TOWARDS UNIFYING SPREADSHEETS WITH DATABASES FOR AD-HOC INTERACTIVE DATA MANAGEMENT AT SCALE

BY

MANGESH BENDRE

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2018

Urbana, Illinois

ABSTRACT

We are witnessing the increasing availability of data across a spectrum of domains, necessitating the interactive ad-hoc management and analysis of this data, in order to put it to use. Unfortunately, interactive ad-hoc management of very large datasets presents a host of challenges, ranging from performance to interface usability. This thesis introduces a new research direction of manipulation of large datasets using an interactive interface and makes several steps towards this direction. In particular, we develop DataSpread, a tool that enables users to work with arbitrary large datasets via a direct manipulation interface. DataSpread *holistically unifies spreadsheets and relational databases* to leverage the benefits of both. However, this holistic integration is not trivial due to the differences in the architecture and ideologies of the two paradigms: spreadsheets and databases. We have built a prototype of DataSpread, which, in addition to motivating the underlying challenges, demonstrates the feasibility and usefulness of this holistic integration. We focus on the following challenges encountered while developing DataSpread. *(i) Representation*—here, we address the challenges of flexibly representing ad-hoc spreadsheet data within a relational database; *(ii) Indexing*—here, we develop indexing data structures for supporting and maintaining access by position; *(iii) Formula Computation*—here, we introduce an asynchronous formula computation framework that addresses the challenge of ensuring consistency and interactivity at the same time; and *(iv) Organization*—here, we develop a framework to best organize data based on a workload, *e.g.*, queries specified on the spreadsheet interface.

To my parents, wife, and children, for their love and support.

ACKNOWLEDGMENTS

This project would not have been possible without the support and generous help from all the people around me.

First, I would like to express my sincere gratitude to my advisors Professor Kevin Chang and Professor Aditya Parameswaran for their continuous support of my Ph.D. study and related research, for their patience, motivation, and immense knowledge. I am thankful to them for not only the insightful discussions but also the rigorous coaching I needed to dig deeper and think bolder. Next, I would also like to thank the rest of my committee members Professor ChengXiang Zhai and Professor Arnab Nandi for their insightful comments, which helped me to widen my research from various perspectives. Finally, I want to thank Professor Karrie Karahalios for enabling me to look at my research from a very different perspective.

I would also like to thank all the collaborators of DataSpread: Xinyan Zhou, Ding Zhang, Vipul Venkataraman, Sajjadur Rahman, Tana Wattanawaroon, Kelly Mack, Pingjing Yang, Himel Dev, Yu Lu, Yuyang Liu, Shichu Zhu, Shan Bai, Joon Park, Ti-Chung Cheng, Shike Zhang, and Bofan Sun—this project would not be a reality without you all!

Additionally, I am also thankful for my lab mates for all the stimulating discussions that I had over the years. I'd like to thank Long Pham, Amin Javari, Aravind Sankar, Tarique Siddiqui, Mun-Thye Mak, Mianwei Zhou, Rui Li, Yuan Fang, among many others.

Last, but not least, I would like to thank my wife, Ketaki, and my children, Anvi and Anay, for the encouragement and emotional support I needed to complete this challenging process.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

We are witnessing increasing availability of data across a spectrum of domains, necessitating interactive and ad-hoc management of this data: a business owner may want to manage customer data and invoices, a scientist experimental measurements, a professor student grades, and a fitness enthusiast heart rate and activity traces. However, while there are two dominant software paradigms for providing a direct manipulation interface and enabling effective management of big data—spreadsheets and databases respectively—neither of them fulfill the desired requirements, as we illustrate below.

Paradigm 1.1: Spreadsheets. Spreadsheets, from the pioneering VisiCalc [1] to Microsoft Excel [2] and Google Sheets [3], have found ubiquitous use in ad-hoc manipulation, management, and analysis of tabular data. Recent estimates from Microsoft posit that there are over 750 million users of spreadsheets, specifically Microsoft Excel [4]. They provide a common artifact and language for data to the billions who use them, enabling collaboration; they provide statistical analysis functionality; and they provide visualization tools to make sense of data. Above all, they provide this functionality in a *direct manipulation* interface [5, 6] that caters to both novice as well as advanced users, spanning businesses, schools, organizations, and home. Spreadsheets have, in fact, been heralded as the pioneering example of a direct manipulation interface.

This mass adoption of spreadsheets breeds new challenges. With the increasing number of people using spreadsheets, size of data sets, types of analyses, and extent of collaboration, we see a frenzy to push the limits: *users are struggling to handle larger and more complex data*; they are trying to import large data sets into Excel (*e.g.*, billions of gene-gene interactions), compose complex operations not naturally supported (*e.g.*, "joins" between multiple tables), handle errors and inconsistencies resulting from manual entry or their own makeshift strategies for collaboration (*e.g.*, by emailing back and forth), with no data validation and error recovery mechanisms [7, 8]. In response, *spreadsheets are stretching the size of data and functionality they can support.* For example, Excel has lifted its size limits from 65k to 1 million rows and added Power Query and Power Pivot [9, 10] to import data from databases in 2010; Google Sheets has expanded its size limit to 2 million *cells* and touted its support of SQL, although restricted to one table. Despite these moves, spreadsheets are far from the kind of scale (*e.g.*, beyond memory limits) and functionality (*e.g.*, relational operations, transactional semantics) that databases natively provide. We describe a concrete example of the limitations of spreadsheets from our collaborators below.

Example 1.1 (Using Spreadsheets for Genomic Data Analysis)**.** *During the course of genomic data analysis, biologists, such as our collaborators at the KnowEnG center at Mayo Clinic, generate data describing genomic variants as VCF (variant cell format) files, akin to CSV files. These VCF files are large, with tens of millions of rows and hundreds of columns, plus a raw size of many gigabytes. Unfortunately, many biologists, like scientists in many other domains, are adept at using spreadsheet software, but are not comfortable enough with programming to use databases. To interactively explore or browse their VCF data, they struggle to load such files into spreadsheet software. For example, Microsoft Excel limits datasets to 1M rows and Google Sheets to 2M cells. And even when one can load large datasets, these tools become sluggish and unresponsive. In fact, many biologists are unable to explore the datasets they create, instead sending them to bioinformatics collaborators to analyze.*

Similar to this example, a recent study of scalability issues in spreadsheet software via analysis of Reddit posts [11] revealed 83 separate accounts of issues with Excel that were, at least in part, due to the scale of data or operations on the spreadsheet. Anecdotal evidence also indicates that users struggle with a variety of scalability problems on large and/or complex spreadsheets. Many of the Reddit posts indicate scalability issues arising from as few as tens of thousands of rows, well below the size limits of the spreadsheet software [12].

Thus, to summarize, even though recent attempts have spreadsheets increasingly supporting database-like functionality, at their core, spreadsheets are facing inherent limitations due to their lack of support for non-main-memory resident data, relational operations, and transactional semantics, among others.

Paradigm 1.2: Relational Databases. Relational databases, such as Oracle, Microsoft SQL Server, IBM DB2, MySQL, and PostgreSQL, being scalable, expressive, and transactional, are the de-facto standard for large-scale data storage, processing, and management, within most organizations. The decades of research and development since Codd's relational model [13] has validated *relational algebra* as a solid foundation for computation on tables, with SQL as its realization, *external-memory algorithms and indexing* for scalable query processing, and *transaction processing* for maintaining data consistency. However, databases have their limitations, as the following example from our collaborators indicate.

Example 1.2 (Using Databases for Customer Management). *The owner of a small retail startup in Champaign, Illinois created a MySQL database for managing customers and sales, organized in a schema comprising 15 tables. There are several actions that he and his staff would like to routinely perform, such as insert (customers), modify (due dates of invoices), filter (overdue invoices), join (invoices and payments), and aggregate (the total amounts). To perform these operations without requiring SQL, he has to employ a programmer to develop database applications. Instead, he wants to manipulate data for ad-hoc operations interactively, but no such tools exist.*

As the example demonstrates, this ubiquity and impact of databases comes at a price—with SQL as its "interface", databases are not naturally interactive [14, 15] and do not support direct manipulation of tables, *e.g.*, selecting or modifying a range of data on-demand, placing it in a location, ordering and accessing rows by position, adding "derived" columns and computation in-situ via formulae, and propagating changes of formula-linked data throughout the visible windows. Consequently, users access databases either via pre-programmed database applications (Figure 1.1a), or SQL clients (Figure 1.1b), which only support operations on entire relations at a time, as opposed to directly interacting with data for updates and analysis. Above all, such access is inflexible and does not cater to ad-hoc direct manipulation needs, thus *locking* data *behind* fixed applications. To this end, there have been a number of papers on making databases usable, *e.g.*, [14, 16, 17, 18, 19], but this research has not witnessed widespread adoption.

Summary of Issues with Paradigms. Overall, we are critically lacking a solution for interactive ad-hoc management of data. On the one hand, spreadsheet software, while being heralded as a prime example of a direct manipulation [5] tool, lacks *scalability*, due to its inability to operate on datasets that go beyond main memory capabilities, and *expressiveness*,

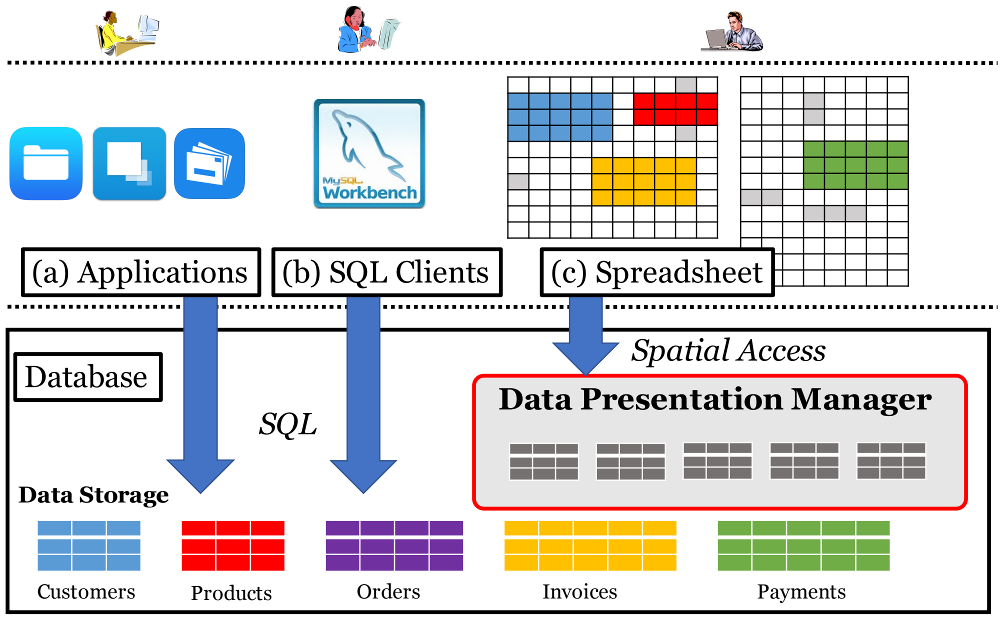


Figure 1.1: Connecting spreadsheets and databases for enabling interactive ad-hoc data management.

since its formulae only operate on one cell at a time, necessitating complicated means (*e.g.*, VLOOKUP) to orchestrate simple operations like joins. On the other hand, while databases provide both scalability and expressiveness, they lack support for direct manipulation vital for interactive ad-hoc data management.

Vision. This thesis aims at addressing this challenge, by advocating a new research direction of supporting *a direct manipulation system for ad-hoc management of big data* and taking the first few steps towards this research direction. As a concrete goal of we are developing DataSpread, a one-stop tool for interactive ad-hoc data management that enables users to collaboratively work with arbitrarily large datasets via a direct manipulation interface. Thus, DataSpread should *(i)* provide a continuous representation of data on an interactive interface, *(ii)* provide physical actions (movement and selection by mouse, touch screen, *etc.*) or labeled button presses instead of complex syntax, *(iii)* provide rapid, incremental, reversible operations whose impact on the object of interest is immediately visible, *(iv)* provide a layered or spiral approach to learning that permits usage with minimal knowledge, and *(v)* effectively work with arbitrarily large datasets that go beyond main-memory sizes.

To build DataSpread, rather than starting from scratch, we propose *holistically inte-*

grating spreadsheets and relational databases, the two dominant paradigms that provide a direct manipulation interface and enable effective management of big data, respectively, to leverage the benefits of both. We have two objectives in building DataSpread: *(i)* (**database objective**) manipulating data within databases on a spreadsheet interface, without relying on pre-programmed applications or SQL clients—thereby enabling interactive ad-hoc data management for a database, while *(ii)* (**spreadsheet objective**) operating on datasets not limited by main memory—thereby addressing a key limitation of present-day spreadsheets. While our ultimate goal is to develop a spreadsheet-like interactive front-end interfaces for all kinds of data stores, including row stores, column stores, and key-value stores, in this thesis, we focus on relational row stores because they are mainstream and universally popular.

Using a system like DataSpread, users can view and manipulate data in a spatially organized tabular interface (Figure 1.1c), in addition to standard approaches (Figure 1.1a,b)—here, the spreadsheet modality equips a database with ad-hoc querying and manipulation capabilities while presenting data for browsing. They can work with large tables (*e.g.*, VCF files) presented on the interface and stored in the database. They can operate at various granularities, embodying the principles of *direct manipulation* [5]—from cells (like a spreadsheet) to tables (like a database)—and add computation in the form of formulae or queries on the interface, alongside data. They can flexibly arrange data, from structured tables, reports, and forms, to ad-hoc layouts combining data and queries. They can refer to data by tables or attributes (as in a database) or position (as in a spreadsheet). While we primarily focus on spreadsheets, the same capabilities can enable other spatial interfaces for interactive ad-hoc data management.

Challenges. Developing DataSpread presents a host of engineering and research challenges, ranging from storage and indexing to interface usability. For example, supporting interactive operations for large datasets necessitates not only an efficient storage mechanism that is not bogged down by the size of data but also an indexing and computational framework that enables updates to the data interactively. To go beyond main-memory limitations, we need to ensure that DataSpread works with a two-tiered storage model, where the upper tier, *i.e.*, main-memory, is limited in size and acts as a cache, and the lower tier, *i.e.*, disk, acts as a persistent data store. In addition, representing and supporting operations for a large dataset on an interactive interface in an interpretable manner is challenging. Additionally, spreadsheets and relational databases adopt very different architectures and ideologies, which makes the holistic integration challenging. In particular, we need to deal with the following challenges:

- *Schema:* databases have a strict schema-first data model based on tables and tuples, while the spreadsheet data model is based on sheets with rows and columns, and has no explicitly defined schema.
- *Addressing:* spreadsheets treat rows and columns as equivalent, while databases operate on sets of tuples.
- *Window:* spreadsheets have the notion of the current window, *i.e.*, is the portion of the spreadsheet that a user is currently looking at; there is no such notion in databases.
- *Modifications:* spreadsheets support updates at any level and granularity: rows or columns, while databases only support modifications that correspond to SQL queries.
- *Updates:* spreadsheets do not support automatic updates to underlying data, while databases support automated SQL commands (generated from, say, a program).
- *Computation:* spreadsheets support value-at-a-time formulae to support computation of derived data, while databases support arbitrary SQL queries operating on groups of tuples at once.
- *Interface:* the interfaces and semantics of spreadsheets are not designed to work with large datasets. For example, scrolling through billion row spreadsheets is impossible.

While addressing all of the aforementioned challenges is non-trivial and goes beyond the scope of a single thesis, our goal in this thesis is to demonstrate the feasibility of the overall unification vision and make several useful steps towards the holistic unification. In particular, in Chapter 2, we describe prototypes we have developed that make two important contributions: *(i)* enabling spreadsheets to act as a relational database front-end, and *(ii)* supporting the manipulation of large datasets in a spreadsheet interface. In Chapters 3–5, we address key challenges encountered along the way, namely *(i)* representation, *(ii)* indexing, *(iii)* formula computation, and *(iv)* organization. Then, in Chapter 6, we outline what is lacking in our prototypes and outline a vision for the future to truly address all of the aforementioned challenges and others—a vision we term *directed data management*. We now describe each chapter in more detail:

Chapter 2: System Development Overview and Motivation. Here, we describe our first prototype of DataSpread, using Microsoft Excel as the front-end and PostgreSQL as the back-end. We identify the key challenges of this method of holistic integration, focusing on the limitations of relational databases in supporting a spreadsheet-like direct manipulation interface, and propose a high-level architecture for DataSpread. We describe the following novel features of DataSpread that essentially enables spreadsheets to act as a relational database frontend: *(i)* querying a relational database via a spreadsheet interface using SQL, *(ii)* importing of data from a relational database to the spreadsheet interface and exporting of data from the spreadsheet interface to a relational database, and *(iii)* linking a region on the spreadsheet interface to a table in the underlying relational database, thereby enabling a two-way synchronization between the region on interface and the back-end table.

