport, it is desirable to prioritize the computation of cells that the user is currently viewing—for this purpose we introduce a weighted variation of unavailability. Here, each cell c is given a *weight*, denoted as $\text{weight}(c)$. The more important a cell is, the higher its weight. For example, we can prioritize computation of cells in the viewport by assigning a high weight, $w \gg 1$, to cells within the viewport and a low weight, 1, to other cells. It may also be desirable to assign a medium weight to cells just outside the viewport, as scrolling to these cells is likely. The following formalization of a *weighted unavailability* modifies our previously defined unavailability (see Definition 8) such that if a high-weight cell is left dirty, and therefore unavailable, for an extended period, the metric's value is much higher.

Definition 13 (Weighted Unavailability). *The weighted unavailability W_M for a computation model M over a spreadsheet S is $W_M = \sum_{c \in S} (\text{weight}(c) \cdot \text{unavailable}(c))$, where $\text{weight}(c)$ is the weight of c, $\text{unavailable}(c)$ is the unavailable time for c, and S is the set of all cells within the spreadsheet.*

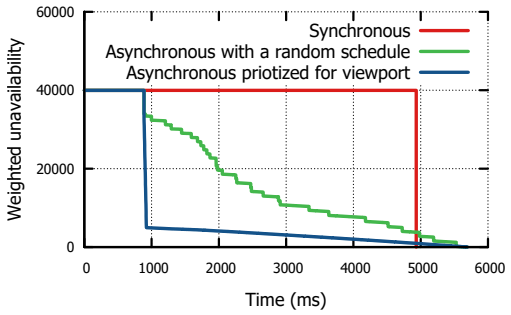


Figure 11: Weighted unavailability comparing synchronous computation models and asynchronous computation model with and without viewport prioritization.

Using the weighted unavailability, we now formalize a weighted variation of our computation scheduling problem that aims at minimizing weighted unavailability while adhering to dependency constraint.

Problem 4 (Weighted Computation Scheduling). *Given a set of dirty cells (Δ) along with their weights and the dependencies among them, determine an order $c_1, \ldots, c_n$ of all the cells in Δ that minimizes weighted unavailability, i.e., $\sum_{c_i \in \Delta} (\text{weight}(c_i) \cdot \text{dirty}(c_i))$, where $\text{dirty}(c_i) = \sum_{j=1}^{i} \text{cost}(c_j) = \text{dirty}(c_{i-1}) + \text{cost}(c_i)$ and $\text{weight}(c)$ is the weight of c, under the Dependency constraint.*

Weighted Computation Scheduling is trivially NP-Hard, since it is a generalization of Computation Scheduling discussed in Section 4.1, which is NP-Hard.

Illustrative Experiment 4: Weighted Scheduling. This experiment demonstrates a weighted variation of Experiment 1, with Figure 11 showing a weighted variation of Figure 3. Here, we assign a weight of 1,000 for 30 formula cells within the user's viewport and 1 for the remaining. We plot time on the x-axis and weighted unavailability (the product of the number of unavailable cells and their weights) on the y-axis. Past the 890 ms mark, the red curve, which represents the synchronous model, maintains the same level of weighted unavailability until all of the cells have been computed and marked clean at around 5,000 ms. For the asynchronous model (in blue) that prioritizes cells in the viewport when scheduling, the weighted unavailability drops off very quickly between 890 ms and 1,000 ms, and then slowly decreases to 0 afterwards. This sharp decline represents the time when the system is computing the highly-weighted cells within the viewport. The remaining, lower-weighted cells outside the viewport are computed afterwards. On the other hand, the asynchronous computation model that uses random scheduling, slowly decreases over time, as high-weighted cells are left in the dirty state due to randomized scheduling. As can be clearly seen in Figure 11, the model which prioritizes cells in the viewport when scheduling performs the best under weighed unavailability.

Takeaway: Weighted computation scheduling enables prioritization of important cells such as those visible on the user interface.

On-the-fly Weighted Scheduling. For weighted computation scheduling, we adapt the on-the-fly scheduling algorithm discussed in Section 4.2 by updating the cost calculation to additionally consider the weight of the cell. Intuitively, we would like to prioritize cells that have a higher weight but a lower cost. Thus, in Algorithm 2, we sort the cells by $cost(c)/weight(c)$, where $cost(c)$ is the computation cost for c and $weight(c)$ is the weight that we assign to c. Additionally, we dynamically update the cell weights when the user changes their viewport by scrolling. Further more we can also modify Algorithm 2 to first pick up the cells that are within the viewport.

