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

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

ABSTRACT

Spreadsheet systems enable users to store and analyze data in an intuitive and flexible interface. Yet the scale of data being analyzed often leads to spreadsheets hanging and freezing on small changes. We propose a new asynchronous formula computation framework: instead of freezing the interface we return control to users quickly to ensure interactivity, while computing the formulae in the background. To ensure consistency, we indicate formulae being computed in the background via visual cues on the spreadsheet. Our asynchronous computation framework introduces two novel challenges: (a) How do we identify dependencies for a given change in a bounded time? (b) How do we schedule computation to maximize the number of spreadsheet cells available to the user over time? We bound the dependency identification time by compressing the formula dependency graph lossily, a problem we show to be NP-Hard. A compressed dependency table enables us to quickly identify the spreadsheet cells that need recomputation and indicate them as such to users. 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. We have incorporated asynchronous computation in DataSpread, a scalable spreadsheet system targeted at operating on arbitrarily large datasets on a spreadsheet frontend.

ACM Reference Format:
Mangesh Bendre, Tana Wattanawaroon, Kelly Mack and Kevin Chang, Aditya Parameswaran. 2019. Anti-Freeze for Large and Complex Spreadsheets: Asynchronous Formula Computation . In *2019 International Conference on Management of Data (SIGMOD '19), June 30-July 5, 2019, Amsterdam, Netherlands.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3299869.3319876

SIGMOD '19, June 30-July 5, 2019, Amsterdam, Netherlands

ACM ISBN 978-1-4503-5643-5/19/06…$15.00
https://doi.org/10.1145/3299869.3319876

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 [24]. From personal bookkeeping, to complex financial reports, to scientific data analysis, the ubiquity of spreadsheets as a computing system is unparalleled. Nardi and Miller [26] identify two reasons for their success: an intuitive tabular *presentation*, and in-situ formula *computation*. In particular, formula computation enables end-users with minimal programming experience to be able to interrogate their data and compute derived statistics.

However, the sheer volume of data available in a host of domains exposes the limitations of traditional formula computation. A recent study exploring forum posts on Reddit describes several instances of Microsoft Excel becoming unresponsive while computing formulae [22]. One user posted that complex calculations on Excel can take as long as four hours to finish, during which the 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 personal life (with a complex network of formulae), and periodically cull data to keep the size manageable, but they still had trouble with computation:

> *"…the spreadsheet locks up during basic calculations, the entire screen freezes…"*

The chief culprit for unresponsiveness is that, in traditional spreadsheet systems, every change to 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 in data exploration have shown that even delays of 0.5s can lead to fewer hypotheses explored [20], so *synchronous computation* is not desirable. As is evident in the anecdotes

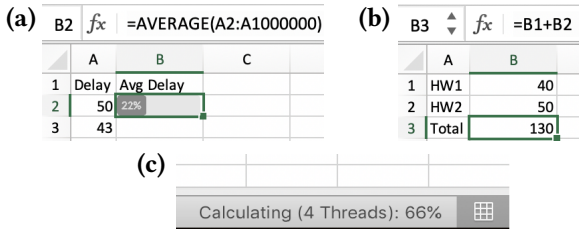

Figure 1: (a) Our asynchronous approach maintains interactivity and consistency by showing computation status instead of a stale value. (b) Manual computation achieves interactivity but violates consistency. (c) Automatic calculation achieves consistency but keeps the user interface non-responsive for the duration of the computation.

above, users get frustrated waiting for multiple seconds to minutes to get back control of the spreadsheet.

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) 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 B2 and B3 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[1].

While the asynchronous computation approach dramatically minimizes the time during which users cannot interact with the spreadsheet, it requires a fundamental redesign of the formula computation engine, thanks to two primary challenges: *dependencies*, and *scheduling*:

[1]Thus, asynchronous computation ensures that spreadsheets never hang, make you give up, let you down, or desert you [1].

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 formula dependency graph 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 to identify dependent cells. Unfortunately, for computationally heavy spreadsheets, a traditional dependency graph that captures formula dependencies at the cell-level [34] can be large, so identifying dependencies can take arbitrarily long.

To enable fast lookups of dependencies, we introduce *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 compactly represent formula dependencies, minimizing false positives. The size of our representation impacts the dependency lookup time, and the false positives impact the formula computation time, both impacting unavailability. We show that optimal graph compression to minimize unavailability is NP-Hard. We propose approximate techniques for compressing the dependency graph and its maintenance.

Scheduling: Challenges and Approach

Once we have identified the dependent cells (with possibly a few false positives), we then need to compute them efficiently to decrease unavailability as quickly as possible. In asynchronous computation, we incrementally return the values of the dependent cells as soon as they are computed. When adhering to a schedule, or an order in which the cells are computed, the time that a dependent cell is unavailable comprises the time *(i)* waiting for prior cells in the schedule to complete, and *(ii)* computing the cell itself. 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 also respect dependencies: the computation of a cell must be scheduled only after all the cells that it depends on are computed. In fact, not only is finding an optimal schedule NP-Hard, merely obtaining a schedule can be prohibitively expensive, as it requires traversal of the entire dependency graph—this can negate the benefits 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. We further extend our scheduling technique to a weighed version that prioritizes cells that the user is likely to visit next.

Putting It All Together

We incorporate our asynchronous computation model into a scalable spreadsheet system that we are building, DataSpread [8, 9], 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, wherein data resides in an underlying relational database and is fetched on-demand into main-memory. This introduces additional challenges that go beyond those found in traditional main-memory-resident spreadsheets. (Note, however, that our techniques for decreasing unavailability apply equally well to traditional spreadsheets as well as DataSpread.)

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. We propose the novel *unavailability* metric to 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 efficient techniques for compression and maintenance of this graph.

3. Computation Scheduling. In Section 4, we discuss the importance of finding an efficient formula computation schedule. Since obtaining a schedule is expensive, we propose *on-the-fly scheduling*. Our algorithm can be extended to a weighted variation that prioritizes for the user's viewport.

4. DataSpread **Prototype.** In Section 5, we describe DataSpread, a scalable spreadsheet system we built that incorporates the ideas discussed in this paper.

5. Evaluation. Throughout the paper, we provide illustrative experiments to demonstrate individual ideas. In Section 6, we discuss our experimental setup and provide a thorough evaluation of asynchronous computation on spreadsheet structures drawn from real-world spreadsheets, studying the impact of the dependency structure and its size, the back-end data store, algorithmic parameters, and building blocks (whether scheduling or compression is used).

2 ASYNCHRONOUS COMPUTATION

We propose asynchronous computation to address the interactivity issues of traditional spreadsheet systems. We first define key spreadsheet terminology. We then introduce two principles that influence the design of our model, as well as new concepts for our proposed model.

For simplicity, we explain the concepts and techniques in the context of standard main-memory-based spreadsheet systems, where, the cost of data retrieval is negligible compared to the cost of formula evaluation. In Appendix C, we extend our techniques to two-tier memory systems wherein data retrieval cost is significant. While the techniques discussed here 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.