We build on lessons learned from our first prototype to develop our second one. This prototype introduces features enhancing the user experience, beyond our previous prototype and what traditional spreadsheets provide, with a goal to enable users to work with large spreadsheets efficiently. Here, instead of using Microsoft Excel as a front-end, we used ZK (open-source) spreadsheet frontend [20]. We specifically describe: *(i)* an *asynchronous and lazy computational model* to address the issue of poor interactivity on large and complex spreadsheets; *(ii)* a *navigation interface* to enable users to drill-down to desired areas while examining a summarized view of the data to improve navigability; and *(iii) support for table-oriented formulae*, a simple but effective means to express relational operations on tabular regions to improve expressiveness.

Chapter 3: Spreadsheet Storage Modeling. To enable DataSpread to efficiently work with datasets that go beyond main-memory sizes, it fetches data *on-demand* from the underlying database when triggered by a user action (like scrolling) or from a system action (like calculating a formula). This requires DataSpread's storage engine to address two challenges.

The first challenge is how to flexibly represent spreadsheets in a relational database. A user may manage several table-like regions within a spreadsheet, interspersed with empty rows or columns, along with formulae. Storing such a spreadsheet, a mix of sparse and dense regions, as a single relation is not only wasteful in terms of storage space but also detrimental for access performance (*e.g.*, during scrolling or formula computation). Thus, we adopt a hybrid scheme that takes storage and access patterns into account and stores dense regions as tables, with tuples as spreadsheet rows, and attributes as spreadsheet columns; and sparse regions as tables with tuples as key-value pairs. Unfortunately, it is NP-Hard to identify the optimal hybrid representation scheme.

To address this limitation, we develop an efficient approach to identify the optimal representation from an important and intuitive subclass of representations. In particular, we focus on hybrid data models that can be obtained by recursive decomposition. For this subclass, we can obtain the optimal representation in PTIME by using a dynamic programming-based algorithm. We extend this algorithm to make it even more efficient, at a small cost to optimality. We empirically show that our solutions are close to optimal and can be obtained efficiently.

Our second challenge is in supporting and maintaining spatial access. Storing positional information, *i.e.*, row and column numbers, as-is can lead to expensive cascading updates (as we will describe in Chapter 3) during insert/delete operations. Moreover, we need "positional" indexes that allow range based accesses. Cascading updates make it hard to maintain a traditional index for recording position, *e.g.*, B+ Tree, across edit operations. We introduce positional access mechanisms along with corresponding data structures that don't suffer from the issue of cascading updates, leading to almost sub-linear access and modification performance—this is crucial to ensure interactivity while working with large spreadsheets.

Chapter 4: Asynchronous Formula Computation. Traditional spreadsheets, *e.g.*, Microsoft Excel, adopt a *synchronous computation model*, where users are kept waiting until formula computation is complete—this disrupts interactivity for large and computationally heavy spreadsheets. To address this, DataSpread adopts an *asynchronous computational model* instead, returning control back to the user immediately, while masking the cells whose values have not been computed yet. To quantitatively evaluate the different computation model, we introduce a novel metric of *unavailability*, which we define as the area under the curve, for a computation model, plots the number of cells unavailable to users to act upon.

The primary challenge for enabling asynchronous computation is to maintain both consistency and interactivity at the same time. This requires addressing two problems both of which are NP-Hard: *(i)* identifying the impacted cells after an update on a spreadsheet in a bounded period of time and *(ii)* determining a schedule for computing the impacted cells. To ensure interactivity, we compress the formula dependency graph lossily and propose techniques for compression and maintenance. For scheduling computation of the impacted cells, we propose an on-the-fly scheduling technique. We experimentally show that out techniques ensure interactivity, *i.e.*, returning control to the user in about 100 ms, and perform 12x better over the synchronous computation model as evaluated by our unavailability metric on a real world spreadsheet.

Chapter 5: Relational Schema Design. In Chapter 3, we described how we design a storage manager for ad-hoc spreadsheet data—this can result in many tables stored in the backend database. In this chapter, we complement the discussion by focusing on optimizing the schema of these relational tables. We develop a framework, along with the relevant theory and algorithms, for "quantitatively" designing a schema, *i.e.*, designing a schema that minimizes the execution cost for a workload (*e.g.*, access patterns from formulae).

Quantitatively designing a relational schema is essential not only in the context of DataSpread for efficiently representing backend tabular data, but is also in generic database settings. Among all of the factors that determine the performance of a relational database system, the schema is not just highly important, but also often overlooked. Also, the trend of using databases as an invisible "back-end" demands high performance from a schema as opposed to the normalization or intuitiveness, as is traditionally emphasized. In DataSpread, in particular, we additionally enable users to persist tabular data on the spreadsheet interface as relational tables within the underlying database. Also, on a spreadsheet interface, users can embed queries in the form of formulae along with tabular and non-tabular data. To ensure the interactivity of the interface, efficient execution of these queries is necessary.

Furthermore, the spreadsheet interface acts as an abstraction, isolating users from the underlying database. Thus, it is possible to optimize the schema of the relational tables stored in the underlying database in a transparent manner while still being able to service the queries.

We formulate the goal of quantitatively designing a relational schema as an optimization problem, where we select a schema from a set of possible candidates, *i.e.*, a search space of candidates, that minimizes the execution cost for a workload. We need to address two main challenges: *(i) Search Space:* Defining our search space of candidates requires new semantics to describe the requirements of schema faithfully. Traditionally, dependencies, both functional and multivalued, were introduced to capture database constraints using which appropriate schema can be designed [21, 22, 23, 24, 25]. Unfortunately, they fall short for describing many-to-many relationships. We address this limitation by introducing a novel concept of *attribute associations* to describe schema requirements, thereby enabling us to define our search space. Unfortunately, our search space is exponential with respect to the number of associations. *(ii) Optimization:* To find the optimal schema, *i.e.*, one that minimizes the cost of a workload, within our exponential search space, we develop an *anti-monotonic property* to efficiently prune the schema search space. The anti-monotonic property makes very few assumptions about the underlying database, and thus we can use it across a variety of relational databases. We experimentally demonstrate up to 2x speedups on three datasets, a significant improvement over the current state-of-the-art schema optimization methods.

Chapter 6: Future Work: Directed Data Management. In previous chapters, we introduced the idea of holistically integrating spreadsheets and databases to enable direct manipulation at scale. However, due to the scaling limitations of the spreadsheet interface, this version of holistic integration does not completely address our goal of interactive ad-hoc data management at scale. So to fully address this goal, we propose a new research direction, termed *directed data management*, to effectively bring the usability benefits of spreadsheets to databases, while not sacrificing the power and scalability of databases. We argue that on extremely large datasets the vanilla direct manipulation capabilities offered by spreadsheets are no longer effective, necessitating extensions to support multiple perspectives, accelerated actions, and progressive feedback. In this chapter, we describe the challenges underlying directed data management.

CHAPTER 2: OUR VISION AND SYSTEM DEVELOPMENT

As we argued in the introduction, ad-hoc interactions with databases are challenging. This is due to two reasons: *operations* and *representation*. First, even if it is easier to issue correct SQL queries using recent work such as gestures [26, 18], natural language [17], and auto-completion [19, 27], SQL is still an indirect means for *operating* on data, requiring users to issue declarative queries in "batch mode" on relations at a time, waiting until the entire query is crafted and issued before seeing any results. Second, databases do not persist the state of the analysis, nor can users organize and *represent* their analysis results in a way that is easy to understand and share.

While the usability of databases for ad-hoc interactions is questionable, spreadsheet software is incredibly popular for such tasks. Spreadsheets provide *direct manipulation capabilities*, as defined by Shneiderman [5], allowing users to interact with a continuous *representation* of the object of interest (data in a collection of cells) via incremental, reversible *operations* (updates, filters) by performing physical actions (clicks, scrolling)—addressing the same two aspects that were problematic for databases. Shneiderman attributes the success and usability of spreadsheets to its direct manipulation capabilities. Nardi and Miller [28] attribute the success of spreadsheets to these same two aspects that were problematic for databases, *operations*—actions that match user tasks and shield them from cumbersome programming—and *representation*—a table-oriented interface that allows users to spatially organize their data along with the analysis.[1]

Our first goal is to *bring the usability benefits of a spreadsheet-like interface—the direct manipulation capabilities and ease-of-use—to relational databases*, thereby providing a solution to database usability for ad-hoc interactions. Unfortunately, we find that adding spreadsheet-based direct manipulation capabilities to databases is not straightforward, in part due to fundamental incompatibilities between direct manipulation and the declarative querying paradigm, as we discuss below.

From Declarative Queries to Direct Manipulation. To make databases more usable,

[1]Nardi and Miller referred to these notions as *computation* and *presentation* respectively, but we use the terminology from Shneiderman [5] for uniformity.

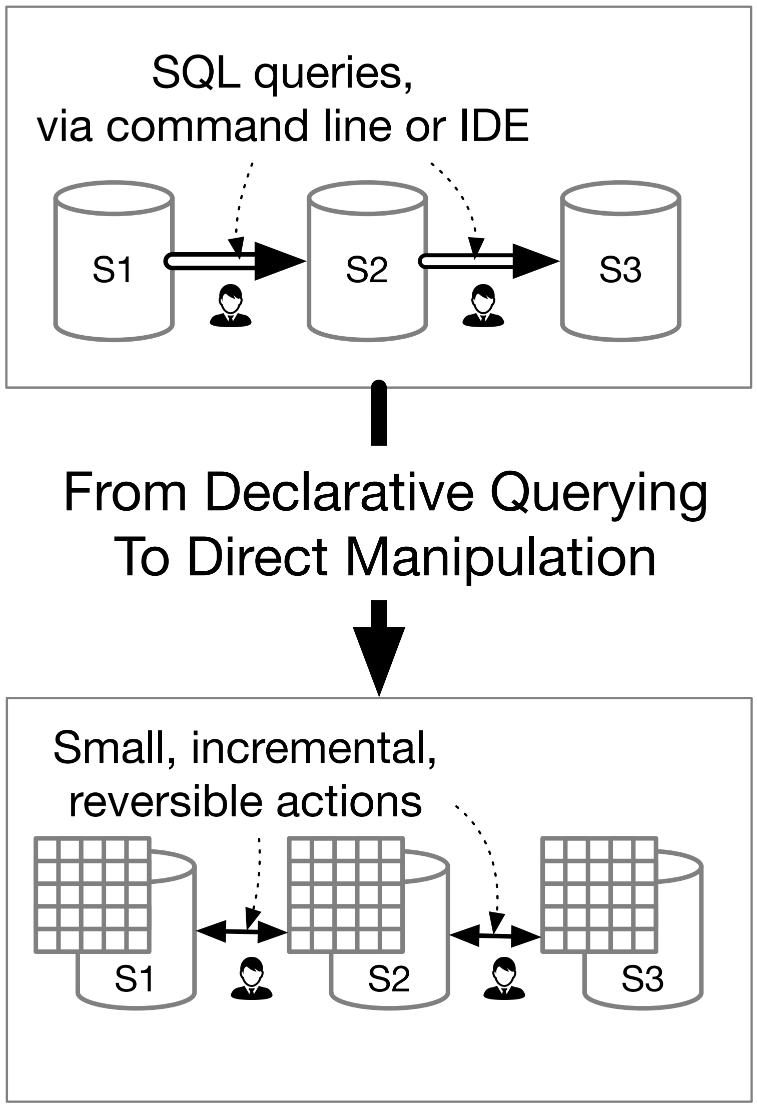


Figure 2.1: From Declarative Queries to Direct Manipulation.

we need to equip databases with direct manipulation capabilities via a spreadsheet-like front-end, while applications requiring SQL access to the database continue to have such unfettered access to the underlying data, as shown in Figure 2.1. Unlike traditional data management, where there are irreversible transitions between database states (depicted as drums), based on coarse-grained SQL queries on entire relations (depicted as unidirectional thick arrows), our goal is to support a spreadsheet representation (depicted as a grid) as well as the database state at each juncture, and supports incremental, reversible, and fine-grained operations (depicted as bidirectional thin arrows) to change the representation along with data. However, databases *do not natively support an ad-hoc spatial representation, with* *operations that impact one or more cells at a time, referred to by their position on the sheet.* From a representation standpoint, data on spreadsheets is organized in an ad-hoc, task-dependent manner, with tables interspersed with whitespace and formulae; databases instead organize data in uniform unordered relations. Moreover, the organization heavily relies on position—but maintaining position within a database is hard: adding or deleting a row can lead to cascading updates to row numbers of subsequent rows, necessitating indexing structures that can maintain positional information efficiently. From an operation standpoint, direct manipulation encourages the embedding of formulae along with data, leading to new optimization challenges: we need to provide databases with the ability to execute a complex network of formulae, prioritizing for what the user is seeing. Overall, we need to equip databases with what we're calling *representation*-awareness and *operation*-awareness. Moreover, databases are not optimized for human-centered objectives that go beyond the typical latency and throughput metrics, such as ease-of-use, response time, lack of frustration, and lack of errors. We describe our vision along with the challenges in detail in Section 2.1.

We are building DataSpread as a concrete attempt towards realizing the goal of equipping databases with direct manipulation capabilities via a spreadsheet-like front-end. We envision that DataSpread will serve as a one-stop tool for interactive ad-hoc data management that enables users to work with arbitrarily large datasets via a spreadsheet-like direct manipulation interface. In the rest of the chapter, we discuss our development of various prototypes of DataSpread. Our first prototype, as discussed in Section 2.2, serves as a proof-of-concept demonstrating the feasibility and usefulness of holistically unifying spreadsheets with relational databases. Our second prototype, as discussed in Section 2.3, focuses on features that enhance the user experience beyond what traditional spreadsheets provide with a goal to enable the users to work with large spreadsheets efficiently.

2.1 OUR VISION: DIRECT MANIPULATION AT SCALE

As we argued earlier, direct manipulation on a spreadsheet-like interface hinges on human-interpretable *representations* of the state of the data and intuitive *operations* that allow transitions between states: users *"view a concrete, visible representation of data as a tabular layout of cells, upon which to perform incremental operations"* [28]. As it turns out, adding support for direct manipulation to a database requires every layer of the database stack to be more aware of human-centered needs.

On the one hand, to allow users to interact with a "concrete, visible representation of data in a tabular layout of cells", this requires the data to be *ordered* in either dimension, and *positioned* on a grid. Moreover, this representation is a "model of the problem of interest" [29]. Such a task-specific representation is necessarily *irregular*, with multiple modularized regions of data, interspersed with whitespace, all on a single sheet. Unfortunately, these notions of ordered, positional, and irregular data representations are in sharp contrast to the relational model in a database, which is set-oriented, not organized in a grid, regular (*i.e.*, each tuple has the same set of attributes), and often normalized and thus disallows redundancy (no derived values). Thus, in order to support direct manipulation, databases must be equipped with *representation-awareness*—a new notion.

On the other hand, to support rapid, incremental, and reversible operations, we should provide *small operations* with *guaranteed responsiveness* (*e.g.*, sub-second) to cater to the fact that users are impatient. Once again, we find that these notions are very different from the declarative querying paradigm in a database, where a query is a "batch" command, with non-guaranteed response time. To support direct manipulation, databases must also be equipped with *operation-awareness*.

2.1.1 Supporting Representation Awareness

We now consider some of the key challenges in bringing representation-awareness to databases.

Irregularity. Direct manipulation of data requires the flexibility to develop a concrete and visual representation of data, for a specific activity. For example, a realtor may want to organize a set of Airbnb listings by region to prepare a report. The database should persist this layout, along with the data, enabling the realtor to recall where specific data or analysis is located in order to complete a task, and create multiple such layouts for different tasks.

This imposes a new challenge: *How do we support irregular layouts of data using the rigid relational model*?

Our insight here is that we may use a hybrid representation by carving out dense tabular regions of the spreadsheet, directly stored as database tables, and the remaining sparse cells, stored as key-value pairs, with the position as the key. We can also take into account access patterns, *e.g.*, via formulae. Unfortunately, identifying the optimal hybrid representation is NP-HARD; however, we have developed near-optimal approaches that yield substantial reductions in storage (up to 50%), and formula evaluation time (up to 50%)—see Chapter 3.