5 DATASPREAD SYSTEM

In this section, we introduce DataSpread, the spreadsheet system we have implemented, that utilizes the techniques discussed throughout the paper. We also describe the system's architecture, explaining how components work together to implement asynchronous computation.

Until this point, we focus on the setting of main-memory systems throughout the discussion and the illustrative experiments, for faithfulness with existing main-memory-based systems. However, we develop DataSpread to not only take the available main memory into account, but also utilize larger (disk) storage to handle spreadsheets at scale. In Appendix A, we discuss how the problem changes when we consider such a memory model as used by DataSpread, a *two-tiered memory model*, and how the techniques can be adapted to account for different cost considerations.

5.1 System Description

We have implemented a fully functional prototype for the spreadsheet system DataSpread. The system is designed to be a *one-stop tool that unifies the capabilities of spreadsheets and databases to provide an intuitive direct-manipulation* [29] *interface for managing big data.* Thus, we have designed our tool to be scalable—the current prototype supports interactive browsing of *billion cell spreadsheets* on moderately powered desktop hardware. The support for datasets of this size is only possible by going beyond the limitations of main-memory. Our system leverages an RDBMS's ability to handle data at scale, essentially using it as a paging solution for our spreadsheet system.

Its back-end utilizes a PostgreSQL database for its ability to handle data at scale. DataSpread is a web-based application that uses the React framework [3] for the front-end, and ZK Spreadsheet [5] for the formula engine. The prototype, along with its source code, documentation, and user guide, can be found at http://dataspread.github.io.

DataSpread adopts a two-tiered memory model (Appendix A) where it persists the spreadsheet data in the back-end database and fetches it *on-demand* when triggered by a user action (like scrolling) or a system action (like calculating a formula). In addition to utilizing the techniques described in Sections 2, 3, and 4, we support the following features to ensure interactivity and scalability. *(i)* We reduce the front-end to back-end communication latency by using web-sockets. *(ii)* We implement caching at multiple layers using an LRU based evacuation scheme. *(iii)* Our back-end is context-aware, meaning it keeps track of what the user is looking at and what data is cached in the web-browser. *(iv)* All the communication between back-end threads is event-driven. *(v)* We adopt a push-based architecture, where upon scroll event or change in data, updated cell values are pushed by the back-end to the front-end via web sockets and vice-versa. *(vi)* We

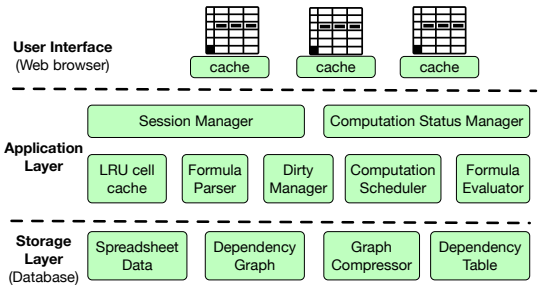


Figure 12: DataSpread **Architecture**

adopt the idea of a virtual DOM (Document Object Model), where the web browser only renders the visible cells.

5.2 Architecture

Figure 12 illustrates DataSpread's architecture, which, at a high level, can be divided into three main layers, *(i)* user interface, *(ii)* application layer, and *(iii)* storage layer. The *user interface* consists of a custom developed *spreadsheet component* based on the React framework, which runs in the web browser and presents a spreadsheet to the user and handles interactions on it. The *application layer* consists of components responsible for main operations of the spreadsheet system, including formula evaluation. These components are developed in Java and reside on an application server. The *storage layer* consists of a relational database and is responsible for persisting spreadsheet data and metadata, including dependency information.

The front-end back-end communication designed using the Spring [4] framework uses *(i)* RESTful APIs for non-latency critical communication, *e.g.*, getting a list of spreadsheets, and *(ii)* web-sockets for latency critical communication, *e.g.*, updating cells on the user interface after an event.

Partial Result Presentation Components. As discussed in Section 2, the ability to present partial results is the main advantage of asynchronous computation. To determine which values are available to the user, the *dirty manager* is responsible for maintaining a collection of regions that are dirty and thus need computation. It utilizes the dirty cell data structures discussed in the Appendix.

The *session manager* keeps track of the user's current viewport and the collection of cells that are cached in the browser—thus upon a scroll event on the user interface, the application layer can determine whether the browser already has the required cells or if new cells need to be pushed. Its viewport information is also useful for viewport prioritization, as discussed in Section 4.3.

It communicates with the *dirty manager* to determine which cells are shown to the user, and make proper changes to the front-end when cells change their availability. It also communicates with the *computation status manager*, which periodically checks the progress of the computations and informs the front-end about the progress, and the front-end updates the progress bars to reflect the progress.