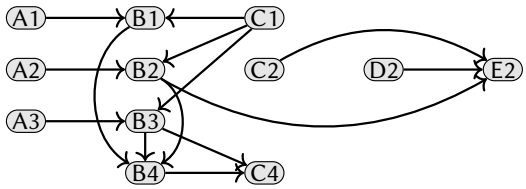

Figure 2: A dependency graph that captures the dependencies of Example 1 at the granularity of cells.

2.1 Standard Spreadsheet Terminology

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

Spreadsheet Components. A *spreadsheet* consists of a collection of *cells*, each 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 corresponding expression. 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. We focus only on computation in this paper.

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).*

Here, the cell B4 has a formula SUM(B1:B3), which indicates that B4's value depends on B1:B3's values. 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. 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 a* precedent *of v, and v is a* dependent *of u. We denote the set of dependents of u as Δ_u.*

One can construct a conventional *dependency graph* of direct dependencies. Figure 2 depicts the graph for Example 1 at the granularity of cells. Each vertex corresponds to a single cell, *e.g.*, A1, while edges are direct dependencies. For example, the edge from A1 to B1 indicates a direct dependency due to formula A1*C1 in cell B1. The dependencies of a cell u are the vertices that are reachable from u in the graph. For example, cell B1 has B4 and C4 as dependents. Since cyclic dependencies are forbidden in spreadsheets, the dependency graph is acyclic. As this graph captures dependencies at the granularity of cells, it grows quickly when the ranges mentioned in the formulae are large [34]. 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

Spreadsheet systems must 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, rapidly, and provide results as soon as possible. Thus, we introduce two design principles for our solution, followed by two computation models.

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.*

Synchronous Computation Model. Traditional spreadsheet systems adopt *synchronous computation*, where, on updating u, the entire spreadsheet becomes unavailable during the evaluation of cells that are dependent on u. The spreadsheet system waits for the entire computation to complete before providing updated values to the user—thereby adhering to the consistency principle. However, when the number of cells dependent on u is large, this model sacrifices interactivity, with often minutes of unresponsiveness.

Asynchronous Computation Model. To provide interactivity in addition to consistency, we propose asynchronous computation. Here, on updating u, the cells dependent on u are computed asynchronously in the background without blocking the user interface. To satisfy consistency, 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 on updating A1 the computation of cell B2 is performed in the background and the 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. For our 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. 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). *A cell u is* clean *if and only if (i) all of u's precedents are clean and (ii) u's evaluated value has been computed based on its formula and the clean values of its precedents.*

Definition 4 (Dirty Cell). *A cell u is* dirty *if and only if (i) at least one of u's precedents are dirty or (ii) u's evaluated value has not been computed based on its formula and the clean values of its precedents.*

Adhering to the consistency principle, *(i)* for clean cells, the evaluated value is displayed on the user interface, and *(ii)* for dirty cells, the cell displays a progress bar depicting the status of its computation, 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. A dirty cell may in fact 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, a cell is said to be unavailable if it cannot be used by the user for various reasons, defined as follows.

Definition 5 (Unavailable Cell). *A cell c is* unavailable *if and only if (i) c is determined to be dirty, (ii) the system has not yet determined if c is in the clean or dirty state, or (iii) the user interface is unresponsive.*

Overall, we aim to provide users with cells 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. Incrementally computing and marking cells as clean allows the number of unavailable cells to gradually decrease over time.

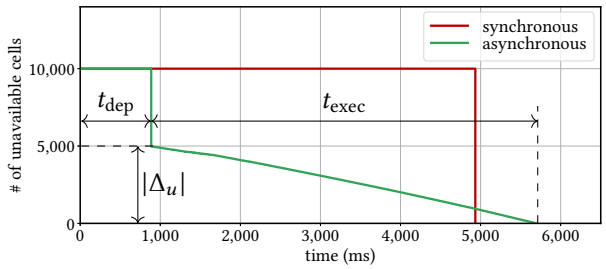

Figure 3: Comparing the unavailability of synchronous and asynchronous models. For the asynchronous model, t_{dep} is the dependency identification time, Δ_u is the set of cells that are determined to be dependent on u and thus need computation, and t_{exec} is the computation time for these cells.

Unavailable and Dirty Time. Quantifying the time a cell is unavailable for the user to act on is an important factor for quantifying the usability of the spreadsheet. Similarly, the dirty time is the time a cell spends in the dirty state. We formalize these notions below.

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

For the above definitions as well as subsequent ones, the spreadsheet state is a hidden (implicit) parameter that we omit to keep the notation simple.

Unavailability. To quantitatively evaluate different computation models, we introduce the *unavailability* metric, defined as the area under the curve that, for a computation model, plots the number of unavailable cells over time.

Definition 7 (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 on over time. For the synchronous computation model, for the entire time the user interface is unresponsive, all of the cells 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 to determine if c is clean or dirty.

Illustrative Experiment 1: Asynchronous vs. Synchronous Computation. The goal of this experiment is to compare the unavailability of the asynchronous and synchronous computation models. We use a synthetic spreadsheet that follows the "Rate" dependency structure that we describe in Section 6. The spreadsheet has a total of 10,000 cells, of which 5,000 are formulae dependent on cell A1 via the formula Ci=A1*Bi for $i \in \{1 \ldots 5{,}000\}$. This dependency structure is inspired from real-world spreadsheets that have multiple formulae dependent on a single cell encoding some common information, like interest rate. We adopt a conventional dependency identification mechanism as described in Section 2.1 and a naïve schedule for computing cells—we develop better variants later. We update A1's value and plot the number of unavailable cells as a function of time for both computation models in Figure 3. Synchronous computation (in red) performs poorly under unavailability, since it keeps the interface unresponsive for the entire duration of computation of all of the dependent cells. Asynchronous computation (in green) performs better in terms of unavailability, since it allows users to interact with most of the spreadsheet cells while performing calculations asynchronously.

Upon updating A1 (at $time = 0$), the asynchronous model first identifies the 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 the asynchronous model has determined which cells are clean and which cells are dirty, it returns the clean cells to the user. Thus, the number of unavailable cells drops to 5,000 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 synchronous model returns control of all of the cells to the user (4,900 ms). 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; the synchronous model does not perform this step. Overall, the area under the green curve is less than that under the red curve, and therefore the asynchronous model performs better in terms of unavailability.

> *Takeaway: Asynchronous computation improves spreadsheet usability without forgoing correctness, by (i) quickly returning control and (ii) incrementally making cells available.*

While this experiment shows that asynchronous computation 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

We now propose our first technique for decreasing unavailability: *identifying dependencies in a bounded time*. That is, we aim to reduce t_{dep} in Figure 3—the time during which the interface is unresponsive for asynchronous computation. Reducing t_{dep} is particularly crucial when the update affects a small number of cells relative to the size of the spreadsheet.