Order and Position. In a spreadsheet-like interface, users can organize the data in very specific ways, tailored to their task. For example, our realtor can manipulate the order of the listings (*e.g.*, by sorting the listings by district), and then drag the rows corresponding to "Queens" to be next to "Brooklyn" to compare those listings on other attributes. Users expect these changes to be persistent and expect to be able to able to refer to the data by position, *e.g.*, by scrolling to or selecting a region of interest. We call these orders as *implicit*, as they are user-defined and do not correspond to any natural ordering based on attribute values. Thus, *how do we support the notion of order (both implicit and explicit) and position within a database, which is unordered in both rows and columns*? One simple approach is to simply support this by adding an attribute that captures the position, *e.g.*, the row, for each row in the interface. However, even a small change, such as the addition of a row, can cause the cascading updates of the row numbers of all subsequent rows.

To address this, we can encode the position attribute via monotonically increasing proxies such that the insertion or deletion does not impact subsequent tuples. Then, we can use a hierarchical B+tree-like indexing structure that appends each node with the counts of nodes in the subtree below, allowing us to identify the tuple in the k-th position in $O(\log n)$. In practice, our encoding and indexing scheme ensures interactivity ($\approx$a few ms) for billions of cells—see Chapter 3.

2.1.2 Supporting Operation Awareness

We now discuss how to enable *operation awareness* in a database, via *small operations* with *guaranteed responsiveness.*

Guaranteed Responsiveness on Small Operations. Direct manipulation presents new challenges for query optimization and execution. Rather than writing complex SQL queries, users perform computations in small, incremental steps using formulae, embedded along with data— often as many formulae as items of data. Formulae can refer to the results of other formulae, introducing dependencies. The *scale of queries and the notion of dependencies* introduce new challenges, not found in traditional query optimization. Even on present-day spreadsheets that have many formulae, formula computation ends up leading to system unresponsiveness spanning from minutes to even hours [12].

To facilitate formula computation, we need to capture formula dependencies in a dependency graph, which could be arbitrarily deep and wide. One open question here is: *how and where do we maintain this graph as users add or delete formulae*? Given such a representation, we can make formula computation more responsive if we incorporate two techniques: *asynchronous computation*—as described in Chapter 4, allowing computation to happen in the background while users can still interact with the sheet, and *lazy computation*—since users are only viewing a window at a time, prioritizing for what they are seeing over what they are not. While enabling these techniques, we need to ensure that users see a *consistent view* of the analysis—no stale values should be shown.

Enabling both of these techniques leads to a number of challenges: *(i) How do we identify and optimize for the dependencies*? To prioritize for the user window, we need to quickly traverse the large dependency graph to find the formulae that are impacted. One approach is to store a compressed version of this dependency graph that can be traversed efficiently to identify dependent formulae, with minimizing false positives (identifying a formula as a dependent, when it is not), but no false negatives. *(ii) How do we identify and optimize for the redundancies*? When executing these formulae, since many of them have a similar structure and refer to the same data, we can share access and computation. This challenge, like the previous one, is reminiscent of multi-query optimization. However, the new structural characteristics, in terms of the dependencies and redundancies, make the challenges distinct and novel, necessitating new techniques—see Chapter 4.

2.2 FIRST PROTOTYPE: UNIFYING DATABASES AND SPREADSHEETS

We build the first version of DataSpread to explore the research issues discussed in the previous section. Externally, DataSpread retains many of the front-end user interface aspects of spreadsheets that make it as easy to use, while at the same time enhanced and supported by a back-end relational database, providing efficiency and expressiveness. In the front-end, in addition to all the traditional spreadsheet commands, DataSpread supports the use of arbitrary SQL via custom `DBSQL` and `DBTable` commands, enabling the import, and constant updating of data from relational databases, as well as the computation of selections and joins of data contained in the spreadsheets. Conceptually, these commands, along with other spreadsheet commands, are stored as *interface views* in the underlying database. In the back-end, an optimizer, optimizes for keeping the user window up-to-date and in-sync with the underlying relational database. Even though the spreadsheet can only support a few rows, as the user pans through the spreadsheet, the burden of supplying or refreshing the current window is placed on the relational database, which is very efficient.

In the next section we propose a desired design by developing a unification semantics. We then use the semantics to propose an architecture for DataSpread. Finally, we discuss usage scenarios for our DataSpread prototype.

2.2.1 Design of DataSpread

In this section, we describe the semantics for DataSpread. In particular, we discuss some important concepts and challenges that arise due to the unification of the two disparate ideologies: spreadsheets and databases.

DataSpread Overview

With a goal of unifying databases and spreadsheets, we now propose a framework for DataSpread based on two key ideas. First, to leverage the intuitiveness and the richness of a spreadsheet interface, rather than changing it significantly, we enhance it with concepts borrowed from databases. Underneath the interface, we propose to have a relational database that is enhanced to support the spreadsheet interface. Second, to improve the expressiveness of the interface, we expose some database features, for example, declarative querying, from the underling database to the interface. Using these two key ideas, we enable users to leverage the strengths of both spreadsheets and databases for dealing with tabular data.

Semantics and Syntax

Although spreadsheets and databases have both been designed to manage data in form of tables, their treatment of this data is vastly different. Spreadsheets have been developed primarily with presentation of data in mind and hence their design focuses primarily on simplicity, intuitiveness and a rich user interface. On the other hand databases have been designed with powerful data management capabilities to work with large tables. Hence, certain data manipulation operations, e.g., queries, joins, summarization, are very naturally expressed as SQL statements in databases.

We propose semantics for DataSpread such that we are able to naturally leverage the strengths of both systems. Since we plan to enrich databases to effectively support interfaces, we use the strong points of spreadsheets to motivate our semantics.

Support for Dynamic Schema. Spreadsheets enable users to effortlessly create tables and update their schema. A user typically structures data on a spreadsheet as tables, with columns and rows, where columns generally correspond to attributes and rows to tuples. Here, adding an attribute, which is essentially a change to schema, is as natural and convenient as adding a tuple. This is due to the fact that spreadsheets do not treat columns and rows differently when we consider the operations possible on each. On the other hand, relational databases have a schema-first data model. Relational tables, which belong to a database's schema, need a predefined structure in terms of attributes. Since changing the structure of a table in a database requires an update to all its tuples, it is not efficient as adding, deleting or updating the tuples of the table.

To make relational table creation as effortless as table creation on a spreadsheet, we propose the ability for a user to select an arbitrary range on the spreadsheet and use it to define the structure and the data for a table within the database. Once created it should behave like a regular table within the database, and the user should be able to refer to it and use it in queries.

To streamline the concept of a dynamic schema, we propose that a user is able to update a table's schema and tuples that are displayed on a spreadsheet, which in turn updates the schema and tuples of the underlying table in the database. Further, the database should be able to handle this schema change with an efficiency similar to tuple updates. This makes table updates within a database as natural as updating them on a spreadsheet.

Make Databases Interface Aware. Since spreadsheets have been designed with an interface in mind, they very naturally lay out data that is both consumed and manipulated by users. This interface has a very strong influence on functionality offered to the user. Features like laying out a table in a desired format and obtaining the totals of some attributes beneath the table (using a spreadsheet formula) feel natural. Thus, the interface provides a *context* to the operations performed on a spreadsheet.

Positional addressing, which enable users to address data based on its position on a spreadsheet, is an intuitive and effective way to refer to presented data. By laying out data on a spreadsheet, a position gets implicitly assigned to the displayed data, due to which a spreadsheet is able to use positional referencing, e.g., a cell reference of `A2` from cell `C2` implies a cell that is two columns left and in the same row. The positional referencing is a commonly used feature while building expressions as it enables us to copy expressions across cells while still maintaining the relative references.

Conversely, databases completely lack interface aspects. Once a query result is output, the database is no longer cognizant of how that result is consumed. This disconnect is a key weakness due to which a database cannot be used as-is to effectively support a spreadsheet interface. For instance, when a user wants to update a specific attribute of a displayed table, the database is unable to help because it is not aware of the tuple or attribute being modified.

We propose to make databases aware of the interface's data layout. This enables them to understand interactions on the presented data, e.g., for a join using displayed tuples, the database is able to identify the tuples just based on their implicit context. This further enables the databases to optimize the query execution by prioritizing the displayed tuples over the ones that are not displayed.

After making the database interface aware, we propose to leverage this to enable positional addressing in databases. This implies that the user should be able to refer to a value by its location on the spreadsheet and use it in any arbitrary query.

Novel Spreadsheet Constructs. We now describe how the positional addressing is leveraged in the front-end spreadsheet, enabling users to pose rich SQL queries while referring to data in the spreadsheet as well as the underlying relational table.

We encapsulate SQL references within the spreadsheet using one of two formulae: DBSQL and DBTable. DBSQL enables users to pose arbitrary queries combining data present on the spreadsheet, and data stored in the relational database. DBTable enables users to declare a portion of the spreadsheet as being either exported to or imported from the relational database, i.e., that portion of the spreadsheet directly reflects the contents of a relational database table.

In order to support arbitrary positional addressing or referencing of data on the spreadsheet for DBSQL, we add two new constructs: RangeValue and RangeTable. This enables users to refer to a cell and a table on a sheet respectively relative to the cell where the query is entered. RangeValue enables a user to refer to scalar values contained in a cell, e.g., SELECT FROM Actors WHERE ActorId = RangeValue(A1), referring to the value in cell A1. RangeTable on the other hand enables a user to refer to a range, and perform operations on it assuming it is a regular database table. This enables any range on a spreadsheet to be potentially a table, and all the operations, e.g., join, that the database allows on a table can be performed, e.g., SELECT FROM Actors NATURAL JOIN RangeTable(A1:D100).

Other Semantic Issues. Although we have discussed two important concepts, there are still many semantics that require attention if we want to realize a complete unification. Due to the space restriction, rather than discussing them in detail we have listed a few of them below: *a*) SQL support on spreadsheets: To leverage the expressiveness of SQL and the simplicity of formulae we propose to support both, and give flexibility to the user to interchangeably use either. *b*) Real-time sync: Using spreadsheets users are accustomed to having an always updated copy with them. For this we propose a real time two way synchronization of the displayed on the spreadsheet with the underlying database. *c*) Data typing: Spreadsheets dynamically type the data stored as cells. To make this work with databases, we propose the idea of automatically assigning data types within the databases based on the tuples. *d*) Computation optimization: By scaling up the amount of data, which can be presented on a spreadsheet, efficient computation become a necessity. We propose to leverage the presentation information for prioritizing computations for the data that is displayed. *e*) Lazy Computation: To maintain interactivity, we propose that the calculations of the visible cells should be prioritized and the remaining long running computations should be performed in background.

Challenge. Realizing the unified semantics is not a trivial task, since it stretches the capabilities of today's relational databases beyond what is available. For example, consider the semantics of schema, for today's databases a table's schema change requires an update to all the tuples of the table. Further, the activity is considered as "data definition language" and generally cannot participate in transactions. This requires us to propose the architecture of DataSpread by radically rethinking the databases' architecture.

2.2.2 Proposed Database Architecture

Since relational databases are not designed to be interface-aware, when we unify the presentation layer of spreadsheets with databases, we need to redesign the underlying architecture of the database, as well as the interaction with the front-end interface.

To enable databases to support the semantics described earlier, we propose a redesigned database architecture as shown in Figure 2.2, where the shaded blocks represent new or enhanced components. The *interface manager* is tasked with the goal of making databases

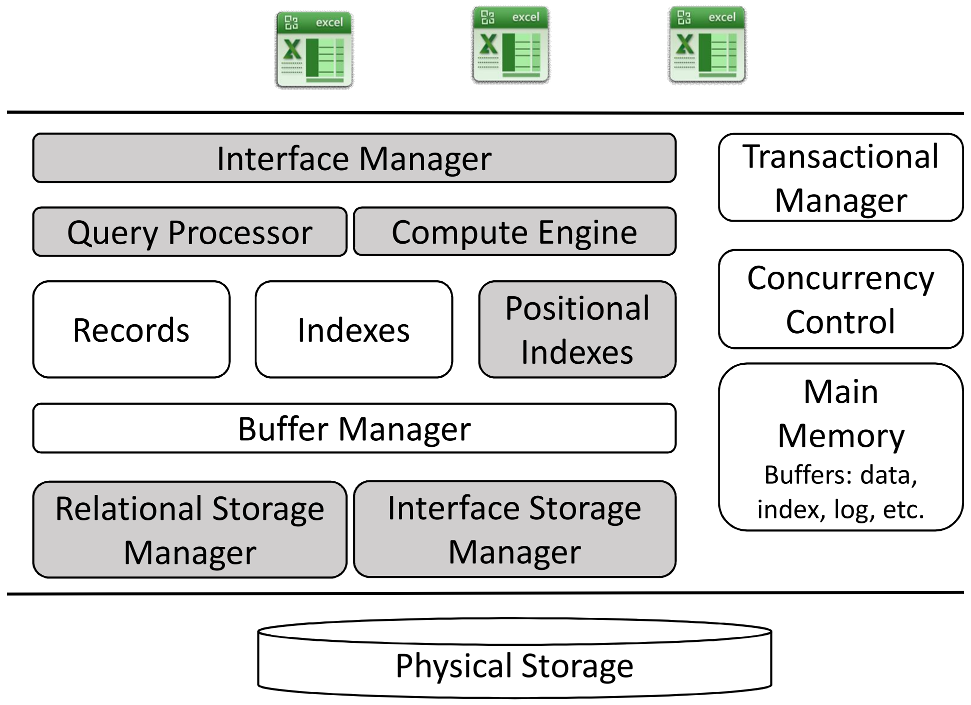
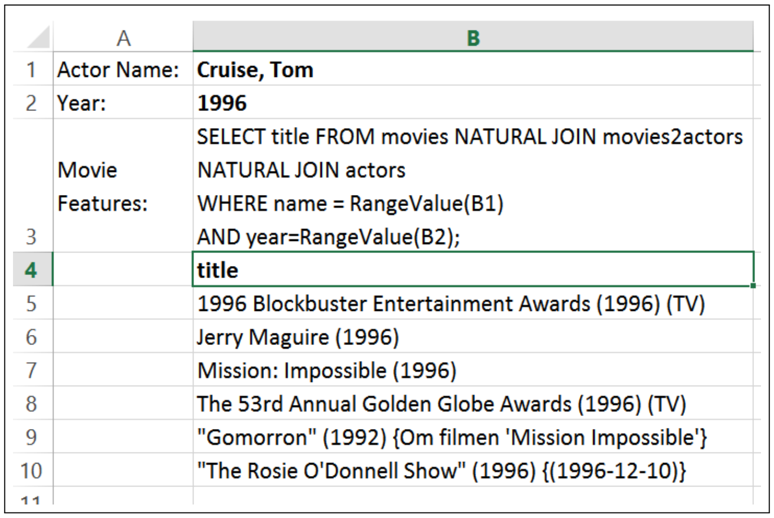


Figure 2.2: DataSpread Architecture.

interface-aware. The *query processor* is enhanced to support and optimize the execution for positional addressing, a natural way to locate data presented on the interface. The *compute engine* leverages interface aspects, e.g., windowing, to optimize execution. We introduce a new type of index, *positional*, which makes interface-oriented operations, e.g., ordered presentation, efficient. The *interface storage manager* stores data that is presented on the interface but not designated as a relational table. The *relational storage manager* is enhanced to effectively support interface related operations such as schema changes.

While we have identified the extent of modifications needed for databases to effectively support an interface, our current implementation and discussion focuses on enhancing some core components. Naturally, there are other components that require modification, such as the transaction manager, and we leave them for future work.

Interface Storage Manager. In this unified framework, a spreadsheet not only has tabular data, corresponding to relational tables in the underlying database, but also has other interface data, e.g., formulae or data entered by the user. This interface data requires special treatment as it does not have a schema. The interface storage component stores this



	A	B
1	Actor Name:	**Cruise, Tom**
2	Year:	**1996**
3	Movie Features:	SELECT title FROM movies NATURAL JOIN movies2actors NATURAL JOIN actors WHERE name = RangeValue(B1) AND year=RangeValue(B2);
4		**title**
5		1996 Blockbuster Entertainment Awards (1996) (TV)
6		Jerry Maguire (1996)
7		Mission: Impossible (1996)
8		The 53rd Annual Golden Globe Awards (1996) (TV)
9		"Gomorron" (1992) {Om filmen 'Mission Impossible'}
10		"The Rosie O'Donnell Show" (1996) {(1996-12-10)}

Figure 2.3: Executing SQL with relative referencing.

data as a collection of cells. To enable efficient retrieval for a given range, the component groups the cells together by proximity and splits the groups into data blocks as required by the underlying storage. To enable efficient access, the blocks are further indexed by a two-dimensional indexing method.