Formula Evaluation Components. The *dependency graph* maintains dependencies between cells and regions. The compressed *dependency table*, maintained by the *dependency table compressor*, allows the system to support fast dependency identification, as discussed in Section 3.

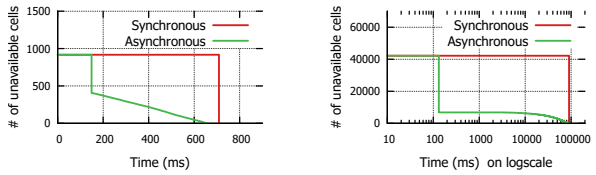
Formula evaluation is triggered by updates to cells on the user interface. Upon a cell update, fast dependency identification mark dependents of the updated cell as dirty in the *dirty manager*. In addition, if the update involves adding, removing, or modifying a formula, the *formula parser* interprets the formula and identifies what cells are required for computation—this information is sent to the *dependency graph* and the *dependency table* to make appropriate updates.

The *computation scheduler* coordinates the formula computation. It retrieves dirty cells from the *dirty manager* and schedules their computation as discussed in Section 4. The actual formula evaluation is done using the *formula engine*, which fetches the cells required for computing the formula from the *LRU cell cache* in a read-through manner, *i.e.*, the cache fetches the cells that are not present on demand from the storage layer. The formula engine then computes the result of the formula. Finally, it persists the calculated result by passing it back to the *LRU cell cache* in a write-through manner, *i.e.*, the cache pushes its updates to the storage.

6 ADDITIONAL EXPERIMENTS

Throughout the paper, we have provided illustrative experiments along with takeaways to demonstrate the individual aspects of our proposed asynchronous computation model. In this section, we describe our setup and additionally provide an evaluation on real-world spreadsheets. The experiments described in this paper aim to demonstrate: *(i)* quantification of the benefit of the asynchronous computation model, *(ii)* the necessity and the benefit of compressing the dependencies, *(iii)* the necessity of finding an efficient computation schedule, *(iv)* how the weighted variation of unavailability prioritizes the computations of cells in a user's viewport, and *(v)* the applicability of our ideas on real-world spreadsheets.

Environment. We have implemented the asynchronous model along with graph compression and computation scheduling within our scalable spreadsheet system, DataSpread, which uses PostgreSQL 10.5 as a backend data store. We run all of our experiments on a workstation running Windows 10 on an AMD Phenom II X6 2.7 GHz CPU with 16 GB RAM. While we have a functional prototype, to eliminate the impact of communication between front-end and back-end, we design our test scripts as single threaded independent applications that directly utilize DataSpread's back-end.



(a) **(b)**

Figure 13: Unavailability comparing synchronous and asynchronous computation models for two real world spreadsheets of different sizes. (a) small (b) large

Dataset. We evaluate our algorithms on a variety of synthetic spreadsheets and some real-world spreadsheets that we collected by a survey from spreadsheet users. We conducted the survey across multiple colleges in a university asking users to send in their largest, most complex spreadsheets. We received tens of spreadsheets, which we then examined one at a time to find a few representative ones with complex and computationally intensive formulae.

Illustrative Experiment 5: Real-world Spreadsheets. In this experiment, we use two real-world spreadsheets to compare the two computation models. For both spreadsheets, we find a cell u that has the highest number of dependents. The first spreadsheet has complex financial calculations, with a total of 917 cells, out of which 406 formula cells are dependent on u. The second spreadsheet is targeted towards inventory management and had a total of 42,181 cells, out of which 6,803 formula cells are dependent on u. We use a naïve synchronous computational model as described in Experiment 1. For the asynchronous model, we use a compressed dependency table with $K_{\text{comp}} = 5000$ (Section 3.2) and schedule computations on-the-fly (Section 4.2). We update u and plot the number of unavailable cells on the y-axis with respect to time on the x-axis—see Figure 13. Observations are similar to Experiment 1. In terms of unavailability, for the asynchronous model, we see an improvement of 2x and 12x over the synchronous model for the first and second spreadsheet respectively, thus confirming the applicability of our ideas to real-world spreadsheets.

Takeaway: Our proposed asynchronous computation model maintains interactivity and consistency thus improving usability of spreadsheet systems for large and complex spreadsheets.

7 RELATED WORK

The asynchronous formula computation model presented in this paper is an alternative to the synchronous model adopted by traditional spreadsheet systems. Problems similar to graph compression and scheduling are studied in various contexts with different goals and constraints. There are also related papers that handle spreadsheets at scale by utilizing database systems for content storage. We now discuss in more detail each of these categories of related work.