3.1 Motivation and Problem Statement

After a user updates a cell u, 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 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 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 the 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. Modifications to the data structure, i.e., inserts and deletes, should be quick (again, preferably in bounded time).*

One method of capturing dependencies is to maintain a *dependency graph.* Rather than recording dependencies between individual cells (Figure 2), we can capture dependencies between regions, substantially reducing the size of the graph. Figure 4 shows the dependency graph for Example 1. A 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.

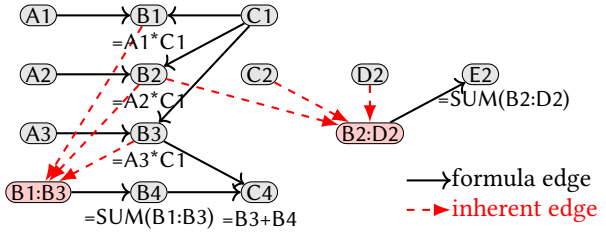

Figure 4: Dependency graph capturing dependencies between regions thus reducing the graph size.

One naïve approach to represent this graph is to persist the *formula edges* as adjacency lists. 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 B1. When storing the *inherent edges*, rather than storing them 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. Consider 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.

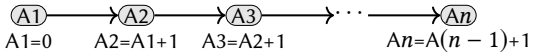

Figure 5: Long Dependency Chain

3.2 Compressed Dependency Table (CDT)

To overcome this 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.

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 an unnecessary recalculation of c_{FP}. In other words, the dependency table is *false positive tolerant*—false positives do 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,

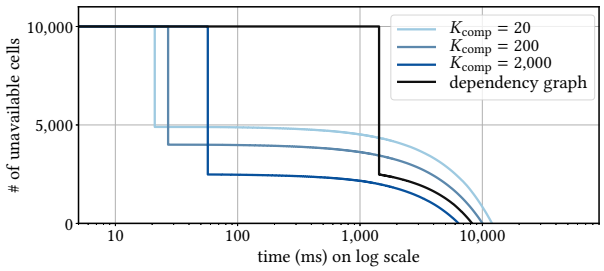

Figure 6: Comparing unavailability for a system using dependency graph vs dependency table with varying K_{comp}.

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 8 (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 in constant time, we limit the size of the cover to a constant K_{comp}. In Figure 7(b), K_{comp} is 2. Varying K_{comp} can significantly impact unavailability, due to the trade-off between the time 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). The less time we spend identifying dependencies, the smaller the K_{comp} and the more false positives we introduce. This increase in false positives causes the cost of the cover, and therefore the total number of dirty cells following dependency identification, to increase. Ultimately, we need a value of K_{comp} that minimizes unavailability. Due to the flat nature of CDT, a query is a simple non-recursive lookup, and thus cycles introduced by compression do not impact querying for dependencies.

(a)

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
⋮	⋮

(b)

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

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

Illustrative Experiment 2: Impact of K_{comp}. In this experiment, we demonstrate the benefit of using a CDT instead of a traditional dependency graph, as well as the impact of varying K_{comp}. We use a synthetic spreadsheet targeted at "stress-testing" our CDT approach; details can be found in Section 6. Our spreadsheet is a "hard" modification of the "Rate" dependency structure from Experiment 1. The spreadsheet has 10,000 cells, out of which 5,000 cells contain formulae. Out of the 5,000 formulae cells, 50% of the cells follow the "Rate" structure and are dependent on A1 (once again indicating a common rate parameter, as in Experiment 1), that we intersperse with cells that are independent of A1, where each cell performs a summation using the SUM function of 50 non-formula cells. The reason for interleaving formulae this way is because more compression will lead to more false positives (*i.e.*, the cells not dependent on A1), and there is a penalty for having to compute them.

We update A1's value and plot the number of unavailable cells with respect to time for asynchronous computation—see Figure 6. Due to a large number of dependencies, dependency identification using the dependency graph (in black) 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 dark blue curve where K_{comp} is 2,000—the dependency identification takes around 60 ms. On the other hand, the light blue 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. These are the heavy non-A1-dependent cells, requiring a lot of computation each. Therefore, even though the $K_{comp} = 20$ curve returns control to the user in a few milliseconds, it takes more time to clean all the dirty cells. In this "hard" example, $K_{comp} = 2{,}000$ (in dark blue) performs the best, as its curve encloses the least area. Note that real-world spreadsheets typically do not have such problematic dependency structures, and therefore even heavy compression does not lead to a large number of false positives, as we will see in Section 6.

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 CDT, 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 A.

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

Greedy Heuristic. Since efficiently finding the best graph compression is hard, we propose a greedy algorithm: 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 compression is given in Algorithm 1.

Input: a set of rectangular regions R, and an integer k
Output: a cover R' of 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

3.4 Maintenance of the CDT

We now discuss how to update the CDT when formulae are changed—leading to edge deletions or additions in the dependency graph—as quickly as possible. At a high level, deletions of edges are handled lazily, since such a deletion method can only lead to false positives but no false negatives. Additions require more care, since the addition of a single edge in the dependency graph can, in the worst case, cause $O(n)$ dependencies to be added to $O(n)$ entries in the CDT, by connecting two disconnected graph components. Rather than eagerly expanding the dependencies induced by new connections in the CDT, we handle such new connections by annotating them with a "must-expand" flag, traversing them when the need arises, populating the dependencies lazily in the background. In this manner, by having the flag, we ensure no false negatives. See Appendix B for details.

4 COMPUTATION SCHEDULING

We now propose our second technique for decreasing unavailability: *computation scheduling*. After updating a cell u, we need to find an efficient schedule for computing the cells that depend on u. 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 then extend this solution to prioritize computation based on what users are currently seeing.

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.

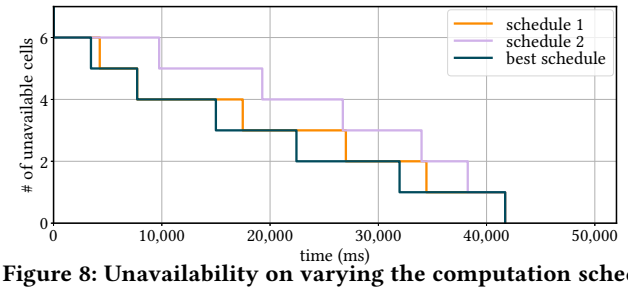

Figure 8: Unavailability on varying the computation schedule (dependency identification time of 20 ms is shown but is dominated by computation)

Illustrative Experiment 3: Computation Schedule. The goal of this experiment is to demonstrate the importance of scheduling. Here, we consider a synthetic spreadsheet that adopts a variation of the "RunTotalSlow" dependency structure (described in Section 6), where there are precisely six formula cells that perform summations using the SUM function of increasing sized ranges, namely A1:A2000, A1:A4000, ..., A1:A12000, simulating varying complexities. This structure is meant to capture computation over increasing sizes of data, such as a running total over transactions (*e.g.*, sum of sales for month 1, month 1 and 2, month 1, 2, and 3, and so on.) We selected this configuration because the workload required for each computation varies per cell, and therefore the choice of schedule of computations can significantly impact the unavailability metric for the sheet. For this simple spreadsheet, we update A1's value and plot the number of unavailable cells on the y-axis over time—see Figure 8. 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 significantly 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? 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, thus computation cost is

independent of schedule; we will discuss its impact later and relax this assumption.