Relational Storage Manager. Our unification semantics demand that the schema changes to the tabular data, which we persist in the database as relational tables, should be very efficient, almost as efficient as changes to tuples. With an insight to reduce the disk blocks to update during a schema change, the relational storage manager uses a hybrid of column-store and row-store to physically store the table. Here, data is structured along a collection of attribute groups, thereby radically reducing the disk blocks that need an update during a schema change.

Interface Manager. The interface manager keeps close tabs on the data presented to the user. For every data item, e.g., the output of a query, a table imported from the database, that is displayed on the interface, the presentation manager assigns a *context*; a context comprises a positional address along with a reference to the sheet. This context can then be utilized to enable functionalities such as two-way sync and relative addressing.

Along with positional addressing, the interface manager allows a two-way synchronization for the tables displayed on the interface. Since primary keys are a natural way to identify tuples in a relational database, the interface manager maintains a mapping between a tuple's key attribute and its corresponding location. This enables translation of an update on the interface, having a locational context, to the underlying relational database, which requires a key to uniquely identify a tuple.

Compute Engine. To optimally support interface interactions and data updates, we introduce a new component termed as "compute engine". By using ideas like shared computation, the compute engine enables efficient handling of formulae and queries with positional referencing, e.g., DBSQL. It performs computations asynchronously, free from a user's context, as updates are made to either the interface or the database. It further improves the interface's interactivity by prioritizing the computation for visible cells.

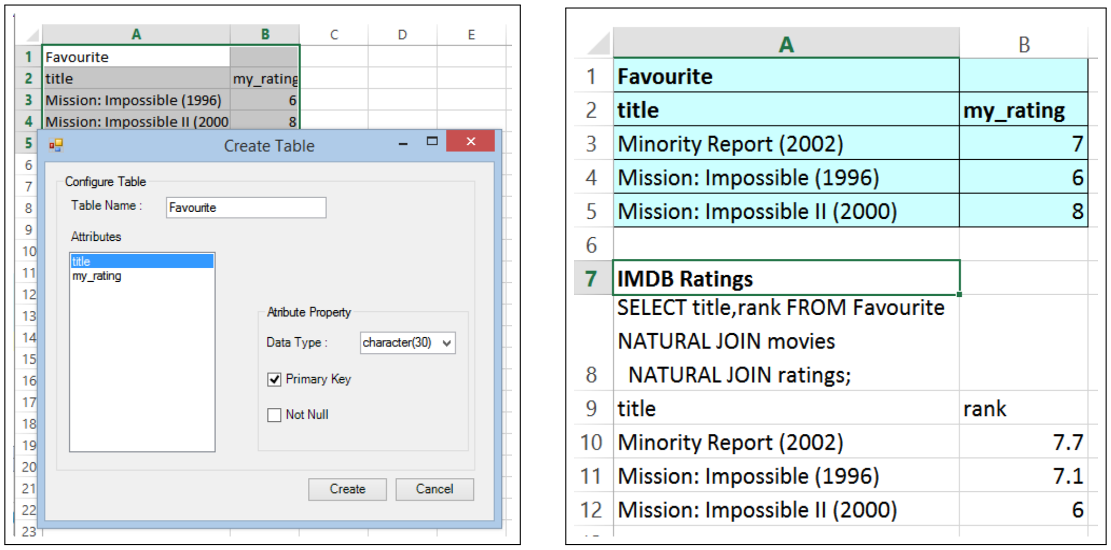
2.2.3 Usage Scenarios

Our DataSpread prototype is implemented using Microsoft Excel (that presumably most conference attendees as well as eventual users are already familiar with) as the front-end spreadsheet application, backed by PostgreSQL as the relational database back-end. All the screenshots we depict are from our current prototype.

We describe the following features of the DataSpread prototype: *a*) Analytic queries that reference data on the spreadsheet, as well as data in other database relations. *b*) Importing or exporting data from the relational database. *c*) Keeping data in the front-end and back-end in-sync during modifications at either end.

Feature 2.1: Querying. Consider Figure 2.3. Here, expressed using the DBSQL spreadsheet function, the SQL query in B3 uses data from three relations in the database (movies, movies2actors, actors), and references the two cells above (B1 and B2), via special relative referencing commands (RangeValue(B1) and RangeValue(B2)). The output of the query is not limited to a single cell, but spans the range B3:B10. This enables the collection of cells to be computed collectively in a single pass (as opposed to traditional spreadsheet formulae that are one-per-cell). Thus, DataSpread provides the ability to naturally query the underlying database, and other data in the spreadsheet.

Feature 2.2: Import/Export. Consider Figure 2.4a. Here, on selecting a range in



	A	B	C	D	E
1	Favourite				
2	title	my_rating			
3	Mission: Impossible (1996)	6			
4	Mission: Impossible II (2000	8			

Create Table

Configure Table

Table Name : Favourite

Attributes: title, my_rating

Atribute Property

Data Type : character(30)

[✓ Primary Key]

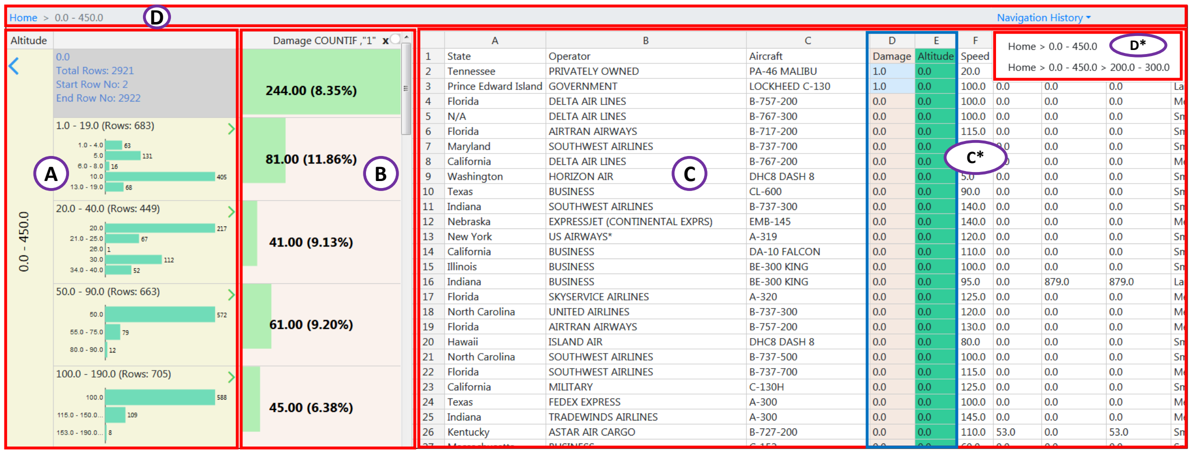
Not Null

	A	B
1	**Favourite**	
2	**title**	**my_rating**
3	Minority Report (2002)	7
4	Mission: Impossible (1996)	6
5	Mission: Impossible II (2000)	8
6		
7	**IMDB Ratings**	
8	SELECT title,rank FROM Favourite NATURAL JOIN movies NATURAL JOIN ratings;	
9	title	rank
10	Minority Report (2002)	7.7
11	Mission: Impossible (1996)	7.1
12	Mission: Impossible II (2000)	6

Figure 2.4: (a) Table creation. (b) Two-way table sync.

the sheet and selecting the create table command from the add-ins menu, we provide the ability to users to transform it into a relational database table. The schema of this table is automatically inferred using the column heading and the data. Optionally, users will be allowed to specify constraints on the table, such as primary keys. On completion, the table is created in the underlying database. The data on the sheet is replaced by DBTable, which is a spreadsheet function that selects data from the database and displays it on the spreadsheet. DBTable could also be used to directly import data already present in the relational database into the spreadsheet. Thus, DataSpread allows us to import or export data to and from the relational database.

Feature 2.3: Modifications. Consider Figure 2.4b. Here, after a table is displayed on the spreadsheet using DBTable, and formatted in cells A3 to B5, as modifications are made to the table on the front-end the data in the relational database is updated, and the data displayed in cells from A10 to B12 (corresponding to a DBSql command referencing that data) is immediately updated. Thus, DataSpread provides the ability to keep data in-sync during modifications at both the front-end and back-end


Home > 0.0 - 450.0

Navigation History

Altitude

0.0 - 450.0

0.0
Total Rows: 2921
Start Row No: 2
End Row No: 2922

1.0 - 19.0 (Rows: 683)

1.0 - 4.0	63
5.0	131
6.0 - 8.0	16
10.0	405
13.0 - 19.0	68

20.0 - 40.0 (Rows: 449)

20.0	217
21.0 - 25.0	67
26.0	1
30.0	112
34.0 - 40.0	52

50.0 - 90.0 (Rows: 663)

50.0	572
55.0 - 75.0	79
80.0 - 90.0	12

100.0 - 190.0 (Rows: 705)

100.0	588
115.0 - 150.0...	109
153.0 - 190.0...	8

Damage COUNTIF ,"1"

244.00 (8.35%)

81.00 (11.86%)

41.00 (9.13%)

61.00 (9.20%)

45.00 (6.38%)

	A	B	C	D	E	F				
1	State	Operator	Aircraft	Damage	Altitude	Speed				
2	Tennessee	PRIVATELY OWNED	PA-46 MALIBU	1.0	0.0	20.0				
3	Prince Edward Island	GOVERNMENT	LOCKHEED C-130	1.0	0.0	100.0	0.0	0.0	0.0	La
4	Florida	DELTA AIR LINES	B-757-200	0.0	0.0	100.0	0.0	0.0	0.0	Me
5	N/A	DELTA AIR LINES	B-767-300	0.0	0.0	100.0	0.0	0.0	0.0	Sm
6	Florida	AIRTRAN AIRWAYS	B-717-200	0.0	0.0	115.0	0.0	0.0	0.0	Sm
7	Maryland	SOUTHWEST AIRLINES	B-737-700	0.0	0.0	[illegible]	0.0	0.0	0.0	Sm
8	California	DELTA AIR LINES	B-767-200	0.0	0.0	[illegible]	0.0	0.0	0.0	Me
9	Washington	HORIZON AIR	DHC8 DASH 8	0.0	0.0	5.0	0.0	0.0	0.0	Sm
10	Texas	BUSINESS	CL-600	0.0	0.0	90.0	0.0	0.0	0.0	Sm
11	Indiana	SOUTHWEST AIRLINES	B-737-300	0.0	0.0	140.0	0.0	0.0	0.0	Sm
12	Nebraska	EXPRESSJET (CONTINENTAL EXPRS)	EMB-145	0.0	0.0	140.0	0.0	0.0	0.0	Me
13	New York	US AIRWAYS*	A-319	0.0	0.0	120.0	0.0	0.0	0.0	Sm
14	California	BUSINESS	DA-10 FALCON	0.0	0.0	110.0	0.0	0.0	0.0	Sm
15	Illinois	BUSINESS	BE-300 KING	0.0	0.0	100.0	0.0	0.0	0.0	Sm
16	Indiana	BUSINESS	BE-300 KING	0.0	0.0	95.0	0.0	879.0	879.0	La
17	Florida	SKYSERVICE AIRLINES	A-320	0.0	0.0	125.0	0.0	0.0	0.0	Me
18	North Carolina	UNITED AIRLINES	B-737-300	0.0	0.0	120.0	0.0	0.0	0.0	Me
19	Florida	AIRTRAN AIRWAYS	B-757-200	0.0	0.0	130.0	0.0	0.0	0.0	Me
20	Hawaii	ISLAND AIR	DHC8 DASH 8	0.0	0.0	80.0	0.0	0.0	0.0	Sm
21	North Carolina	SOUTHWEST AIRLINES	B-737-500	0.0	0.0	100.0	0.0	0.0	0.0	Sm
22	Florida	SOUTHWEST AIRLINES	B-737-700	0.0	0.0	115.0	0.0	0.0	0.0	Me
23	California	MILITARY	C-130H	0.0	0.0	125.0	0.0	0.0	0.0	Sm
24	Texas	FEDEX EXPRESS	A-300	0.0	0.0	100.0	0.0	0.0	0.0	Me
25	Indiana	TRADEWINDS AIRLINES	A-300	0.0	0.0	145.0	0.0	0.0	0.0	Me
26	Kentucky	ASTAR AIR CARGO	B-727-200	0.0	0.0	110.0	53.0	0.0	53.0	Sm

Home > 0.0 - 450.0
Home > 0.0 - 450.0 > 200.0 - 300.0

Figure 2.5: Navigation Panel: (a) multi-perspective representation, (b) accelerated action through formula (chart) computation, (c) spreadsheet-like interface, and (d) navigation context.

2.3 SECOND PROTOTYPE: INTERACTIVE, NAVIGABLE, AND EXPRESSIVE UI

In this chapter, we identify three key shortcomings of traditional spreadsheet tools related to the scale of the data, using Microsoft Excel as a concrete example. *(i) Interactivity.* Excel ceases to be interactive when dealing with computationally heavy spreadsheets. One user posted on Reddit that complex calculations on Excel can take as long as four hours to finish during which the user interface is unresponsive: *"approximately 90% of the time I spend with the spreadsheet is waiting for it to recalculate"*[2]. *(ii) Navigability.* While Excel supports basic browsing, its tabular representation of data does not support quick navigation to desired areas of the spreadsheet, or provide a high-level understanding of the data distribution within a spreadsheet. One Reddit user commented: *"Inexperienced Excel users are unable to navigate data efficiently"*[1]. *(iii) Expressiveness.* Excel uses a cell-at-a-time-based formula query model—this makes it cumbersome to express the much more convenient and powerful table-oriented (or relational) operations when working with tabular data. For example, to filter records that occur between two dates, one of the Reddit users suggested the following Excel formula: `IFERROR(INDEX(A:A, SMALL(IF(($A$20:$A$1000 >=$E$40) * ($A$20:$A$1000<=$F$40),ROWS ($A$20:$A$1000)), ROWS ($C$5:$C5))),"")`[1], which is a cumbersome way to express the simple relational operation.

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

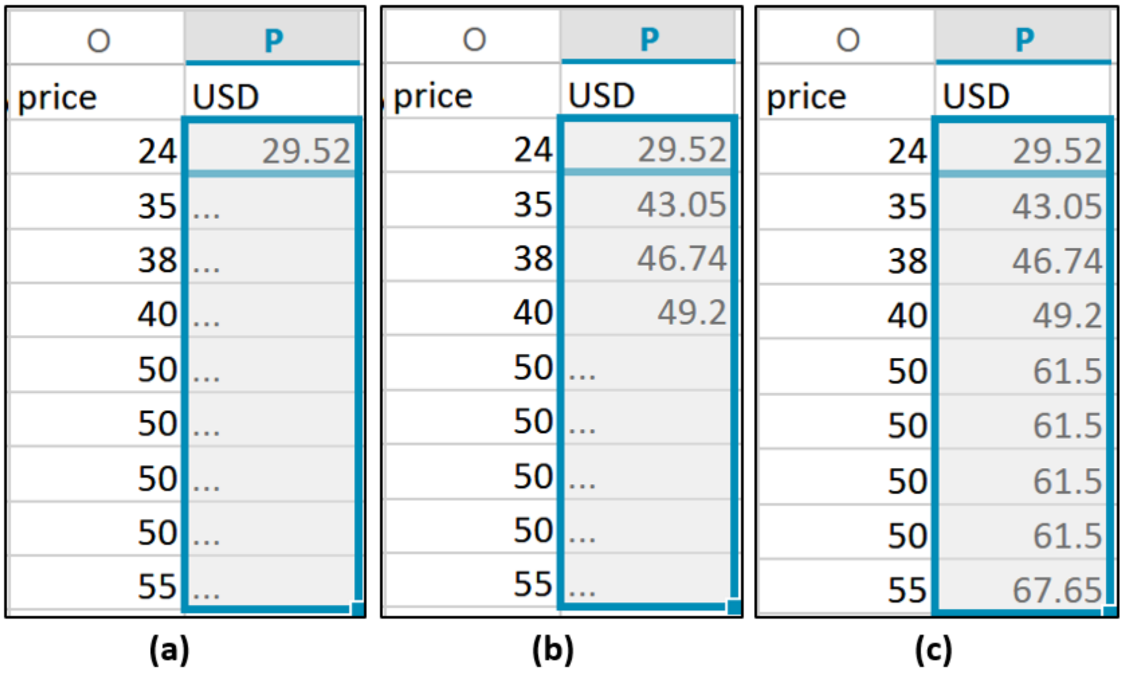
DataSpread addresses the above limitations, making it interactive, navigable, and expressive while working with large spreadsheets, with the following features: *(i)* an *asynchronous and lazy computational model* to address the issue of poor interactivity; *(ii)* a *navigation interface* to enable users to drill-down to desired areas while examining a summarized view of the data to improve navigability; *(iii) support for table-oriented formulae*, a simple but effective means to express relational operations on tabular regions to improve expressiveness. For example, the complex formula described earlier can be expressed using the following table-oriented formula: `SELECT(A20:A1000, ATTR_DATE` $\leq$ `F40 && ATTR_DATE` $\geq$ `E40)`, which is cleaner and easier to understand than the `INDEX` function.