Computation Models. Asynchrony has been used in operations with delayed actors, such as in crowdsourcing [22, 26] and web search [11] but never for spreadsheets—their concerns and objectives are very different. The synchronous model of traditional spreadsheets utilize the idea of dependency graph [23, 28, 32] to avoid unnecessary computations, but it does not avoid the performance degradation due to large and complex dependencies [33]. Our proposed asynchronous computation model along with compressed dependency table alleviate such issues as discussed in Section 3.

Graph Compression. Alternate representations of graph-structured data have been introduced for numerous applications, including for web and social graphs. While some papers focus on a high-level understanding of the network via clustering [12], those that obtain a concise representation of graphs to improve query performance are related to our work. Graph compression methods, surveyed by Liu et al. [18], have different focuses, such as compactness with bounded errors [25], pattern matching queries [20], and dynamic graphs [30]. Our setting is different from these works because of *(i)* our goal of quickly obtaining a representation of dependents of a cell, *(ii)* the one-sided (false positive only) tolerance, and *(iii)* the spatial nature of cell ranges.

Scheduling. Scheduling under precedence constraints is a thoroughly studied problem, especially in operations research, with various settings and metrics, including ones similar to the unavailability metric [17]. Similar scheduling problems arise in this paper, and some hardness results are drawn from previous work. However, as discussed in Section 4, in the prior work, schedules are built up front, whereas obtaining a complete schedule up front is prohibitive in our setting. For this reason, we introduce on-the-fly scheduling.

Handling Scale. Previous work on supporting spreadsheets at scale has been done by 1010data [1], ABC [27], Airtable [2], DataSpread [9, 10], and Oracle [34, 35]. While these systems address scale, interactivity for formula computation is not their focus. Our work is distinguished from them by its ability to provide partial results with consistency guarantees.

8 CONCLUSIONS

Our proposed asynchronous computation model improves the interactivity of spreadsheets without violating the consistency while working with large datasets. To support asynchrony without violating consistency, we introduced the idea of partial results, which blurs out the formulae that are being computed in the background. We ensured interactivity by proposing a compressed dependency table to identify dependent cells after a cell update in a bounded amount of time. For usability, we developed an on-the-fly scheduling technique to minimize the number of cells that are pending computation. We have implemented the aforementioned

ideas in DataSpread and demonstrated improved interactivity compared to traditional spreadsheet systems. Thus, our new computation model's improved interactivity allows the use of spreadsheet systems in data analysis situations where it was once inconceivable.

REFERENCES

[1] [n. d.]. 1010 Data. https://www.1010data.com/.

[2] [n. d.]. Airtable. https://www.airtable.com/.

[3] [n. d.]. React:A JavaScript library for building user interfaces. https://reactjs.org.

[4] [n. d.]. Spring Framework. https://spring.io/.

[5] [n. d.]. ZK Spreadsheet. https://www.zkoss.org/product/zkspreadsheet.

[6] 2017. Memory usage in the 32-bit edition of Excel 2013 and 2016. https://support.microsoft.com/en-us/help/3066990/memory-usage-in-the-32-bit-edition-of-excel-2013-and-2016.

[7] Mangesh Bendre et al. 2015. DataSpread: Unifying Databases and Spreadsheets. *VLDB* 8, 12, 2000–2003. https://doi.org/10.14778/2824032.2824121

[8] Mangesh Bendre et al. 2018. Towards a Holistic Integration of Spreadsheets with Databases: A Scalable Storage Engine for Presentational Data Management. In *ICDE*.

[9] Mangesh Bendre, Bofan Sun, Ding Zhang, Xinyan Zhou, Kevin Chen-Chuan Chang, and Aditya Parameswaran. 2015. DataSpread: Unifying Databases and Spreadsheets. *Proc. VLDB Endow.* 8, 12 (Aug. 2015), 2000–2003. https://doi.org/10.14778/2824032.2824121

[10] Mangesh Bendre, Vipul Venkataraman, Xinyan Zhou, Kevin Chen-Chuan Chang, and Aditya G. Parameswaran. 2017. Towards a Holistic Integration of Spreadsheets with Databases: A Scalable Storage Engine for Presentational Data Management. *CoRR* abs/1708.06712 (2017). http://arxiv.org/abs/1708.06712

[11] Sergey Brin and Lawrence Page. 1998. The anatomy of a large-scale hypertextual web search engine. *Computer networks and ISDN systems* 30, 1-7 (1998), 107–117.

[12] R Cilibrasi and PMB Vitanyi. 2005. Clustering by compression. *IEEE Transactions on Information Theory* 51, 4 (2005), 1523–1545.

[13] Richard Walter Conway, William L Maxwell, and Louis W Miller. 2003. *Theory of scheduling*. Courier Corporation.