Definition 9 (Cost). *The cost of a cell c, $\text{cost}(c)$, is the amount of time needed to compute the evaluated value of c, assuming the values of its precedents are already computed.*

Note that for synchronous computation, computation scheduling is unimportant. The total evaluation time for all cells dependent on u is $\sum_{c\in\Delta_u}\text{cost}(c)$, where Δ_u is the set of cells dependent on u. Therefore, in the synchronous model, since all cells remain unavailable until all of the computation is completed, the unavailable time for every cell in the spreadsheet is equal to $t_{\text{dep}} + \sum_{c\in\Delta_u}\text{cost}(c)$, 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.

However, when we incrementally return cells in the asynchronous model, $\text{dirty}(c)$ is not the same across all cells because a cell becomes clean as soon it is evaluated. Therefore, choosing the order in which cells are computed is crucial. 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. On 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 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 10 (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 11 (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* $\text{dirty}(c_i) = \sum_{j=1}^{i}\text{cost}(c_j) = \text{dirty}(c_{i-1}) + \text{cost}(c_i)$.

We formalize our scheduling problem as follows, which is shown as NP-Hard by Lawler [18].

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, on updating u, merely obtaining a schedule can be expensive. The dirty time (Definition 11) 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 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. 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 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 direct precedents provides a good approximation. We can determine the number of precedents by looking at a cell's formula.

Definition 12 (Cost Approximation). *The approximate cost of a cell u, denoted by $\text{cost}^*(u)$, is $|P_u|$, where P_u is the set of u's direct precedents.*

Other alternatives, such as the number of dependents or precedents up to k hops away, could also be employed, trading off the accuracy of cost^*, with the time it takes to estimate cost^*. We opt for the number of direct precedents here for simplicity, leaving a thorough evaluation of these variants for future work. On updating u, we perform on-the-fly scheduling as follows. We draw k cells from Δ_u into a 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 require 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 schedule the chosen m cells for computation obeying the dependency constraint, dirty precedents of each of them must be computed before the cell itself can be computed. To discover dirty precedents of a cell c, one can use a typical (reverse) graph traversal algorithm to find cells that can reach c. If a precedent p of c is found to be clean, there is no need to traverse further to precedents of p, because they must also be clean (by Definition 3). The dirty precedents, once discovered, must then be computed in a 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 computed and 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, instead looking at collections of cells at a time. Algorithm 2 summarizes the on-the-fly scheduling pseudocode.

Input: a set of 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; $P \leftarrow P - M$;
  $M' \leftarrow$ union of the dirty precedents of $c, \forall c \in M$;
  Append $M \cup M'$, in topological order, to S;
end
return S

Algorithm 2: On-the-fly Scheduling algorithm

4.3 Weighted Computation Scheduling

Due to limited screen real estate, users do not see all the cells of a spreadsheet at the same time. We define the rectangular range of cells that a user can interact with at any given time as the *viewport*. The user can change the viewport either by scrolling or jumping to the desired part of the spreadsheet.

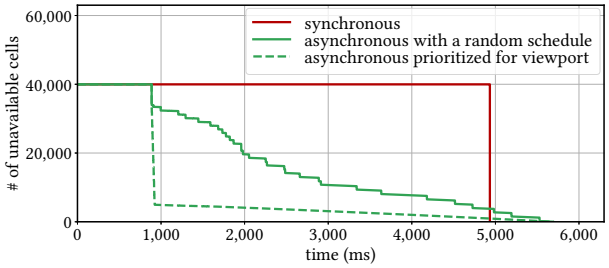

Figure 9: Weighted unavailability comparing synchronous computation models and asynchronous computation model with and without viewport prioritization.

Since users can only view the cells 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 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 *weighted unavailability* modifies our previous Definition 7 such that if a high-weight cell is left 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* weight($c$) *is the weight of* c, unavailable(c) *is the unavailable time for* c, *and* S *is the set of all cells within the spreadsheet.*

We now formalize a weighted variation of our computation scheduling problem.

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 adheres to the dependencies and 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* 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 9 showing a weighted variation of Figure 3. The spreadsheet is the same as Experiment 1, except that, here, we assign a weight of 1,000 for the 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 that prioritizes cells in the viewport (green dashed) 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 9, 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, we modify Algorithm 2 to prioritize the cells by $^{\text{cost}^*(c)}/_{\text{weight}(c)}$. Additionally, we dynamically update the cell weights when the user changes their viewport by scrolling. Furthermore we can also modify Algorithm 2 to first pick up the cells that are within the viewport.

5 THE DATASPREAD SYSTEM

DataSpread is our spreadsheet system that utilizes the techniques in this paper [8–10]. So far, we have focused on main-memory resident systems. DataSpread not only takes the available main-memory into account, but also uses disk storage to handle spreadsheets at scale. In Appendix C, we discuss how the solutions discussed change when we consider the *two-tiered memory model* employed by DataSpread.

At a high level, DataSpread is designed to *unify the capabilities of spreadsheets and databases to provide an intuitive direct-manipulation* [31] *interface for managing big data*. DataSpread is scalable—the current prototype supports interactive browsing of *billion cell spreadsheets* on standard desktop hardware. Support for datasets of this size is only possible by going beyond main-memory limitations, using the back-end database as a paging solution: data is fetched *on-demand* when triggered by a user action (like scrolling), or a system action (like formula computation). Specifically, DataSpread's back-end utilizes a PostgreSQL database, React framework [4] for the front-end, and ZK Spreadsheet [6] for the formula engine. The system architecture is described further in Appendix D. The prototype, along with its source code, documentation, and user guide, can be found at http://dataspread.github.io.

6 ADDITIONAL EXPERIMENTS

In this section, we describe our setup for all the experiments in this paper and provide additional experiments to quantify the benefit of *(i)* asynchronous computation, and *(ii)* individual techniques: dependency identification and computation scheduling, for spreadsheets with different structures.

6.1 Experimental Setup

We now describe the experimental setup, specifically, *(i)* the test cases, *(ii)* the configurations, and *(iii)* the environment used for running the experiments.

Test Cases. To generate our spreadsheet test cases, we first conducted a survey in a university asking users to send us their largest, most complex spreadsheets. We combined the tens of spreadsheets we received with hundreds from the publicly available Enron dataset [16], and then manually examined sheets that are formula-dense in this collection. We identified three common usage patterns capturing three different dependency structures, each parametrized by n:

- **Rate**: This dependency structure captures the use case of a column derived from another based on a rate value, such as conversion or interest rate, or grading weight. When the rate changes, the derived column must be recomputed. This case has many cells dependent on the changed rate (big fan-out), but short dependency chains.
- **RunTotalFast**: This dependency structure captures the use case of overlapping partial calculations, such as a running total (*e.g.*, sum of sales from day $1 \ldots i$). It uses an efficient implementation that avoids redundant computation. Each formula cell depends on the previous formula cell and one new value. Thus, a long chain of dependencies is introduced to the dependency graph, and there is only one order in which the cells can computed.
- **RunTotalSlow**: This dependency structure serves the same use cases as RunTotalFast, but uses a naïve implementation, invoking overlapping calls to SUM. When a value changes, all the relevant totals must be recomputed. This case features a large computation workload ($O(n^2)$), and dependent cells with a large skew in their computation workload ranging from $O(1)$ to $O(n)$.