Challenges. Developing the aforementioned features presents a host of engineering and research challenges, ranging from storage and indexing to interface usability. First, maintaining a balance between interactivity and consistency of the asynchronous, lazy computation model requires us to compactly encode the dependencies across formulae to "hide" cells that need recomputation, as well as schedule computation in a way that takes advantage of shared context and locality. Second, seamlessly integrating the navigation interface within the spreadsheet ecosystem introduces design challenges in both the data-structure that can capture changing orders as a first-class citizen, and simultaneously provide summarized representations of the data. Third, introducing table-oriented formulae akin to relational algebra requires careful design to ensure consistency with the cell-at-a-time model of Excel, and the fact that the multi-cell results of table-oriented formulae, when left unchecked, can "overwrite" other data in the spreadsheet.

In this section, we describe a scalable web-based prototype that introduces features that enhance the user experience beyond what traditional spreadsheets provide with a goal to enable the users to work with large spreadsheets efficiently.

2.3.1 Overview of New Features

We now discuss in-depth the new features we developed to address the shortcomings of traditional spreadsheet tools with respect to interactivity, navigability, and expressiveness and the challenges we solved along the way.



(a)

O	P
price	USD
24	29.52
35	...
38	...
40	...
50	...
50	...
50	...
50	...
55	...

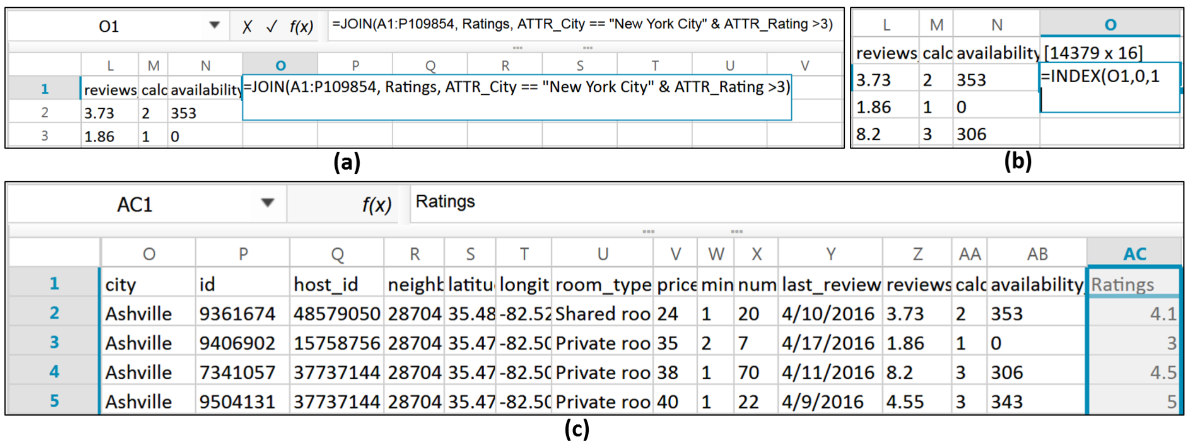
(b)

O	P
price	USD
24	29.52
35	43.05
38	46.74
40	49.2
50	...
50	...
50	...
50	...
55	...

(c)

O	P
price	USD
24	29.52
35	43.05
38	46.74
40	49.2
50	61.5
50	61.5
50	61.5
50	61.5
55	67.65

Figure 2.6: Asynchronous formula execution: (a) user writes formula and make copies by dragging the autofill handle, (b) display partial results, and (c) formula execution completed.



O1 | X ✓ f(x) | =JOIN(A1:P109854, Ratings, ATTR_City == "New York City" & ATTR_Rating >3)

	L	M	N	O	P	Q	R	S	T	U	V
1	reviews	calc	availability	=JOIN(A1:P109854, Ratings, ATTR_City == "New York City" & ATTR_Rating >3)							
2	3.73	2	353								
3	1.86	1	0								

(a)

L	M	N	O
reviews	calc	availability	[14379 x 16]
3.73	2	353	=INDEX(O1,0,1
1.86	1	0	
8.2	3	306	

(b)

AC1 | f(x) | Ratings

	O	P	Q	R	S	T	U	V	W	X	Y	Z	AA	AB	AC
1	city	id	host_id	neighb	latitu	longit	room_type	price	min	num	last_review	reviews	calc	availability	Ratings
2	Ashville	9361674	48579050	28704	35.48	-82.52	Shared roo	24	1	20	4/10/2016	3.73	2	353	4.1
3	Ashville	9406902	15758756	28704	35.47	-82.50	Private roo	35	2	7	4/17/2016	1.86	1	0	3
4	Ashville	7341057	37737144	28704	35.47	-82.50	Private roo	38	1	70	4/11/2016	8.2	3	306	4.5
5	Ashville	9504131	37737144	28704	35.47	-82.50	Private roo	40	1	22	4/9/2016	4.55	3	343	5

(c)

Figure 2.7: JOIN operation in DataSpread: (a) user writes the JOIN formula, (b) user writes the INDEX formula to retrieve JOIN results, and
(c) results displayed.

Asynchronous Computational Model

In Excel, each change the user makes (*e.g.*, changing values or formulae), triggers a sequence of recomputation of formulae, which may take minutes to complete, depending on the size of the data. Excel only returns control to the user when the computation is complete, adopting a *synchronous computation model*—here, the user is kept waiting until the control returns, disrupting interactivity. DataSpread adopts an *asynchronous computation model* instead, returning control back to the user immediately, while masking the "dirty" cells (*i.e.*, those whose values have not been computed yet) using ellipsis ("...") on the interface (see Figure 2.6), and computing them lazily in the background, exploiting shared computation, and prioritizing for what is seen over what is not seen. Thus, this model ensures interactivity by bounding the time for which the system remains unresponsive after an update.

Instead of masking the "dirty" cells, a simpler approach would be to display the current (stale) values of these cells, and not mark them inaccessible with a "..." on the interface. However, this approach can confuse the user by showing them inconsistent data, hence one of our goals is to ensure consistency of data shown to the user at all times. Thus, we adopt the approach of using "..." to indicate cells whose values have not been computed yet.

Challenges. The primary challenge for enabling asynchronous computation is to maintain consistency and interactivity at the same time. This requires addressing two problems both of which are NP-Hard: identifying the impacted cells after an update on a spreadsheet in a bounded period of time, and determining how to compute the impacted cells. To quickly return control of the spreadsheet back to the user, we also require indexing mechanisms that can support positional (*i.e.*, row and column number based) access, which is required by the formulae embedded within spreadsheets.

Insight. Spreadsheet formulae introduce dependencies between different cells on a spreadsheet, which DataSpread captures via a *dependency graph*. The dependency graph can tolerate false positives, *i.e.*, identifying a region as being impacted by an update, even when it is actually not, and can be compressed lossily; false negatives are not permitted, as they violate consistency. We develop a greedy compression algorithm to tackle the NP-Hard problem of dependency graph compression to minimize false positives. Using this lossily compressed graph, DataSpread can identify the impacted cells in a bounded time, ensuring interactivity. Scheduling the computation of the impacted cells is also an NP-Hard problem. We implement a variant of the weighted shortest job first problem [30] to compute the dirty cells efficiently in a cache-friendly manner and prioritize visible cells, thereby minimizing the time that users see dirty cells. These execution algorithms are aided by novel data models that minimize the amount of data accessed, as well as positional indexing mechanisms that allow rapid access of data by position, as described in Chapter 3.

Navigation Interface

Say a user wants to access a specific area of interest within a spreadsheet. At present, the user would have to resort to scrolling to skim over the spreadsheet to arrive at the desired area, which can be painful if the spreadsheet is large. Thus, present spreadsheet tools such as Excel lack a navigation interface. In DataSpread, we have developed a navigation interface, which presents a hierarchical view of the spreadsheet, thereby providing an interactive and effective alternative to basic spreadsheet operations such as scrolling and filtering. DataSpread's navigation interface organizes and summarizes the spreadsheet so that users can skip over irrelevant regions and access the desired area via simple clicks as opposed to scrolling endlessly (see Figure 2.5a).

Challenges. The primary challenge in introducing such an interface alongside the traditional interface is to seamlessly integrate both interfaces: this integration should allow users to effortlessly perform interactions on both interfaces, enabling rapid interactive exploration and drill-down. In addition, we need a data structure that satisfies the requirements of such an interactive navigation interface, *e.g.*, for dynamic reordering and summarization of data. As the scale of the data grows, maintaining the data structure such that the navigation interface remains interactive is an added challenge.

Insight. Similar to the idea adapted by online maps, *e.g.*, Google Maps, the hierarchical navigation interface abstracts the tabular data at different levels of granularity, where users can freely move across different granularities. To enable seamless integration of the navigation interface with the spreadsheet data, we leverage our earlier work on hierarchical positional indexes for tabular data on a spreadsheet. Each level in the positional index maps to a corresponding level in the hierarchical navigation interface. At the lowest level of hierarchy, we display the raw spreadsheet data. At each level of the hierarchy, DataSpread abstracts the spreadsheet by a group of blocks—the grouping is determined based on the distribution the data. Each block contains aggregated information corresponding to the spreadsheet region it spans. Each block contains the block name, number of rows, range of the rows, and a histogram depicting the distribution of the corresponding data (see Figure 2.5a). Users can get an overview via the navigation interface (Figure 2.5a) and organize the data by different attributes (Figure 2.5b). The blocks on the interface are shortcuts that enable users to quickly jump to areas of interest within the spreadsheet.

Table Oriented Formulae

Present spreadsheet tools such as Excel do not support computation that go beyond the cell-at-a-time metaphor: for example, relational algebra operations such as general joins are not permitted or are at least not straightforward (*e.g.*, VLOOKUPs for key-foreign key joins). In DataSpread, we aim to support both SQL as well as general-purpose relational computation via *table-oriented formulae*, supporting operations such as joins, on both tables from the underlying database or spreadsheet regions, *e.g.*, A3:D4, which are treated as tables. Table-oriented formulae retain the semantics of typical formulae on spreadsheets, while also empowering users to use relational primitives.

Challenges. The primary challenge is to ensure that the table-oriented formulae work seamlessly with the cell-at-a-time formula model of Excel in spite of the differences in ideologies and semantics of spreadsheets and databases. Specifically, a table-oriented formula can return multiple records, which makes it incompatible with the standard spreadsheet formula semantics of returning results in a single cell. Returning multiple records is also problematic because unless we are careful, these records can overflow and overwrite other data present on the spreadsheet.

Insight. We support table oriented operations via the following spreadsheet functions: UNION, DIFFERENCE, INTERSECTION, CROSSPRODUCT, JOIN, FILTER, PROJECT, and RENAME. These functions return a single composite table value representing the tabular result of an operation, but this value is not displayed in the cell. We instead show the dimensions of the composite value (similar to matrix dimensions). To retrieve the individual rows and columns within the tabular result encoded as the composite value, we have an INDEX(cell, i, j) function that looks up the (i, j)-th row and column in the composite table value in location cell, and places it in the current location. By forcing users to use INDEX to look up entries in the composite table value, we can avoid the issue of overflow of records, since only the cells that use an INDEX formula will ever refer to data corresponding to the tabular result. Since the input and output of all these functions is a table, the functions can be arbitrarily nested to obtain complex expressions. So for instance, to do a union of three tables, the user would use UNION(table1, UNION(table2,table3)). Alternatively, users can issue SQL queries on DataSpread using the SQL(query, [param1], ...) function, where the first parameter is the SQL query, and the further parameters replace ?'s within the query string. The SQL function is directly executed by the underlying database.

2.3.2 Usage Scenarios

To describe our usage scenarios we consider interactions with DataSpread on a dataset from Airbnb [31], utilizing the new user interface constructs, and contrast their user experience with Microsoft Excel. The Airbnb dataset contains publicly available information about Airbnb listings (*e.g.*, listing type, location, reviews, price, availability) of different cities across the world and has ≈570k rows and 16 columns. Our usage scenarios will simulate the experience of a journalist analyzing the rent price distribution of Airbnb listings using spreadsheets.

Feature 2.3: Asynchronous computation. Suppose the journalist finds out that the rent price of listings in European cities are in € and she wants the price in $. She creates a new column *USD* in the spreadsheet with appropriate formulae to convert the "rent price" to $. The journalist then wants to update the $ to € conversion rate. This impacts all of the cells in the USD column of the sheet. As soon as the attendees update the cell on Excel, the interface will be unresponsive until the computation is complete. On the other hand, the attendees can perform this update in DataSpread interactively; *i.e.*, the control of the sheet is returned back to the user almost immediately. They will notice that some of the impacted cells show ellipses (...), meaning they are under computation 2.6. The ellipses will be computed in the background and eventually, will be replaced with correct values.

Feature 2.4: Navigation Interface. Suppose the journalist now wants to compute the average price of the listings in Chicago. We will ask the attendees to perform similar operations. With Excel, the attendees have to first sort the data by "City" and then filter out the listings belonging to Chicago and finally, compute the average by providing the range of the Chicago listings to the AVERAGE function. Readers will agree how tedious this approach can be for large spreadsheets as it involves scrolling through thousands of rows or move up and down the scrollbar to find the range of the Chicago listings. DataSpread, on the other hand, groups multiple cities based on the alphabetical order to form the highest level of granularity. Figure 2.5a shows two such groups: Ashville-Boston and Chicago-Denver. The attendees can utilize this navigation interface to quickly jump to the region of interest by clicking a block. For example, in Figure 2.5a, clicking on the Chicago-Denver block (highlighted in light blue), enables users to quickly navigate to the corresponding region. Users can drill down further by expanding a block (circled in orange in Figure 2.5c) if required. Using the range information of the region of interest, the attendees can then calculate the aggregate value.

Feature 2.5: Table-oriented Formulae. Assume that our journalist finds another data set that contains average ratings of all the listings in Airbnb. Suppose the journalist wants to see all the listings in NYC that has average rating above some threshold along with their ratings. We will provide a "Ratings" spreadsheet to the attendees and ask them complete the task. The task corresponds to a *natural join*, which is not possible natively in Excel. The closest solution is an *outer join*, which can be done by a combination of VLOOKUP and IF statements which must then be applied to multiple listings by dragging the autofill handle. Using DataSpread, the attendees can use the following two formulae to complete the task: JOIN(A1:P142857, Ratings, City=="NYC" & Rating > 3) and INDEX(O1,0,1) (see Figure 2.7). Note INDEX can be applied to multiple cells by dragging the autofill handle, just like traditional spreadsheet formulae.

CHAPTER 3: SPREADSHEET STORAGE MODELING

In this chapter, we focus on the following fundamental question—*how do we develop a storage manager to support interactive ad-hoc data management at scale?*

Requirements for a Storage Engine. We conducted a survey and user study, described in more detail in Section 3.1, to characterize two key functional requirements for such a storage engine to support the direct manipulation of data in a spatial interface:

(i) Presentational Awareness. Our storage engine must be aware of the layout of data within the spreadsheet interface and be flexible enough to adapt to various ad-hoc modalities users might choose to lay out and manage data (and queries) on spreadsheets, ranging from fully structured tables, to data scattered across the spreadsheet, along with formulae.

(ii) Presentational Access. Our storage engine must support access of a range of data by position: for example, users may scroll to a certain region of the spreadsheet, or a formula may access a range of cells; this access must be supported as a first-class primitive.