[14] J.C. Culberson and R.A. Reckhow. 1994. Covering Polygons Is Hard. *Journal of Algorithms* 17, 1 (1994), 2 – 44. https://doi.org/10.1006/jagm.1994.1025

[15] Antonin Guttman. 1984. R-trees: A Dynamic Index Structure for Spatial Searching. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data (SIGMOD '84)*. ACM, New York, NY, USA, 47–57. https://doi.org/10.1145/602259.602266

[16] Alexander V. Kononov, Bertrand M.T. Lin, and Kuei-Tang Fang. 2015. Single-machine scheduling with supporting tasks. *Discrete Optimization* 17 (2015), 69 – 79. https://doi.org/10.1016/j.disopt.2015.05.001

[17] Eugene L Lawler. 1978. Sequencing jobs to minimize total weighted completion time subject to precedence constraints. In *Annals of Discrete Mathematics*. Vol. 2. Elsevier, 75–90.

[18] Yike Liu, Tara Safavi, Abhilash Dighe, and Danai Koutra. 2018. Graph Summarization Methods and Applications: A Survey. *ACM Computing Surveys (CSUR)* 51, 3 (2018), 62.

[19] Zhicheng Liu and Jeffrey Heer. 2014. The Effects of Interactive Latency on Exploratory Visual Analysis. *IEEE Trans. Visualization & Comp. Graphics (Proc. InfoVis)* (2014). http://idl.cs.washington.edu/papers/latency

[20] Antonio Maccioni and Daniel J Abadi. 2016. Scalable pattern matching over compressed graphs via dedensification. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 1755–1764.

[21] Kelly Mack et al. 2018. Characterizing Scalability Issues in Spreadsheet Software using Online Forums. In *SIGCHI*.

[22] Adam Marcus, Eugene Wu, David R Karger, Samuel Madden, and Robert C Miller. 2011. Crowdsourced databases: Query processing with people. Cidr.

[23] Decision Models. 2014. ExcelâĂŹs Smart Recalculation Engine. http://www.decisionmodels.com/calcsecrets.htm

[24] Bonnie A Nardi and James R Miller. 1990. *The spreadsheet interface: A basis for end user programming*. Hewlett-Packard Laboratories.

[25] Saket Navlakha, Rajeev Rastogi, and Nisheeth Shrivastava. 2008. Graph summarization with bounded error. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. ACM, 419–432.

[26] Aditya Ganesh Parameswaran, Hyunjung Park, Hector Garcia-Molina, Neoklis Polyzotis, and Jennifer Widom. 2012. Deco: declarative crowdsourcing. In *Proceedings of the 21st ACM international conference on Information and knowledge management*. ACM, 1203–1212.

[27] Vijayshankar Raman, Andy Chou, and Joseph M Hellerstein. 1999. Scalable Spreadsheets for Interactive Data Analysis.. In *1999 ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery*.

[28] Peter Sestoft. 2014. *Spreadsheet Implementation Technology: Basics and Extensions*. The MIT Press.

[29] Ben Shneiderman. 1983. Direct Manipulation: A Step Beyond Programming Languages. *IEEE Computer* 16, 8 (1983), 57–69. http://dblp.uni-trier.de/db/journals/computer/computer16.html#Shneiderman83

[30] Nan Tang, Qing Chen, and Prasenjit Mitra. 2016. Graph stream summarization: From big bang to big crunch. In *Proceedings of the 2016 International Conference on Management of Data*. ACM, 1481–1496.

[31] MICROSOFT UK ENTERPRISE TEAM. 2015. How finance leaders can drive performance. https://enterprise.microsoft.com/en-gb/articles/roles/finance-leader/how-finance-leaders-can-drive-performance/.

[32] Charles Williams and Linda Caputo. 2017. Excel performance: Improving calculation performance. https://docs.microsoft.com/en-us/office/vba/excel/concepts/excel-performance/excel-improving-calcuation-performance.

[33] Charles Williams and Linda Caputo. 2017. Speeding up calculations and reducing obstructions. https://docs.microsoft.com/en-us/office/vba/excel/concepts/excel-performance/excel-improving-calcuation-performance#speeding-up-calculations-and-reducing-obstructions.

[34] Andrew Witkowski et al. 2005. Advanced SQL modeling in RDBMS. *ACM Transactions on Database Systems (TODS)* 30, 1 (2005), 83–121. http://dl.acm.org/citation.cfm?id=1061321

[35] Andrew Witkowski et al. 2005. Query by excel. In *VLDB*. VLDB Endowment, 1204–1215. http://dl.acm.org/citation.cfm?id=1083733

A HANDLING SCALE

While working with large datasets, the main-memory based design of current spreadsheet systems fundamentally limits their scalability [6]. Thus, to achieve interactivity at scale, systems must operate beyond main memory limits. In this section, we discuss a *two-tier memory model*, wherein the cost of data retrieval from storage factors into the unavailable time.