These test cases and the associated dependency graphs are depicted in Figure 10. Characteristics of the structures are summarized in Table 1. In all the cases, we change the value of a single cell (A1), which triggers recomputation.

Model Configurations. We compare variants of our asynchronous computation model against a synchronous computation baseline. Different choices of individual techniques can be used for asynchronous computation (see Table 2):

- **Compression**: use graph compression from Section 3 with $K_{\text{comp}} = 20$ ("on") or skip compression ("off"). We study the impact of varying K_{comp} later on.
- **Scheduler**: use the proposed on-the-fly scheduling algorithm from Section 4.2 ("on") or use an algorithm that schedules computation in an arbitrary fashion while still obeying dependencies ("off").

Besides the choice of computation model and its configuration, the rest of the framework, *i.e.*, the code used to evaluate individual formulae, is identical across experiments.

Environment. We implemented asynchronous computation along with graph compression and computation scheduling in Java within DataSpread. We ran all of our experiments on a workstation running Windows 10 on an AMD Phenom II X6 2.7 GHz CPU with 32 GB RAM. To eliminate the impact of communication between front-end and back-end, we designed our test scripts as single threaded

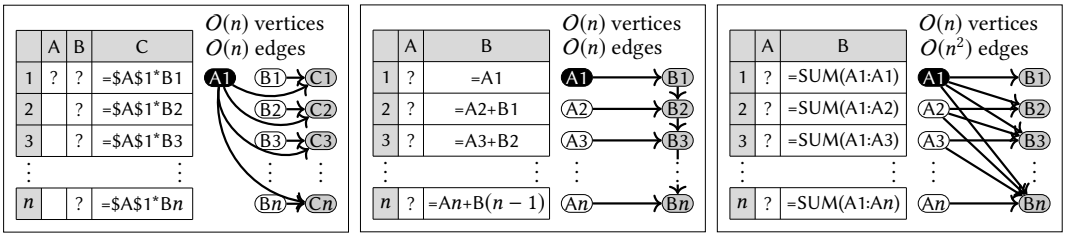

Figure 10: Popular dependency structures found in real-wold spreadsheets, along with their dependency graphs.

independent applications that directly utilize DataSpread's back-end. There are two data store configurations: *(i)* "main-memory", where cell values are all stored main-memory, and *(ii)* "two-tier-memory", which uses PostgreSQL 10.5 as a back-end data store—see Section 5 and Appendix C. While the previous experiments used the "two-tier-memory" setting, the experiments in this section will use the main-memory setting, allowing us to scale experiments. We study the impact of the data store later on. For all the experiments discussed in this section, we ran 10 trials and then took the median.

	Rate	RunTotalFast	RunTotalSlow
max direct precedents	$O(1)$	$O(n)$	$O(n)$
max direct dependents	$O(n)$	$O(1)$	$O(n)$
dependency chain size	$O(1)$	$O(n)$	$O(1)$

Table 1: Properties of the test cases.

name	model	compression	scheduler
Sync	sync	off	off
Async	async	off	off
AsyncComp20	async	on	off
AsyncComp20Sch	async	on	on

Table 2: Different configurations used for evaluation.

6.2 Experimental Results

We now discuss four experiments, focusing on different aspects: *(i)* the dependency structure, *(ii)* the spreadsheet size, *(iii)* the main-memory and external-memory setups, and *(iv)* the compression factor.

Illustrative Experiment 5: Dependency Structure Variation. This experiment demonstrates the extent of the benefit of asynchronous computation and the individual techniques on different dependency structures.

Figure 11 plots the number of unavailable cells along the y-axis with respect to time along the x-axis, one line for each of the configurations in Table 2. We compare the plots between test cases, each with a fixed size n = 300,000, except for RunTotalSlow, with n = 10,000, because of its larger $O(n^2)$ computation workload.

We begin by describing the Rate structure. Here, 300,000 cells out of 600,000 cells depend on cell A1, which is updated. Consider the time 1,500 ms. All 600,000 cells remain unavailable for Sync, while about 200,000 cells and 150,000 cells remain unavailable for Async and AsyncComp20 respectively. This represents a huge reduction in unavailable cells of **67%** and **75%** respectively, relative to Sync, demonstrating the benefits of asynchronous computation. Furthermore, AsyncComp20 performs much better than Async, with dependency identification complete at about 1 ms for the former, and about 750 ms for the latter, highlighting the benefits of the CDT. For the Rate structure, the order of computation scheduling does not matter since all of the formulae require the same amount of computation workload, leading to a linear decrease once dependency identification is complete for all Async variants. The identical computation workload also explains why AsyncComp20Sch takes slightly longer than AsynchComp20: the scheduler does not provide a benefit and introduces a bit of overhead.

For the RunTotalFast structure, once again we find substantial benefits for the Async variants relative to Sync; all the Async variants have their unavailability drop by 50% as soon as dependency identification is complete. Here, unlike Rate, the order of scheduling does matter: if a cell in the middle of the chain is computed first, the computation engine needs to follow the dependency chain to ensure that the cell's precedents are computed in a proper order, during which time no cells are returned to the user. As a result, we do not observe a steady decrease for Async and AsyncComp20. AsyncComp20Sch doesn't do much better—this is because the scheduler assigns the same cost to all formula cells (*i.e.*, the number of direct precedents, a constant). As a result, the scheduler chooses an arbitrary cell for computation at each step. A schedule that utilizes a cell's position in the chain in its cost may be able to return cells more gradually.

Finally, for RunTotalSlow, there are 10,000 cells requiring recomputation—they each have different complexities and do not depend on each other. Here, we see the benefits of scheduling, with AsyncComp20Sch performing better than AsyncComp20. For example, at the 10s mark, AsyncComp20Sch has about half the unavailable cells of AsyncComp20. All configurations finish computation much later than in Rate or RunTotalFast (about 20s) because the computation is much more intensive ($O(n^2)$ as opposed to $O(n)$). For this reason, the drop in unavailable cells from 20,000 cells to 10,000 cells due to dependency identification is barely visible.