Challenges in Supporting a Spatial Interface. In supporting these functional requirements, our first set of challenges emerge in how we can ***flexibly represent spatial information within a database***. A user may manage several table-like regions within a spreadsheet, interspersed with empty rows or columns, along with formulae. One option is to store the spreadsheet as a single relation, with tuples as spreadsheet rows, and attributes as spreadsheet columns—this can be very wasteful for storage and computation due to sparsity. Another option is to store the filled-in cells as key-value pairs: [(row #, column #), value]; this can be effective for sparse spreadsheets, but is wasteful for dense spreadsheets with well-defined tables. One can imagine hybrid representation schemes using both "dense" and "sparse" schemes, as well as those that take access patterns into account. Unfortunately, we show that it is NP-Hard *to identify the optimal representation.*

Our second set of challenges emerge in ***supporting and maintaining spatial access***. Say we use a single relation to record information about a sheet, with one tuple for each spreadsheet row, and one attribute for each spreadsheet column; with an additional attribute that records the spreadsheet row number. Now, inserting a single row in the spreadsheet can lead to an expensive *cascading update* of the row numbers of all subsequent rows; thus, we must develop techniques that allow us to avoid this issue. Moreover, we need *positional indexes* that can access a range of rows at a time, say, when a user scrolls to a certain region of the spreadsheet. While one could use a traditional index such as a B+ tree, on the row number, cascading updates makes it hard to maintain such an index across edit operations.

Our Contributions. In this chapter, we address the aforementioned challenges in developing a scalable storage manager for an interactive ad-hoc data management tool. Our contributions are the following:

1. Understanding Present-day Solutions. We perform an empirical study of four spreadsheet datasets plus a user survey to understand how spreadsheets are presently used for data manipulation and analysis (Section 3.1).

2. Abstracting the Functional Requirements. Based on our study, we define our *conceptual data model*, as well as the operations necessary for interactive ad-hoc data management (Section 3.2).

3. Primitive Representation Schemes. We propose four *primitive data models* that implement the conceptual data model, and demonstrate that they represent "optimal extreme choices" (Section 3.3.2).

4. Near-Optimal Hybrid Representation Schemes. We develop a space of *hybrid data models*, utilizing these primitive data models, and demonstrate that identifying the optimal hybrid is NP-HARD (Section 3.4.1); we further develop multiple PTIME solutions that provide near-optimality (Section 3.4.2), plus greedy heuristics (Section 3.4.3), and show that they can be incrementally maintained (Section 3.4.4).

5. Presentational Access Schemes. We develop solutions to maintain positional information, while reducing the impact of cascading updates (Section 3.5).

6. Storage Engine of DATASPREAD. We have designed the storage engine of DATASPREAD based on the ideas discussed in this chapter (Section 3.6).

Dataset	**Sheets**	**Formulae Distribution**			**Density Distribution**		**Tabular Regions**	
		Sheets with formulae	Sheets with > 20% formulae	%formulae coverage	Sheets with < 50% density	Sheets with < 20% density	Tables	%Coverage
Internet	52,311	29.15%	20.26%	1.30%	22.53%	6.21%	67,374	66.03%
ClueWeb09	26,148	42.21%	27.13%	2.89%	46.71%	23.8%	37,164	67.68%
Enron	17,765	39.72%	30.42%	3.35%	50.06%	24.76%	9,733	60.98%
Academic	636	91.35%	71.26%	23.26%	90.72%	60.53%	286	12.10%

Table 3.1: Spreadsheet Datasets: Preliminary Statistics.

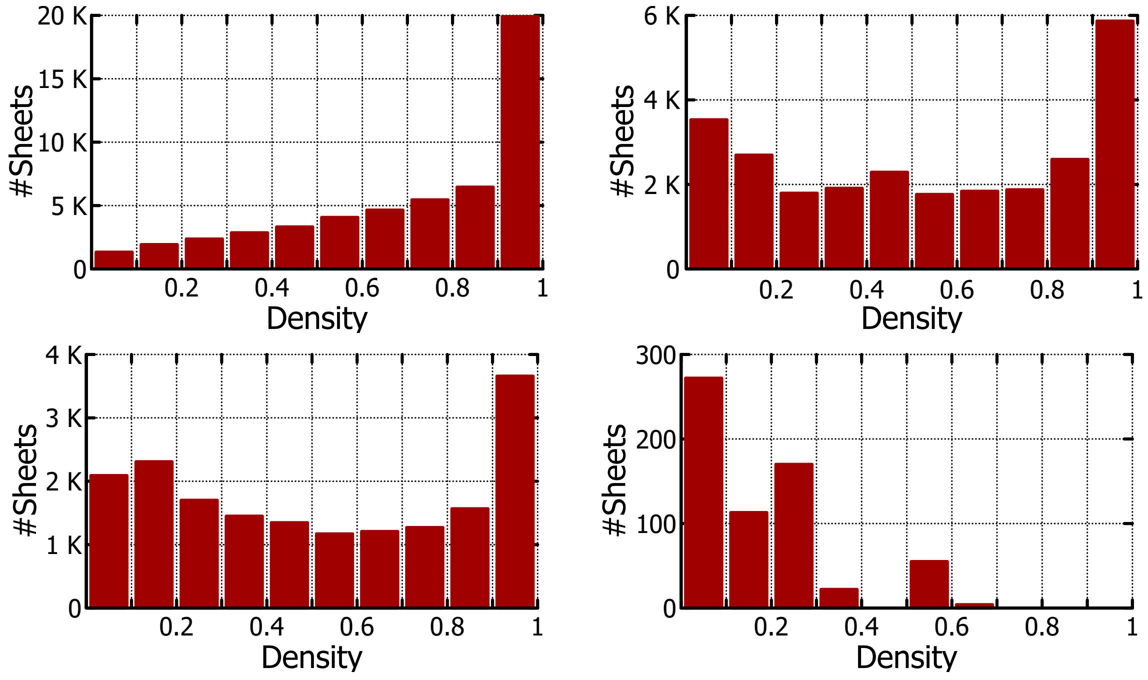


Figure 3.1: Data Density—(a) Internet (b) ClueWeb09 (c) Enron (d) Academic

7. Experimental Evaluation. We evaluate our data models and spatial access schemes on a variety of real-world and synthetic datasets, demonstrating that our storage engine is scalable and efficient. We also conduct a small qualitative evaluation to illustrate how DataSpread handles the use-cases described earlier (Section 3.7).

3.1 UNDERSTANDING REQUIREMENTS FOR A STORAGE ENGINE

We now perform an empirical study to characterize the functional requirements for a storage engine. We focus on two aspects: *(i)* identifying how users *structure* data on the interface, and *(ii)* understanding common interface *operations*. To do so, we first retrieve spreadsheets from four sources and quantitatively analyze them on different metrics. We supplement this analysis with a small-scale user survey to understand the spectrum of operations frequently performed. The latter is necessary since we do not have a readily available

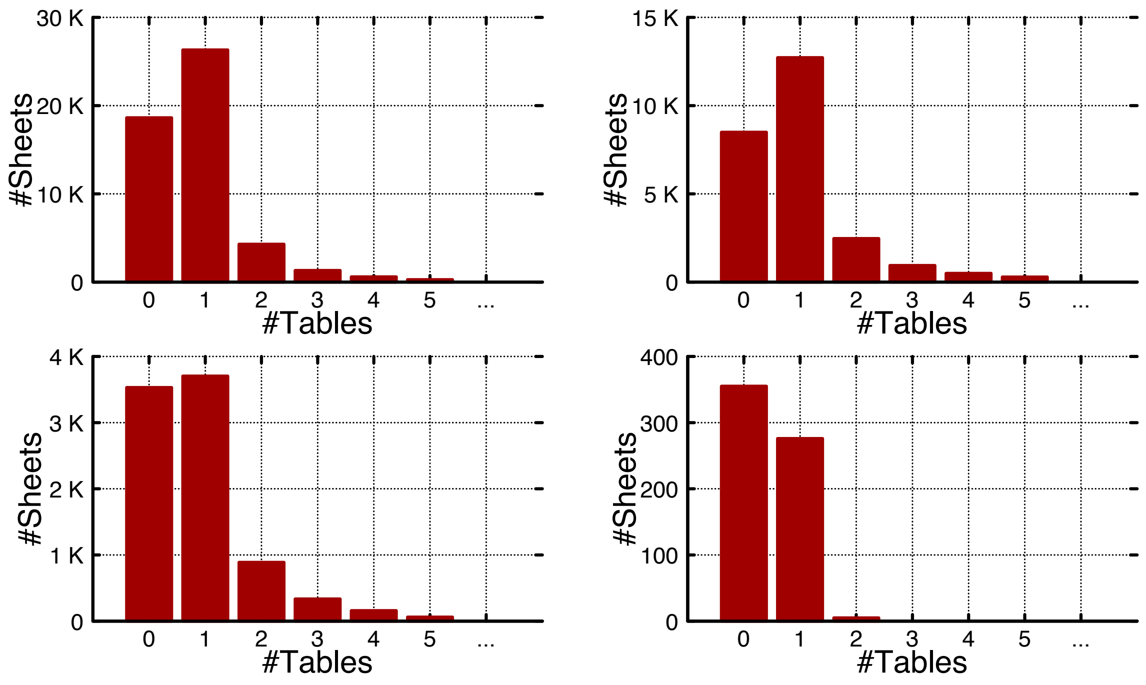


Figure 3.2: Tabular Region Distribution—(a) Internet (b) ClueWeb09 (c) Enron (d) Academic

trace of user operations, *e.g.*, how often do users add rows.

We first describe our methodology for both these evaluations, before diving into our findings.

3.1.1 Methodology

As described above, we have two forms of evaluation described below.

Real Spreadsheet Datasets

For our evaluation of real spreadsheets, we assemble the following four datasets from a variety of sources.

Internet. This dataset of 53k spreadsheets was generated by using Bing to search for .xls files, using a variety of keywords. As a result, these 53k spreadsheets vary widely in content, ranging from tabular data to images.

ClueWeb09. This dataset of 26k spreadsheets was generated by extracting .xls file URLs from the ClueWeb09 [32] crawl.

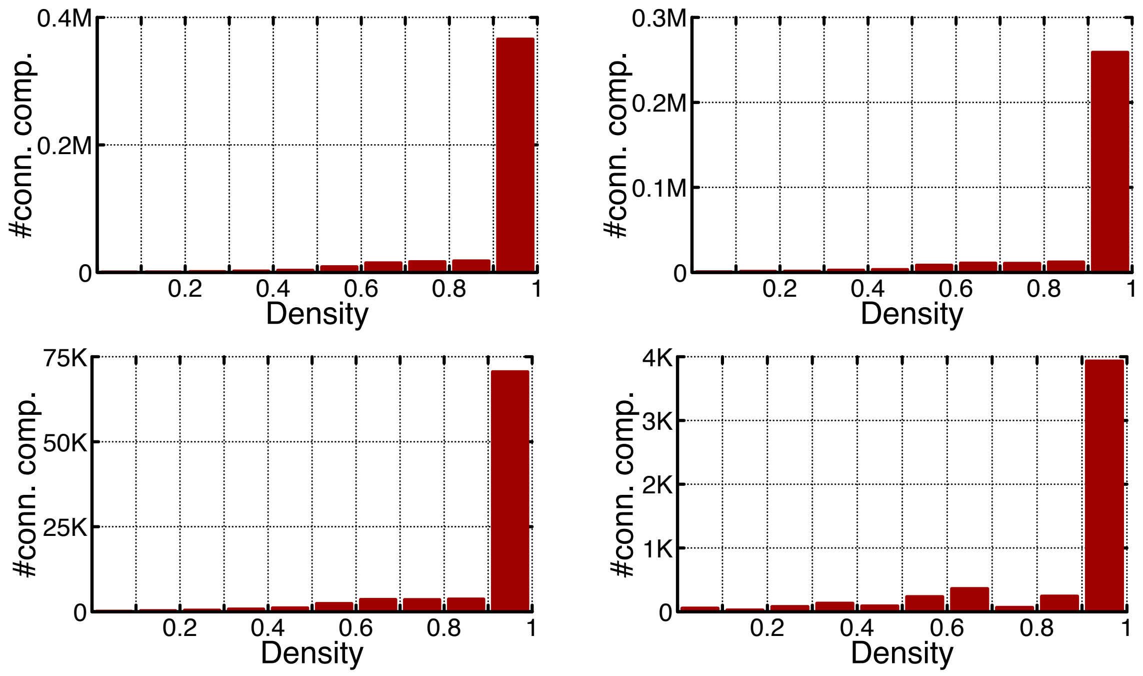


Figure 3.3: Connected Component Data Density—(a) Internet (b) ClueWeb09 (c) Enron (d) Academic

Enron. This dataset was generated by extracting 18k spreadsheets from the Enron email dataset [33]. These spreadsheets were used to exchange data within the Enron corporation.

Academic. This dataset was collected from an academic institution using spreadsheets to manage administrative data.

We list these four datasets in Table 3.1. The first two datasets are primarily meant for *data publication*: thus, only about 29% and 42% of these sheets (column 3) contain formulae, with the formulae occupying less than 3% of the total number of non-empty cells for both datasets (column 5). The third dataset is primarily meant for email-based *data exchange*, with a similarly low fraction of 39% of these sheets containing formulae, and 3.35% of the non-empty cells containing formulae. The fourth dataset is primarily meant for *data analysis*, with a high fraction of 91% of the sheets containing formulae, and 23.26% of the non-empty cells containing formulae.

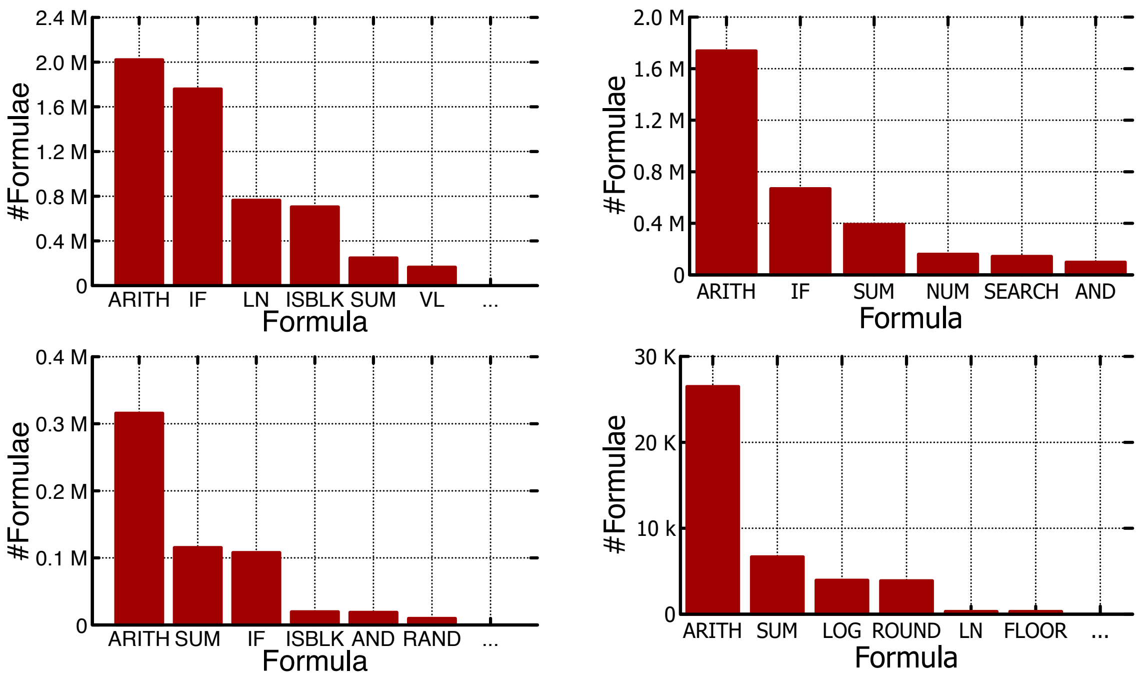


Figure 3.4: Formulae Distribution—(a) Internet (b) ClueWeb09 (c) Enron (d) Academic

User Survey

To evaluate the kinds of operations performed on spreadsheets, we solicited 30 participants from industry who exclusively used spreadsheets for data management, for a qualitative user survey. This survey was conducted via an online form, with the participants answering a small number of multiple-choice and free-form questions, followed by the authors aggregating the responses.

3.1.2 Structure Evaluation

We begin by asking: *how do users structure data in a spatial interface like that of spreadsheets?* Is the data typically organized and structured into tables, or is it largely unstructured? Does the type of structure depend on the intended use-case?