We discussed the techniques in the earlier sections in the context of main-memory systems, wherein computation time is the dominant concern. These techniques still provide a

significant improvement in a two-tier setting, but can be further improved if fetching costs are taken into account. This section discusses how we adapt the techniques described in the earlier sections to work with these additional cost concerns.

A.1 Two-tier Memory Model

We define the two-tier memory model as follows:

Definition 14. *The* two-tier memory model *contains two tiers of memory:*

- *the* main-memory*, which is limited in size, but allows fast data access; the application interfaces with this tier;*
- *the* storage*, which is large, but data access is slow; and the application does not directly interact with this tier.*

Under the two-tiered memory model, the spreadsheet data is persisted in the storage tier—thus any changes must be eventually reflected there. We assume that the storage tier is not accessed directly by a spreadsheet application but rather via the main-memory in a read/write-through manner, meaning *(i)* if the application requires a data not present in the main-memory, then the data is fetched from the storage tier, stored in the main-memory, and returned to the application; and *(ii)* when the application updates data, it is first updated in the main-memory, and the control is returned to the application only when the update is also refelcted in the storage. In particular, for DataSpread, we use a relational database for the storage tier—this enables DataSpread to go beyond main-memory limitations while working with large datasets.

A.2 Techniques under Fetching Cost

The data transfer between the two tiers is time consuming; we incur a fetching cost each time we bring a cell from the storage tier into the main-memory tier. Often, these costs dominate the computation cost. This section explores how the techniques of dependency graph compression and scheduling change when the main cost concern is fetching.

Fast Dependency Identification. The dependency graph is asymptotically as large as the the spreadsheet size, and therefore is persisted in the storage tier. Identifying dependencies naively by traversing the graph is inefficient in main-memory systems, and can be even worse in the two-tier memory model; each step of the traversal requires a query to the storage layer. Even if the query is done in a breadth-first search fashion, such that each step (of the same distance from the origin) is done in a batch, the number of steps required is equal to the length of the longest chain in the graph. The result of fetching for each step in the chain can be far too costly for our purposes.

The compressed dependency table, as presented for main-memory systems, can also be used in the two-tier memory model. The dependency table can be stored as a relational table in the storage layer. A query for dependents of a cell u is often a straightforward lookup in the dependency table, avoiding the aforementioned issues with graph traversal.

Computation Scheduling. Here we introduce a new version of the scheduling problem, which we adapt to include the cost of fetching the direct precedents of dirty cells from storage, as those values are required for computation.

Problem 5 (Computation Scheduling with Fetching Costs). *Given a set of dirty cells (Δ), their direct precedents $P = \{p \mid$ the direct dependency $p \longrightarrow c$ exists for some $c \in \Delta\}$, along with the dependencies among the cells in $\Delta^+ = \Delta \cup P$, determine an order $c_1, \ldots, c_n$ of all the cells in Δ^+ that minimizes the unavailability metric, i.e., $\sum_{c_i \in \Delta} \text{dirty}(c_i)$, where $\text{dirty}(c_i) = \sum_{j=1}^{i} \text{cost}(c_j) = \text{dirty}(c_{i-1}) + \text{cost}(c_i)$, under the Dependency constraint.*

Because both the dirty cells and their precedents need to be fetched from storage, all cells in Δ^+ are relevant in the fetching order. However, the unavailability metric only concerns those of the dirty cells Δ.

We shall show that the Computation Scheduling with Fetching Costs problem is NP-Hard.

Theorem 2. *Computation Scheduling with Fetching Costs is* NP-Hard.

Scheduling for the two-tier memory model can be done in a similar fashion as on-the-fly weighted scheduling for main-memory systems. However, the cost function $\text{cost}(c)$ for a cell c must be adjusted, since fetching costs dominates computation costs in the two-tier context.

In addition, "locality" becomes important. Systems often perform data fetching in blocks, and therefore scheduling computation of formulae in the same block together can be beneficial. Working on formulae whose operands are already fetched into the cache is less costly; switching to completely unrelated formulae may result in cells being evicted from limited-size cache, requiring refetching. These concerns can be factored into in the cost function. It may require dynamic updates as the cache changes in the same way weights are updated when the viewport moves.

B PROOFS AND ALGORITHMS

This section provides proofs for hardness claims stated in the main paper, in addition to pseudocode for some of the key algorithms.