> *Takeaway: For asynchronous computation, different dependency structures impact the rate at which cells are computed and returned to the user. CDT ensures interactivity by reducing the response time across all structures. On-the-fly scheduling helps when the dependent cells vary in terms of their workload.*

Illustrative Experiment 6: Dependency Size Variation. This experiment demonstrates the extent of the benefit of

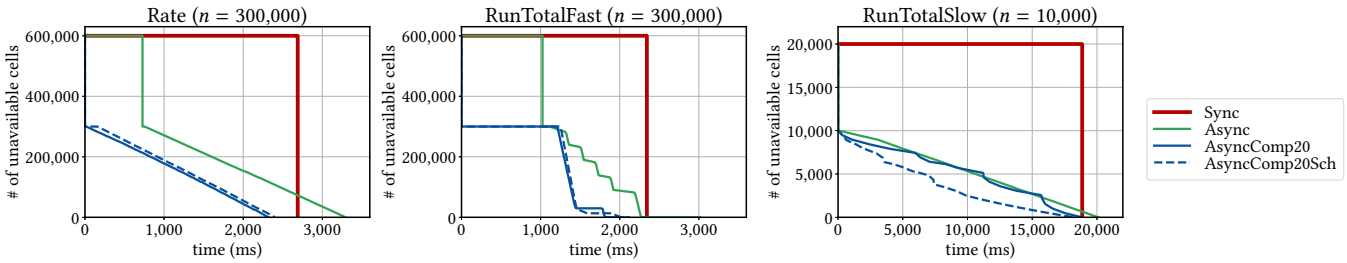

Figure 11: Comparing the plot of the number of unavailable cells over time under different model configurations on multiple test cases. The rate at which the number of unavailable cells decrease depends on the dependency structure.

asynchronous computation and the individual techniques while the sizes of the dependency structures vary. Figure 12 plots unavailability (the area under the curve of unavailable cells over time) as a percentage of Sync, as a function of the test case size n for various configurations. For example, consider Rate with n = 100,000. These four points correspond to the four curves for Rate in Figure 11, comparing unavailability, the area under the curve. While Sync is always at 100%, the area under Async (green) and AsyncComp20 (solid blue) is 51% and 22%, respectively, compared to that of Sync (red). Overall, for Rate, across a range of n, Async reduces unavailability to 51%–68%, and AsyncComp20 further reduces unavailability to 22%–43%. In an ideal situation, dependency identification is instantaneous, and incremental computation occurs at a constant rate, finishing at the same time as Sync. For this situation, unavailability would be 25% of Sync. AsyncComp20 approaches this ideal case as n increases and dependency identification occupies a smaller fraction of the overall time. For a similar reason, AsyncComp20Sch is worse for small n since scheduling introduces overhead; it approaches the performance of AsyncComp20 for large n. (Recall that scheduling doesn't help for Rate.) The behavior for RunTotalFast is similar; however, the percentages are slightly higher, because the long dependency chain leads to the extra work of precedence traversal, as discussed previously. Finally, for RunTotalSlow, Async reduces unavailability to 26%–28% compared to Sync, with or without graph compression. Scheduling reduces unavailability further to 17%–18%. The benefit of graph compression remains, but it is small compared to the much larger computation workload here. Here, scheduling helps because of the skew in the computation cost of the dependent cells. The decreasing trend does not appear in this case, because the overhead as discussed previously is overshadowed by the computation.

Takeaway: The benefit of asynchronous computation measured using unavailability increases with the size of the sheet.

Illustrative Experiment 7: Data Store Configurations. This experiment compares the benefit of asynchrony between the main-memory and the two-tier-memory settings. Figure 13 plots the number of unavailable cells as a function of time, similar to Figure 11. Both graphs here use the Rate test case with size n = 5,000, but the results are similar for other test cases. Comparing the two graphs, the trends of the lines for both systems are almost identical: Async has a large drop off once dependency identification is done and continues to decrease almost linearly, reaching zero unavailable cells slightly after Sync does. The only major difference is in the x axis scale, due to the latency of fetching cells from disk; for instance, the two-tiered system completes in 5,695 ms, whereas main-memory completes in 27 ms for Async.

Takeaway: Both main-memory and two-tiered settings show similar trends for number of unavailable cells over time.

Illustrative Experiment 8: Compression Constant. This experiment demonstrates the impact of the constant K_{comp} for structured dependency graphs. The dependency structures in all three test cases are highly compressible into rectangular regions, in contrast to Illustrative Experiment 2, which varies K_{comp} (Figure 6). Compression, overall, leads to few, if any, false positives, regardless of K_{comp}. To see this, Figure 14(a) plots the unavailability of AsyncCompK_{comp}Sch on RunTotalSlow for values of $K_{\text{comp}} = 2, 20, 200$ as a percentage of Sync along the y-axis with n along the x-axis, similar to the graphs in Figure 12. The dependents of cell A1 are B1, . . . , Bn, a contiguous block in column B. The greedy dependency compression algorithm can combine dependent cells into K_{comp} contiguous blocks without false positives. Figure 14(b) shows an example of how greedy compression may group the cells in column B so that they in total form 2, 20, and 200 regions respectively. As no false positives are added, the value of K_{comp} does not impact performance, as is seen in Figure 14(a).

Takeaway: The effects of K_{comp} are not significantly different on spreadsheets with structured dependency graphs, but can make a large impact on unavailability for spreadsheets with unstructured dependencies (see Figure 6).

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 have been studied in various contexts with different goals. There is also related

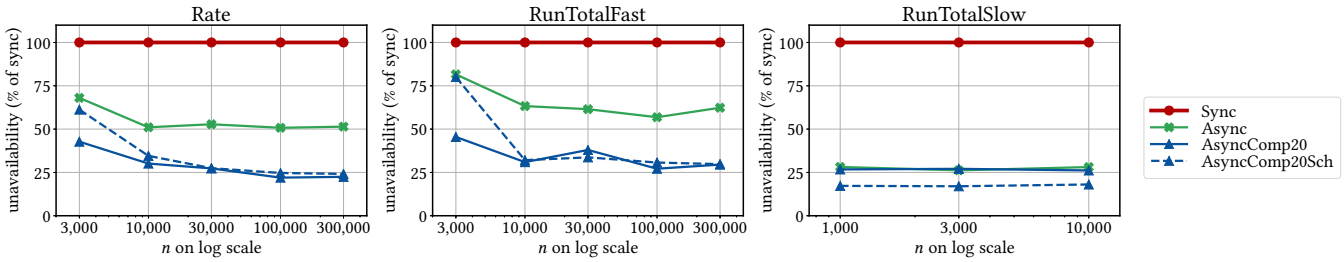

Figure 12: Unavailable cells over time, as a percentage of the Sync configuration, on multiple test cases of various sizes.

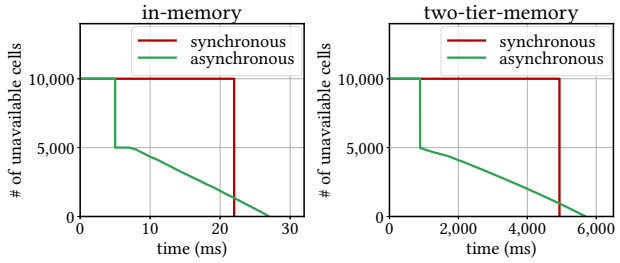

Figure 13: These two graphs plot the number of unavailable cells with respect to time for main-memory and two-tier-memory systems for the Rate test case with n = 5,000. The two graphs display the same trend.