Across Spreadsheets: Data Density. To evaluate whether real spreadsheets are similar to structured relational data, we first we estimate the *density* of each sheet, defined as the ratio of the number of filled-in cells to the number of cells within the minimum bounding rectangular box enclosing the filled-in cells. We depict the results in the last two columns of Table 3.1: the spreadsheets in Internet, Clueweb09, and Enron are typically *dense*, *i.e.*, more than 50% of the spreadsheets have density greater than 0.5. On the other hand, for Academic, a high proportion (greater than 60%) of the spreadsheets have density values less than 0.2. This low density is because the latter dataset embeds many formulae and uses forms to report data in a user-accessible interface.

Takeaway 3.1 (Presentational Awareness)**.** *Structure of data in an spatial interface like that of spreadsheets can vary widely, from highly sparse to highly dense, necessitating data models that can adapt to such variations.*

Within a Spreadsheet: Tabular regions. We further analyzed the sparse spreadsheets to evaluate whether there are regions within them with high density—essentially indicating that these are structured tabular regions. To do so, we first constructed a graph of the filled-in cells within each spreadsheet, where two cells (*i.e.*, nodes) have an edge between them if they are adjacent. We then computed the connected components of this graph. We declare a connected component to be a *tabular region* if it spans at least two columns and five rows, and has a density of at least 0.7, defined as before. In Table 3.1, for each dataset, we list the total number of identified tabular regions (column 8) and the fraction of the total filled-in cells that are captured within these tabular regions (column 9). In Figure 3.2 we plot the distribution of tables across our datasets. For Internet, ClueWeb09, and Enron, we observe that greater than 60% of the cells are part of tabular regions. For Academic, where the sheets are rather sparse, there still are a modest number of regions that are tabular (286 across 636 sheets).

We next characterize the connected components by understanding how they conform to a tabular structure. To study this, we estimate the *density* of each connected component, defined as the ratio of the number of filled-in cells to the number of cells within the minimum bounding rectangular box enclosing the connected component. Figure 3.3 depicts the density distribution of connected components. We note that across all the four data sets the connected components are *very dense*, specifically more than 80% of the spreadsheets have density greater than 0.8.

Takeaway 3.2 (Presentational Awareness)**.** *Even within a single spreadsheet, there is often* *high skew, with areas of high and low density, indicating the need for fine-grained data models that can treat these regions differently.*

3.1.3 Operation Evaluation

We now ask: *What kinds of operations do users naturally perform in a spatial interface like that of spreadsheets?* How often do users employ data manipulation operations? Or analysis operations, *e.g.*, formulae? How do users refer to the portions of data in the operations?

Popularity: Formulae Usage. Formulae use is common, but there is a high variance in the fraction of cells that are formulae (see column 5 in Table 3.1), ranging from 1.3% to 23.26%. We note that Academic embeds a high fraction of formulae since their spreadsheets are used primarily for data management as opposed to sharing or publication. Despite that, all of the datasets have a substantial fraction of spreadsheets where the formulae occupy more than 20% of the cells (column 4)—20.26% and higher for all datasets.

Takeaway 3.3 (Presentational Access)**.** *Formulae are very common, with over 20% of the spreadsheets containing a significant fraction of over $\frac{1}{5}$ of formulae. Optimizing for the access patterns of formulae when developing data models is crucial.*

Formulae Distribution and Patterns. Next, we study the distribution of formulae used within spreadsheets—see Figure 3.4. Not surprisingly, arithmetic operations (ARITH, LN, SUM) are very common, along with conditional formulae (IF, ISBLK). Overall, there is a wide variety of formulae that span both a small number of cell accesses (*e.g.*, arithmetic), as well as a large number of them (*e.g.*, SUM, VL short for VLOOKUP). Moreover, these formulae typically access a small number of rectangular region, *i.e.*, an area defined by a set of contiguous rows and columns, at a time (column 11). Many of the formulae used ended up reproducing relational operations (*e.g.*, VLOOKUP for joins).

To gain a better understanding of how much effort is necessary to execute these formulae, we measure the number of cells accessed by each formula. Then, we tabulate the average number of cells accesses per formula in column 10 of Table 3.1 for each dataset. As we

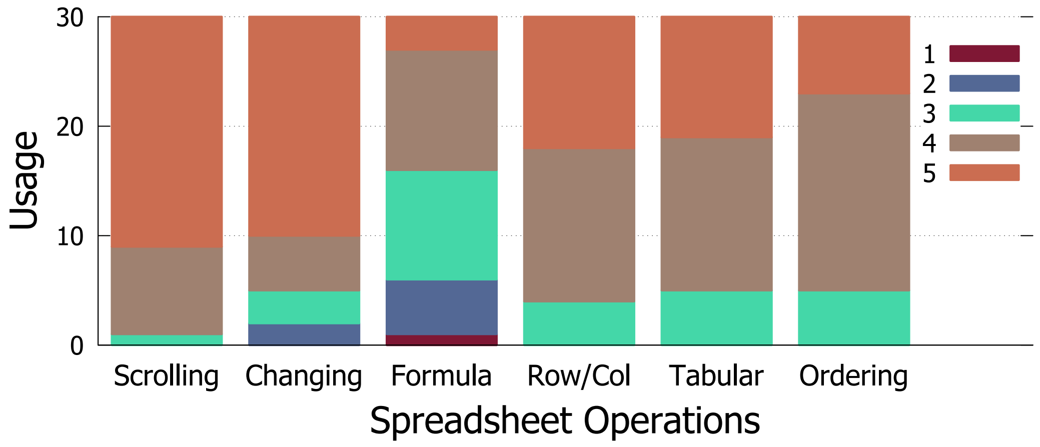


Figure 3.5: Operations performed on spreadsheets.

can see in the table, the average number of cells accesses per formula is not small—with up to 300+ cells per formula for Internet, and about 140+ cells per formula for Enron and ClueWeb09. Academic has a smaller average number—many of these formulae correspond to derived columns that access a small number of cells at a time. Next, we check if the accesses made by these formulae were spread across the spreadsheet, or could exploit spatial locality. To measure this, we considered the set of cells accessed by each formula, and then generated the corresponding graph of these accessed cells as described in the previous subsection for computing the number of tabular regions. We then counted the number of connected components, shown in column 11. Even though the number of cells accessed may be large, these cells stem from a small number of connected components; as a result, we can exploit spatial locality to execute them more efficiently.

Takeaway 3.4 (Presentational Access and Awareness)**.** *Formulae on spreadsheets access cells on the spreadsheet by position; some common formulae such as* SUM *or* VLOOKUP *access a rectangular range of cells at a time. The number of cells accessed by these formulae can be quite large, and most of these cells stem from contiguous areas of the spreadsheet.*

User-Identified Operations. We now analyze the common spreadsheet operations performed by users via a small-scale online survey of 30 participants. This qualitative study is valuable since real spreadsheets do not reveal traces of user operations. Our questions in this study were targeted at understanding *(i)* how users perform operations on spreadsheets and *(ii)* how users organize data on spreadsheets.

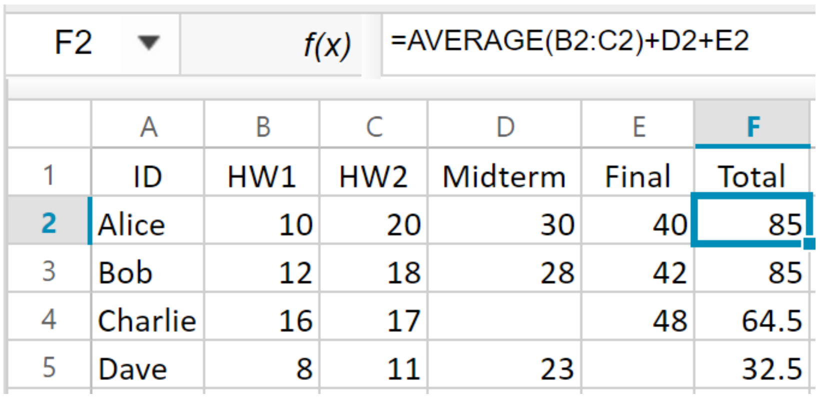
We asked each participant to answer a series of questions where each question corresponded to whether they conducted the specific operation under consideration on a scale of 1–5, where 1 corresponds to "never" and 5 to "frequently". For each operation, we plotted the results in a stacked bar chart in Figure 3.5, with the higher numbers stacked on the smaller ones.

We find that all the thirty participants perform *scrolling*, *i.e.*, moving up and down the spreadsheet to examine the data, with 22 of them marking 5 (column 1). All participants reported to have performed editing of individual cells (column 2), and many of them reported to have performed formula evaluation frequently (column 3). Only four of the participants marked < 4 for some form of *row/column-level operations*, *i.e.*, deleting or adding one or more rows or columns at a time (column 4).

Takeaway 3.5 (Presentational Access and Awareness)**.** *There are several common operations performed by spreadsheet users including scrolling, row and column modification, and editing individual cells.*

Our second goal for performing the study was to understand how users organize their data. We asked each participant if their data is organized in well-structured tables, or if the data scattered throughout the spreadsheet, on a scale of 1 (not organized)–5 (highly organized)—see Figure 3.5. Only five participants marked < 4 which indicates that users do organize their data on a spreadsheet (column 5). We also asked the importance of ordering of records in the spreadsheet on a scale of 1 (not important)–5 (highly important). Unsurprisingly, only five participants marked < 4 for this question (column 6). We also provided a free-form textual input where multiple participants mentioned that ordering comes naturally to them and is often taken for granted while using spreadsheets.

Takeaway 3.6 (Presentational Awareness)**.** *Spreadsheet users typically try to organize their data as far as possible on the spreadsheet, and rely heavily on the ordering and presentation of the data on their spreadsheets.*


F2 f(x) =AVERAGE(B2:C2)+D2+E2

	A	B	C	D	E	F
1	ID	HW1	HW2	Midterm	Final	Total
2	Alice	10	20	30	40	85
3	Bob	12	18	28	42	85
4	Charlie	16	17		48	64.5
5	Dave	8	11	23		32.5

Figure 3.6: Sample Spreadsheet (DataSpread screenshot).

3.2 DATA PRESENTATION MANAGER

Given our findings on spatial awareness and access, we now abstract out the functional requirements of the storage engine. We abstract out the spatial interface of a spreadsheet, as a conceptual data model, as well as the operations supported on it; concrete implementations will be described in subsequent sections.

Conceptual Data Model. A spreadsheet consists of a collection of *cells*, referenced by two dimensions: row and column. Columns are referenced using letters A, . . ., Z, AA, . . .; while rows are referenced using numbers 1, . . . Each cell contains a *value*, or *formula*. A value is a constant; *e.g.*, in Figure 3.6 (a DataSpread screenshot), B2 (column B, row 2) contains the value 10. In contrast, a formula is a mathematical expression that contains values and/or cell references as arguments, to be manipulated by operators or functions. For example, in Figure 3.6, cell F2 contains the formula =AVERAGE(B2:C2)+D2+E2, which unrolls into the value 85. In addition to a value or a formula, a cell could also additionally have formatting associated with it; *e.g.*, width, or font. For simplicity, we ignore formatting aspects, but these aspects can be easily captured without significant changes.

Spreadsheet-Oriented Operations. We now describe the spreadsheet-like operations, drawing from our survey (takeaway 3-5).

1. Retrieving a Range. Our most basic read-only operation is getCells(range), where we retrieve a rectangular range of cells. This operation is relevant in *scrolling*, where the user moves to a specific position and we need to retrieve the rectangular range of cells visible at that position, *e.g.*, range A1:F5, is visible in Figure 3.6. Similarly, *formula evaluation* also accesses one or more ranges of cells.

2. Updating an Existing Cell: The operation updateCell(row, column, value) corresponds to modifying the value of a cell.

3. Inserting/Deleting Row/Column(s): This operation corresponds to inserting/deleting row/column(s) at a specific position, followed by shifting subsequent row/column(s) appropriately: *(i)* insertRowAfter(row) *(ii)* insertColumnAfter(column) *(iii)* deleteRow(row) *(iv)* deleteColumn(column).

Database-Oriented Operations. We now describe the database-like operations, enabling users to effectively use the interface to manage and interact with database tables.

1. Link an existing table/Create a new table: This operation, invoked as linkTable(range, tableName), enables users to *link* a region on a spreadsheet with an existing database relation, establishing a two way correspondence between the spreadsheet interface and the underlying table, such that any operations on the spreadsheet interface are translated by the data presentation manager into table operations on the linked table. Thus, a user can use traditional spreadsheet operations such as updating a cell's value to update a database table. If tableName does not exist, it will be created in the database, and then linked to the spreadsheet interface.

2. Relational Operators: Users can interact with the linked tables and tabular regions via relational operators and SQL, using the following spreadsheet functions: union, difference, intersection, crossproduct, join, filter, project, rename, and sql. These functions return a single composite table value; to retrieve the individual rows and columns within that table value, we have an index(cell, i, j) function that looks up the (i, j)th row and column in the composite table value in location cell, and places it in the current location as described in Sections 2.3.1.

Given the functional requirements for our data presentation manager, in Section 3.3, we develop concrete mechanisms for representing our conceptual data model in a database back-end, and in Section 3.5, we develop data structures that enable efficient access in the presence of updates.

RowID	Col_1	...	Col_6
1	ID, NULL	...	Total, NULL
2	Alice, NULL	...	85, AVERAGE(B2:C2)+D2+E2
...	...	...	...

ColID	Row_1	...	Row_5
1	ID,NULL	...	Dave,NULL
2	HW1,NULL	...	8,NULL
...	...	...	...

RowID	*ColID*	Value
1	1	ID, NULL
...	...	..., ...
2	6	85, AVERAGE(B2:C2)+D2+E2
...	...	..., ...

Figure 3.7: (a) Row-Oriented Model (b) Column-Oriented Model (c) Row-Column-Value Model for Figure 3.6.

3.3 PRESENTATIONAL AWARENESS

We now describe the high-level problem of representation of spreadsheet data within a database. We focus on one spreadsheet, but our techniques seamlessly carry over to the multiple spreadsheet case.

3.3.1 High-level Problem Description

The conceptual data model corresponds to a collection of cells, represented as $C = \{C_1, C_2, \ldots, C_m\}$; as described previously, each cell C_i corresponds to a location (*i.e.*, a specific row and column), and has some contents—either a value or a formula. Our goal is to represent and store C, via one of the *physical data models*, $\mathbb{P}$. Each $T \in \mathbb{P}$ corresponds to a collection of relational tables $\{T_1, \ldots, T_p\}$. Each table T_i records the data in a certain portion of the spreadsheet. Given C, a physical data model T is said to be *recoverable* with respect to C if for each $C_i \in C, \exists$ precisely one $T_j \in T$ such that T_j records the data in C_i. Our goal is to identify physical data models that are recoverable.

At the same time, we want to minimize the amount of *storage* required to record T, *i.e.*, we would like to minimize $\text{size}(T) = \sum_{i=1}^{p} \text{size}(T_i)$. Moreover, we would like to minimize the time taken for accessing data using T, *i.e.*, the *access cost*, which is the cost of accessing a rectangular range of cells for formulae (takeaway 4) or scrolling (takeaway 5), both common operations. And we would like to minimize the time taken to perform updates, *i.e.*, the *update cost*, which is the cost of updating cells, and the insertion and deletion of rows and columns.

Problem 3.1. *Given a collection of cells C, our goal is to identify a physical data model T such that: (i) T is recoverable with respect to C, and (ii) T minimizes a combination of storage, access, and update costs, among all $T \in \mathbb{P}$.*

We begin by considering the setting where the physical data model T has a single relational table, *i.e.*, $T = \{T_1\}$. We develop three simple ways of representing this table, called *primitive data models* all drawn from prior work, each of which works well for a specific structure of spreadsheet (Section 3.3.2). Then, we extend this to $|T| > 1$ by defining *hybrid data models* with multiple tables each of which uses one of the primitive data models to represent a certain spreadsheet region (Section 3.4.1). Given the high diversity of structure within spreadsheets and high skew (takeaway 2), having multiple primitive data models, and the ability to use multiple tables, gives us substantial spatial awareness.