B.1 Proof of Theorem 1

We shall prove Theorem 1 by providing polynomial-time reduction from the Polygon Exact Cover problem, a known NP-Hard problem, defined as follows [14].

Input: a set of rectangular regions R, and an integer k
Output: a cover R' of the union of rectangular regions in R, where $|R'| \leq k$

$R' \leftarrow R$;
while $|R'| > k$ **do**
  Let r_1 and r_2 be two rectangular regions in R' where the bounding box of $r_1 \cup r_2$ introduces the smallest false positives out of all such combinations;
  Let r be the smallest of such a bounding box;
  $R' \leftarrow (R' \setminus \{r_1, r_2\}) \cup r$
end
return R'

Algorithm 1: Incremental Greedy Compression

Input: a set dirty cells Δ_u, and two integers k and m
Output: a computation schedule of the cells in Δ_u

$D \leftarrow \Delta_u$;
$P \leftarrow \emptyset$;
Let S be an empty schedule;
while $|D| > 0$ **do**
  $P' \leftarrow$ subset of $k - |P|$ cells drawn from D;
  $D \leftarrow D - P'$;
  $P \leftarrow P \cup P'$;
  Compute cost of each element in P.;
  $M \leftarrow$ the m elements of P with lowest cost;
  $P \leftarrow P - M$;
  Let M' be the union of the dirty precedents of c for $c \in M$;
  Append $M \cup M'$, in topological order, to S;
end
return S

Algorithm 2: On-the-fly Scheduling algorithm

Problem 6 (Polygon Exact Cover). *Given a simple and holeless orthogonal polygon P and an integer k, is there a set of at most k axis-aligned rectangles whose union is exactly P?*

Proof of Theorem 1. Given a simple and holeless orthogonal polygon P and an integer k in a Polygon Exact Cover instance, perform a rank-space reduction on the coordinates of P; that is, change the actual coordinates into values in $\{1, \ldots, n\}$, where n is the size (number of vertices) in P, such that the coordinates are in the same order. Translate the polygon in the new coordinates into a set C of cells. (Note that the representation size goes up quadratically.) There is a set of at most k axis-aligned rectangles whose union is precisely P if and only if C has a cover whose size does not exceed k and cost is at most $|C|$ (has no false positives), following the natural coordinate mapping between rectangles and ranges. □

B.2 Proof of Theorem 2

Problem 5 is a generalization of the Scheduling With Supporting Tasks problem [16], which is NP-Hard, defined as follows.

Problem 7 (Scheduling With Supporting Tasks). *Let $A = \{a_1, \ldots, a_m\}$ and $B = \{b_1, \ldots, b_n\}$ be sets of tasks, and $R \subseteq A \times B$ be a relation. All tasks take a unit time to complete. The tasks in A and B are to be scheduled on a single machine that can perform one task at a time, under the restriction that if $(a_i, b_j) \in R$, then task a_i must be completed before task b_j. Tasks in A are* supporting tasks *and are not required to be completed (unless required by other tasks in B). Let* $\text{cost}(b_j)$ *denote the time until task b_j is completed in a schedule. Given c, determine whether there is a schedule to complete all tasks in B within the stated restrictions such that the total cost $\sum \text{cost}(b_j)$ is at most c.*

Proof of Theorem 2. We provide a polynomial-time reduction from Scheduling With Supporting Tasks. For each task t in $A \cup B$, create a corresponding cell $\text{cell}(t)$. For each $(a_i, b_j) \in R$, make $\text{cell}(b_j)$ dependent on $\text{cell}(a_i)$; in other words, if $a_{i_1}, \ldots, a_{i_\ell}$ are the elements of $\{a \mid (a, b_j) \in R\}$, create a formula $\text{cell}(b_j) = \text{cell}(a_{i_1}) + \cdots + \text{cell}(a_{i_\ell})$. Mark all cells corresponding to B dirty; that is, let $D = \{\text{cell}(b_i) \mid b_j \in B\}$ and P be their precedents. It follows that a valid schedule in one problem is also valid on the other (given proper translations between tasks and cells), and the cost metrics of the two problems are identical. □

B.3 Algorithms

Algorithm 1 provides an outline of a greedy graph compression process, as we described in Section 3.3. Algorithm 2 provides an outline of an on-the-fly scheduling heuristic, as we described in section Section 4.2.

C DIRTY CELL DATA STRUCTURES

We have introduced the notion of dirty cells in Section 2.3 and have used this concept throughout the paper. We have mentioned the operations of marking cells and regions as dirty upon an update to a cell and marking them as clean as soon as the values are recomputed or known to be correct. However, we have yet to discuss how such marking operations can be performed efficiently.