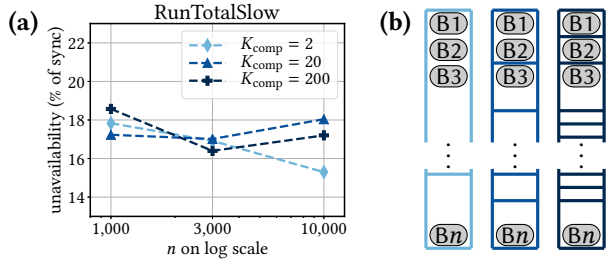

Figure 14: (a) The plot of unavailable cells over time, as a percentage of the Sync configuration, on RunTotalSlow, with various K_{comp} values. (b) Dependencies of cell A1 in RunTotalSlow can be compressed without false positives regardless of the compression constants K_{comp}.

work that handles spreadsheets at scale. We now discuss each of these categories of related work in more detail.

Computation Models. Asynchrony has been used in operations with delayed actors, such as in crowdsourcing [23, 28] and web search [11] but never for spreadsheets. The synchronous model of traditional spreadsheets uses dependency graphs [25, 30, 34] to avoid unnecessary computations, but it does not avoid the performance degradation due to a large and complex network of dependencies [33]. Our proposed asynchronous computation model along with the CDT 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 more related. Graph compression methods, surveyed by Liu et al. [19], have different goals, such as compactness with bounded errors [27], optimizing pattern matching queries [21], and supporting dynamism [32]. Our setting is different because of *(i)* our goal of quickly obtaining a representation of the 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 [18]. Similar scheduling problems arise in this paper, and some hardness results are drawn from previous work. However, as discussed in Section 4, in 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 [2], ABC [29], Airtable [3], DataSpread [8, 9], and Oracle [35, 36]. While these systems address scale, interactivity for formula computation is not their focus. Our work is distinguished from these papers in its ability to provide partial results with consistency guarantees.

8 CONCLUSION

Our proposed asynchronous computation model improves the interactivity of spreadsheets without violating consistency while working with large datasets. To do so, we introduced the notion of partial results, wherein formulae that are being computed in the background are blurred out. We ensured interactivity by proposing the CDT to support the identification of dependent cells after an update in a bounded amount of time. We further 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 substantial improvements in interactivity and usability compared to traditional spreadsheet systems through an extensive set of experiments on spreadsheets, while varying multiple dimensions. *Our new computation model's improved interactivity and usability enables the use of spreadsheet systems in data analysis situations where it was once inconceivable.*

REFERENCES

[1] http://adityagp.net/dataspread-formulae.html.
[2] 1010 Data. https://www.1010data.com/.
[3] Airtable. https://www.airtable.com/.
[4] React:A JavaScript library for building user interfaces. https://reactjs.org.
[5] Spring Framework. https://spring.io/.
[6] ZK Spreadsheet. https://www.zkoss.org/product/zkspreadsheet.
[7] Memory usage in Excel 2013 & 2016. https://support.microsoft.com/en-us/help/3066990/memory-usage-in-the-32-bit-edition-of-excel-2013-and-2016, 2017.
[8] M. Bendre et al. Dataspread: Unifying databases and spreadsheets. In *VLDB*, 2015.
[9] M. Bendre et al. Towards a holistic integration of spreadsheets with databases: A scalable storage engine for presentational data management. In *ICDE*, 2018.
[10] M. Bendre et al. Faster, higher, stronger: Redesigning spreadsheets for scale. In *ICDE*, 2019.
[11] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. *Computer networks*, 30(1-7):107–117, 1998.
[12] R. Cilibrasi and P. Vitanyi. Clustering by compression. *IEEE Transactions on Information Theory*, 51(4):1523–1545, 2005.
[13] R. W. Conway et al. *Theory of scheduling*. Courier Corporation, 2003.
[14] J. Culberson and R. Reckhow. Covering polygons is hard. *Journal of Algorithms*, 17(1):2–44, 1994.
[15] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD*, 1984.
[16] B. Klimt and Y. Yang. Introducing the enron corpus. In *CEAS*, 2004.
[17] A. V. Kononov, B. M. Lin, and K.-T. Fang. Single-machine scheduling with supporting tasks. *Discrete Optimization*, 17:69–79, 2015.
[18] E. L. Lawler. Sequencing jobs to minimize total weighted completion time subject to precedence constraints. In *Annals of Discrete Mathematics*, volume 2, pages 75–90. Elsevier, 1978.
[19] Y. Liu et al. Graph summarization methods and applications: A survey. *ACM Computing Surveys (CSUR)*, 51(3):62, 2018.
[20] Z. Liu and J. Heer. The effects of interactive latency on exploratory visual analysis. *TVCG*, 2014.
[21] A. Maccioni and D. J. Abadi. Scalable pattern matching over compressed graphs via dedensification. In *KDD*, 2016.
[22] K. Mack et al. Characterizing scalability issues in spreadsheet software using online forums. In *SIGCHI*, 2018.
[23] A. Marcus et al. Crowdsourced databases: Query processing with people. In *CIDR*, 2011.
[24] Microsoft UK Enterprise Team. How finance leaders can drive performance. https://enterprise.microsoft.com/en-gb/articles/roles/finance-leader/how-finance-leaders-can-drive-performance/, 2015.
[25] D. Models. Excel's smart recalculation engine, 2014.
[26] B. A. Nardi and J. R. Miller. *The spreadsheet interface: A basis for end user programming*. Hewlett-Packard Laboratories, 1990.
[27] S. Navlakha et al. Graph summarization with bounded error. In *SIGMOD*, 2008.
[28] A. G. Parameswaran et al. Deco: declarative crowdsourcing. In *CIKM*, 2012.
[29] V. Raman et al. Scalable spreadsheets for interactive data analysis. In *DMKD*, 1999.
[30] P. Sestoft. *Spreadsheet Implementation Technology: Basics and Extensions*. The MIT Press, 2014.
[31] B. Shneiderman. Direct manipulation: A step beyond programming languages. *IEEE Computer*, 16(8):57–69, 1983.
[32] N. Tang et al. Graph stream summarization: From big bang to big crunch. In *SIGMOD*, 2016.
[33] C. Williams et al. Speeding up calculations and reducing obstructions. https://docs.microsoft.com/en-us/office/vba/excel/concepts/excel-performance/excel-improving-calcuation-performance.
[34] C. Williams et al. Excel performance: Improving calculation performance. https://docs.microsoft.com/en-us/office/vba/excel/concepts/excel-performance/excel-improving-calcuation-performance, 2017.
[35] A. Witkowski et al. Advanced SQL modeling in RDBMS. *TODS*, 30(1):83–121, 2005.
[36] A. Witkowski et al. Query by excel. In *VLDB*, 2005.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable feedback. We acknowledge support from grants IIS-1513407, IIS-1633755, IIS-1652750, and IIS-1733878 awarded by the National Science Foundation, grant W911NF-18-1-0335 awarded by the Army, and funds from Adobe, Toyota Research Institute, and Google.

APPENDIX

The appendix includes the following: *(i)* the proofs for the NP-Hardness claims, *(ii)* techniques for adding and deleting entries from the CDT, *(iii)* the two-tier memory model used in DataSpread, and *(iv)* the architecture of DataSpread.

A PROOFS

This section provides proofs for NP-Hardness claims.

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

Problem 5 (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. (Sketch) Given a simple and holeless orthogonal polygon P and an integer k in a Polygon Exact Cover instance, we can perform a rank-space reduction on the coordinates of P; that is, we 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. We then 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. □

Proof of Theorem 2. Problem 7 is a generalization of the Scheduling With Supporting Tasks problem [17], which is NP-Hard, defined as follows.

Problem 6 (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, such 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. (Sketch) We provide a polynomial-time reduction from Scheduling With Supporting Tasks. For each task t in $A \cup B$, we create a corresponding cell $\text{cell}(t)$. For each $(a_i, b_j) \in R$, we 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 $\{a \mid (a, b_j) \in R\}$, we create a formula $\text{cell}(b_j) = \text{cell}(a_{i_1}) + \cdots + \text{cell}(a_{i_\ell})$. We then 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 MAINTENANCE OF THE CDT

In this section, we introduce techniques for inserting into and deleting from elements from the CDT.

Deleting dependencies. Deleting a dependent from the CDT can potentially introduce false negatives. Consider the example 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 → 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, *i.e.*, the dependency between C4 and B3 is due to more than one formula. Another issue is deleting a single dependent cell from a range, which is difficult to do efficiently without leading to a highly fragmented, inefficient R-tree.

A simple way to circumvent deletion issues is to make no changes to the CDT on 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 CDT 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.

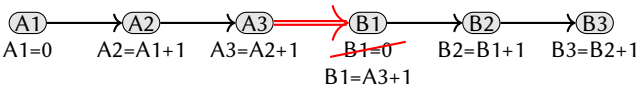

Figure 15: Adding a new direct dependency.

Adding dependencies. Adding a direct dependency can be quite time-consuming in the worst case. Consider the example in Figure 15, 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 CDT by adding B1, B2, B3 as their dependents, which is quite time consuming.

To get around this issue, we introduce *lazy dependency propagation*. The idea is to only add the direct dependency (A3 → B1) to the CDT. 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 CDT 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 CDT in the same manner. We propagate the must-expand dependencies in the background during idle time. Specifically, for a must-expand dependency $u \to v$, 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 CDT leaves the unstable state once all must-expand dependencies are propagated.

Algorithm 3 provides an outline of the process to lazily propagate dependencies of cells that have a must expand flag. We denote the resulting set of cells from looking a cell c up in the CDT as $\text{CDT}(c)$.

Input: a cell c marked with a must-expand flag
$Q \leftarrow \{c\}$; $S \leftarrow \text{CDT}(c)$;
while $|Q| > 0$ **do**
  $c' \leftarrow$ element of Q; $Q \leftarrow Q - \{c'\}$;
  for *each* $d \in \text{CDT}(c')$ *with must-expand flag* **do**
    $Q \leftarrow Q \cup \{d\}$; $S \leftarrow S \cup \text{CDT}(d)$;
  end
end
for *each key* $k \in$ CDT **do**
  if $\text{CDT}(k) \cap \{c\} \neq \emptyset$ **then**
    $\text{CDT}(k) \leftarrow \text{CDT}(k) \cup S$;
  end
end

Algorithm 3: Lazy Dependency Propagation algorithm

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

For example, instead of updating all entries as in Figure 16(a), B1 is added as a must-expand dependent of A3, as in Figure 16(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

(a)

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

(b)

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

Figure 16: Adding dependencies to CDT: (a) naïve method (b) lazy dependency propagation (must-expand dependencies are marked by asterisks)

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 CDT becomes similar to that shown in Figure 16(a).

A 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 operations while we update the CDT, 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 CDT 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.

C HANDLING SCALE

Unfortunately, current main-memory-based spreadsheet systems do not scale to large datasets [7]. Here, we adapt the techniques described earlier to a *two-tier memory model* that does not have similar scalability limitations. Unlike main-memory-based systems, where computation time is the dominant concern, here, data retrieval from storage is also important. Taking these fetching costs into account can lead to even more improvements than what is gained by the original techniques alone.

The Two-Tiered Memory Model. The *two-tier memory model* contains two tiers of memory: (a) the *main-memory*, which is limited in size, but allows fast data access—the application interfaces with this tier; and (b) the persistent *storage*, which is large, but data access is slow—and the application does not directly interact with this tier.

Under this model, 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 the spreadsheet application but rather via the main-memory in a read/write-through manner, meaning *(i)* if the application requires 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 main-memory, and the control is returned to the application only when the update is also reflected in storage. In particular, for DataSpread, we use a relational database for the storage tier.

Unfortunately, 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. Next, we explore how graph compression and scheduling changes when the main cost is the fetching cost.

Fast Dependency Identification. The dependency graph can be as large as 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 two-tier systems; 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 CDT, as presented for main-memory systems, can also be used in the two-tier setting. The CDT can be stored as a relation in the storage layer. A query for dependents of a cell u is often a straightforward lookup in this 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 7 (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.*

Since 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 the dirty cells Δ. We can show that the Computation Scheduling with Fetching Costs problem is NP-Hard.

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

Scheduling for two-tiers 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

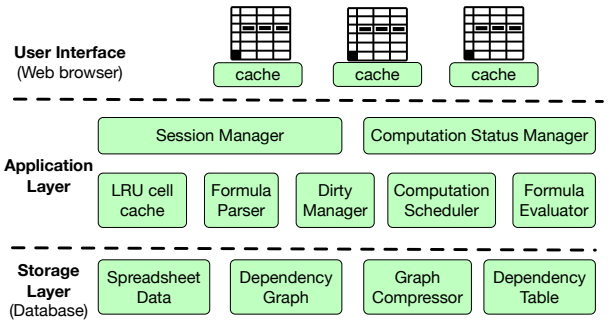

Figure 17: DataSpread **Architecture**

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 the limited-size cache, requiring re-fetching. This can be factored into the cost function. It may require dynamic updates as the cache changes in the same way weights are updated when the viewport moves.

D ARCHITECTURE

Here, we describe the architecture of DataSpread, depicted in Figure 17. We explain how the components that utilize the techniques presented in the paper work together to obtain an interactive and consistent spreadsheet system.

At a high level, DataSpread's architecture 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 [5] 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.

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.

The session manager communicates with the *dirty manager* to determine which cells are shown to the user, and make appropriate 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—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. On a cell update, fast dependency identification marks 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 which 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.

Finally, in addition to utilizing the techniques described in Sections 2, 3, and 4, and described above, DataSpread supports a number of additional features to ensure interactivity and scalability. *(i)* We implement caching at multiple layers using an LRU-based evacuation scheme. *(ii)* 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. *(iii)* 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. *(iv)* We adopt the idea of a virtual DOM (Document Object Model), where the web browser only renders the visible cells.