A naïve solution involves maintaining a boolean dirtiness flag for each cell. However, when the set of operations is extended to involve more than one cell at a time, in scenarios where a region or an entire sheet needs to be marked as dirty or clean, the naive solution is inefficient. Individually changing flags of the cells in the region takes time proportional to the size of the region, which is undesirable.

In this section, we describe data structures that support operations related to dirty cells and regions. We start with a

primitive solution that works for simpler strategies. We then describe modifications required to support an extended set of operations that are required by the new techniques: Fast Dependency Identification (from Section 3) and Scheduling (from Section 4).

C.1 Supporting Base Operations

Upon updating u, the whole sheet becomes temporarily unavailable, because cells have yet to be determined whether they are dependent on u. During this period, any cell is unavailable regardless of its cleanliness status. The temporary unavailability period ends when dependency identification is complete. A boolean flag is sufficient to record whether the sheet is within such a period.

In order to support incrementally returning results, the data structure which tracks dirtiness must be able to support the following *base operations*.

- isDirty(c) returns true if a cell c is dirty and false otherwise.
- markClean(c) marks the cell c as clean.
- markDirty(c) marks the cell c as dirty.

The query operation in this set, isDirty(), is used to decide whether the value can be displayed to the user. When a cell c is identified as a dependent of an update, markClean(c) is called, and when its value is evaluated, markDirty(c) is called.

While the naïve solution of maintaining a boolean dirtiness flag for each cell seems sufficient for the base operations, there can be issues when concurrent updates exist.

In order to record the order in which operations occur, we assume that there is a synchronous clock from which the system can request the current time.

For each cell c, maintain a timestamp cleanTimestamp(c) and a timestamp dirtyTimestamp(c), storing the latest time at which the cell is marked clean and dirty, respectively. The query isDirty(c) compares whether the latest marking of cell c as clean or dirty happens later. More precisely, isDirty(c) returns true if and only if dirtyTimestamp(c) > cleanTimestamp(c).

The data structure requires $O(|S|)$ memory, where S is the set of cells in the spreadsheet, and $O(1)$ time per operation.

C.2 Supporting Fast Dependency Identification

Fast Dependency Identification provides K_{comp} regions containing all dependents of the updated cell. Cells outside of this region are now determined as clean (assuming they were not dirty earlier). The following operation is now required.

- markRegionDirty(r) marks all the cells in region r as dirty.

The K_{comp} regions obtained from Fast Dependency Identification are marked dirty by calling markRegionDirty(r) on each region r.

We use an R-tree to store dirty regions, which allows us to store rectangular regions in a way that facilitates quick lookup of regions overlapping with a specified region (say, a cell). With each entry r in the R-tree, we store additional information regarding the cleanliness of that region: a cleanTimestamp(r) and a dirtyTimestamp(r).

Thus, in order to mark a cell c as dirty, we add the region to the R-tree with a dirtyTimestamp(r) which reflects the current timestamp. For isDirty(c) queries, we look up the region in the R-tree that intersects with r and has the largest timestamp. Such a timestamp determines the latest time c is marked dirty as part of a region. Comparing such a timestamp with the individual timestamps cleanTimestamp(c) and dirtyTimestamp(c) determines whether a cell is currently dirty.

The runtime to perform updates and lookups in an R-tree is $O(\log R)$, where R is the number of entries in the tree. Normally, R increases by K_{comp} after each update. Note that no procedures for removing regions from the R-tree is present, even if all the cells in that region are cleaned at a later time. Thus the tree grows as more operations are performed, deteriorating performance. A method to mitigate this issue is to periodically scan the R-tree for regions that can be safely removed; a better method that also integrates another operation will be described later.

C.3 Supporting Scheduling

To support scheduling, we need the ability to retrieve cells that are currently dirty from the data structure.

- getNextDirtyCell() returns a dirty cell.

In order to support getNextDirtyCell(), we maintain a progress pointer that scans through a dirty region in the R-tree, and yields cells that are dirty. At any time, there is a single *active region* r in the R-tree, with a pointer p pointing to a cell within the region r. When getNextDirtyCell() is called, the pointer scans through the region in an order, say, a row-major order, starting from its current position (where it left off from last time), returning the cell if it is dirty, determined by isDirty(c).

The invariant is that all cells in r that are before where p is currently pointing must have been cleaned after dirtyTimestamp(r), for otherwise the pointer would not have moved past that cell. Therefore, once p have scanned through the entirety of r, the region r can be safely removed. Note that it is possible for a cell that p have moved past to become dirty again; however, the dirtiness event would be captured either by a different range or an individual cell timestamp, and it remains safe to remove r.