3.3.2 Primitive Data Models

Our primitive data models represent trivial solutions for spreadsheet representation with a single table, stored in a relational row store. This enables DataSpread to support relational algebra primitives and SQL seamlessly, while also providing an interactive front-end for the ubiquitous and popular row stores. Before we describe these data models, we discuss a small wrinkle that affects all of these models. To capture a cell's position we need to record a row and column number with each cell. Say we use an attribute to capture the row number for a cell. Then, any insertion or deletion of rows requires cascading updates to the row number attribute for cells in all subsequent rows. As it turns out, all of the data models we describe here suffer from performance issues arising from cascading updates, but the solution to deal with this issue is similar for all of them, and will be described in Section 3.5. Thus, we focus here on *storage* and *access cost*. Also, note that the access and update cost of data models depends on whether the underlying database is a row or a columnar store. We now describe the primitive data models:

Row-Oriented Model (ROM). The row-oriented data model is akin to the traditional relational data model. We represent each row from the sheet as a separate tuple, with an attribute for each column $Col1$, ..., $Colc_{max}$, where $Colc_{max}$ is the largest non-empty column, and an additional attribute for explicitly capturing the row number, *i.e.*, $RowID$. The schema for ROM is: ROM($\underline{RowID}$, $Col1$, ..., $Colc_{max}$)—we illustrate the ROM representation of Figure 3.6 in Figure 3.7(a): each entry is a pair corresponding to a value and a formula, if any. For dense spreadsheets that are tabular (takeaways 1 and 2), this data model can be quite efficient in storage and access, since each row number is recorded only once, independent of the number of columns. Overall, ROM shines when entire rows are accessed at a time. It is also efficient for accessing a large range of cells at a time.

Column-Oriented Model (COM). The second representation is the transpose of ROM. Often, we find that certain spreadsheets have many columns and relatively few rows, necessitating such a representation. For example, there could be tables where the attributes are laid out vertically, one per row, and the tuples are laid out horizontally, one per column. For our Internet spreadsheet dataset, described in Section 3.7, the number of columns dominate the number of rows for 8% of spreadsheets. The schema for COM is: COM($\underline{ColID}$, $Row1$, ..., $Rowr_{max}$). Figure 3.7(b) illustrates the COM representation of Figure 3.6. Note that COM does not correspond to a traditional column store, which is an orthogonal storage mechanism, but is a rather a transpose of ROM where the tuples are the columns—such spreadsheets can contain over a hundred columns and a handful of rows, which correspond to attributes.

Row-Column-Value Model (RCV). The Row-Column-Value Model is inspired by key-value stores, where the Row-Column number pair is treated as the key. The schema for RCV is RCV($\underline{RowID}$, $\underline{ColID}$, $Value$). The RCV representation for Figure 3.6 is provided in Figure 3.7(c). For sparse spreadsheets often found in practice (takeaway 1 and 2), this model is quite efficient in storage and access since it records only the filled in cells, but for dense spreadsheets, it incurs the additional cost of recording and retrieving the row and column numbers for each cell as compared to ROM and COM, and has a much larger number of tuples. RCV is also efficient when it comes to retrieving specific cells at a time.

Table-Oriented Model (TOM). Spreadsheet regions linked via our linkTable operation, which sets up a two-way synchronization between the spreadsheet interface and the back-end database, are stored as *native tables* in the database. The schema of such tables is defined on the spreadsheet interface. We refer to this representation as Table-Oriented Model.

3.4 PRIMITIVE DATA MODELS: OPTIMALITY

Readers may be wondering why we chose these data models (ROM, COM and RCV). As it turns out, these three data models represent extremes in a space of data models that we identify and refer to as *rectangular data models*. We can further demonstrate that these three models do not dominate each other, *i.e.*, there are settings where each of them prevails and are *optimal* within the space of rectangular data models.

Characteristics. We require each primitive data model in our class to have the following characteristics: *(i) The data model should correspond to storing a rectangular region in the spreadsheet.* This constraint naturally stems from the way we perceive tables in a two-dimensional interface, in the sense that tables are rectangular, and our data models are stored as rectangular tables on disk. *(ii) The tuples in each table should correspond to a uniform geometric structure, and be contiguous in the sheet.* The first part of the constraint arises because we store our tables in a relational database, necessitating all tuples to have the same number of attributes. Additionally, we want our tuples to correspond to contiguous regions in the spreadsheet, *i.e.*, they should not have any "holes" in them.

Rectangular and Non-Rectangular Data Models. The data models which satisfy the aforementioned requirements fall into the following two classes: *(i) Rectangular.* In rectangular data models, each tuple corresponds to a rectangle in the sheet. Clearly, they are uniform geometric units, and are contiguous. A typical example is provided in Figure 3.8(a). *(ii) Non-rectangular.* Non-rectangular data models are essentially data-models where each tuple does not correspond to a rectangle. For instance, each tuple can either be diagonal with a fixed length, or have a "zig-zag" shape. A typical example where each tuple has zig-zag shape is provided in Figure 3.8(b).

Updates as Optimality Criterion. We now discuss our optimality criterion. Since we

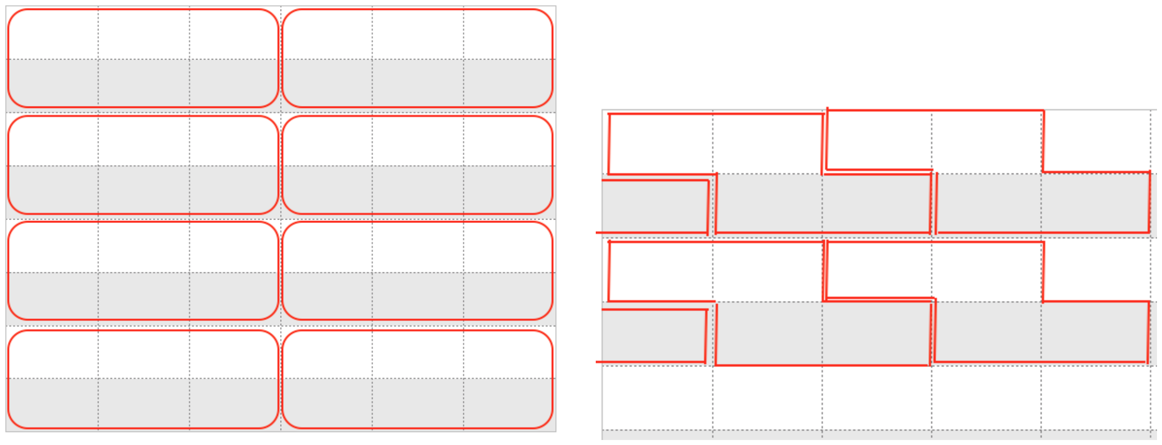


Figure 3.8: Data Model: (a) Rectangular (b) Non-rectangular

consider a single table, storage is not a concern since every data model has to store all of the data in a table. Furthermore, with any vanilla index, *e.g.*, B+ tree, access can be supported in all models in a similar manner, and likewise for single cells updates. Hence, we focus on updates on the sheet, and how they correspond to reorganizations in backend. Specifically, we focus on row/column inserts/deletes since changing values of existing data in the sheet would result in the same time complexity across all data models.

As we shall soon describe, row/column inserts/deletes can greatly influence the performance of our data models.

Theorem 3.1 (Optimality)**.** *Our primitive data models, coupled with our hierarchical positional mapping schemes, are the* only *models which do not result in cascading updates from the class of data models discussed above.*

Proof. Consider any data model which can be rectangular or otherwise. We know all tuples are uniform in shape, and are contiguous in the sheet. Let say the tuple spans p row and q columns.

There are two possibilities: these tuples are either stored in row major form in the table or in column major form. If we use the former, then a row insert would result in data from p rows to be *shifted* in the worst case. Equivalently, if the data is stored in column major form, then a column insert would result in data from q columns to be shifted in the worst case. Therefore, cascading updates can be avoided only when one among p and q equals 1. There are three cases now:

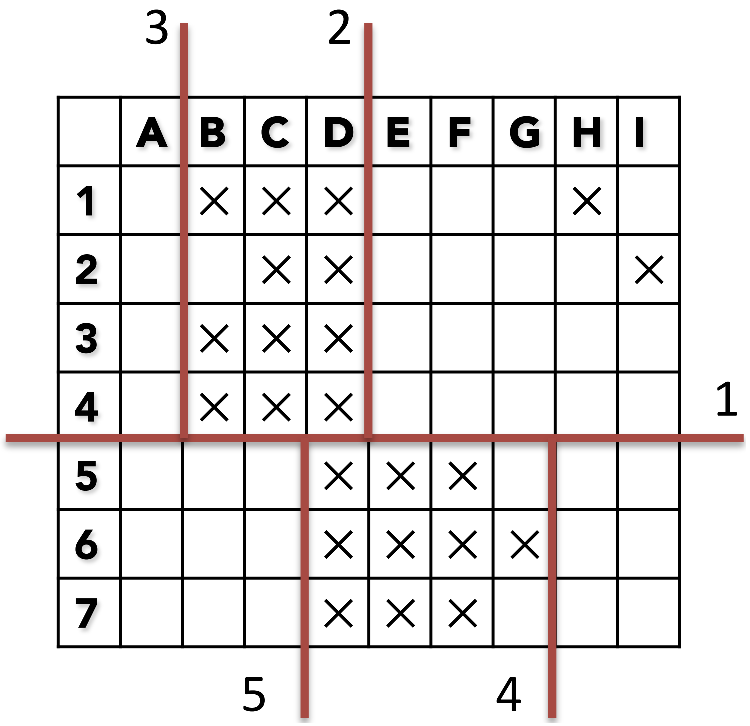


Figure 3.9: Hybrid Data Model: Recursive Decomposition.

1. $p = 1, q \neq 1$: This corresponds to ROM.

2. $p \neq 1, q = 1$: This corresponds to COM.

3. $p = 1, q = 1$: This corresponds to RCV.

This completes our proof.

3.4.1 Hybrid Data Model: Intractability

So far, we developed the primitive data models to represent a spreadsheet using a single table in a database. We now develop better solutions by decomposing a spreadsheet into multiple regions, each represented by one of the primitive data models. We call these *hybrid data models.*

Definition 3.1 (Hybrid Data Models)**.** *Given a collection of cells C, we define* hybrid data models *as the space of physical data models that are formed using a collection of tables T such that T is recoverable with respect to C, and further, each $T_i \in T$ is either a ROM, COM, RCV, or a TOM table.*

As an example, for the spreadsheet in Figure 3.9, we might want the dense areas, *i.e.*, B1:D4 and D5:G7, represented via a ROM table each and the remaining area, specifically, H1 and I2 to be represented by an RCV table.

Cost Model. As discussed earlier, the storage, access, and update costs impact our choice of data model. We now focus on storage. We will generalize to access cost in Section 3.4.4. The update cost will be the focus of Section 3.5. We begin with ROM; we will generalize to RCV and COM in Section 3.4.4.

Given a hybrid data model $T = \{T_1, \ldots, T_p\}$, where each ROM table T_i has r_i rows and c_i columns, the cost of T is

$$\text{cost}(T) = \sum_{i=1}^{p} s_1 + s_2 \cdot (r_i \times c_i) + s_3 \cdot c_i + s_4 \cdot r_i. \tag{3.1}$$

Here, the constant s_1 is the cost of initializing a new table, while the constant s_2 is the cost of storing each individual cell (empty or not) in the ROM table. The non-empty cells that have content require additional storage; however, this is a constant cost that does not depend on the data model. The constant s_3 is the cost corresponding to each column, while s_4 is the cost corresponding to each row. The former is necessary to record schema information per column, while the latter is necessary to record the row information in the $RowID$ attribute. Overall, while the specific costs s_i may differ across databases, what is clear is that all of these costs matter.

Formal Problem. We now state our formal problem below.

Problem 3.2 (Hybrid-ROM)**.** *Given a spreadsheet with a collection of cells C, identify the hybrid data model T with only ROM tables that minimizes* cost(T)*.*

Unfortunately, Problem 3.2 is NP-Hard, via a reduction from *minimum edge length partitioning of rectilinear polygons* [34]; see below.

Theorem 3.2 (Intractability)**.** *Problem 3.2 is* NP-Hard*.*

We show that Problem 3.2 is NP-Hard; for the decision version of the problem, a value k is provided, and the goal is to test if there is a hybrid data model with $\text{cost}(T) \leq k$.

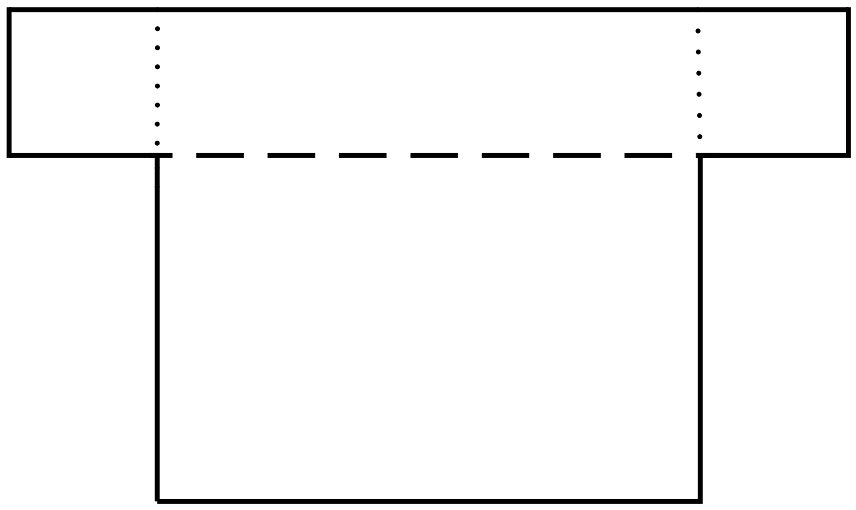


Figure 3.10: Minimum number of rectangles $(---)$ does not coincide with minimum edge length $(\cdots)$.

We use a reduction from the minimum edge length partitioning problem of rectilinear polygons [34]. A rectilinear polygon is one in which all edges are either aligned with the x-axis or the y-axis. The minimality criterion is the total length of the edges (lines) used to form the internal partition. Notice that this doesn't necessarily correspond to the minimality criterion of reducing the number of components. We illustrate this in Figure 3.10, which is borrowed from the original paper [34]. The following decision problem is shown to be NP-Hard in [34]: Given any rectilinear polygon P and a number k, is there a rectangular partitioning whose total edge length does not exceed k?

Proof. Consider an instance P of the polygon partitioning problem with minimum edge length required to be at most k. We now represent the polygon P in a spreadsheet by filling the cells interior of the polygon with arbitrary values, and not filling any other cell in the spreadsheet. Let $C = \{C_1, C_2, \ldots, C_m\}$ represent the set of all filled cells in the spreadsheet. We claim that a minimum edge length partition of the given rectilinear polygon P of length at most k exists iff there is a solution for the following setting of the optimal hybrid data model problem: $s_1 = 0, s_2 = 2|C| + 1, s_3 = s_4 = 1$, where the storage cost should not exceed $k' = k + \frac{\text{Perimeter}(P)}{2} + (2|C| + 1)|C|$ for some decomposition of the spreadsheet.

$\Rightarrow$ Say the spreadsheet we generate using P has a decomposition of rectangles whose storage cost is less than $k' = k + \frac{\text{Perimeter}(P)}{2} + s_2|C|$. We have to show that there exists a partition with minimum edge length of at most k.

First, notice that there exists a valid decomposition that doesn't store any blank cell. Say there is a decomposition that stores a blank cell. Since we are now storing $|C|+1$ cells at minimum, $k' > s_2(|C|+1) = |C|s_2+s_2 = |C|s_2+2|C|+1$ and thus $k' > |C|(s_2+1+1)$, which is the cost of storing each cell in a separate table. Therefore, if we have a decomposition that stores a blank cell, we also have a decomposition that does not store any blank cell and has lower cost. Second, there exists a decomposition of the spreadsheet where all the tables are disjoint. The argument is similar to the previous case since storing the same cell twice in different tables is equivalent to storing an extra blank cell.

From our above two observations, we conclude that there exists a decomposition where all tables are disjoint, and no table stores a blank cell. Therefore, this decomposition corresponds to *partitioning* the given spreadsheet into rectangles. We represent this partition of the spreadsheet by $T = \{T_1, T_2, \ldots, T_p\}$. We now show that this partition of the spreadsheet corresponds to a partitioning of the rectilinear polygon P with edge-length less than k. On setting $s_1 = 0, s_2 = 2|C| + 1, s_3 = s_4 = 1$, we get:

$$\text{cost}(T) = \sum_{i=1}^{p} 0 + s_2|C| + 1 \cdot \left(\sum_{i=1}^{p} c_i + \sum_{i=1}^{p} r_i \right) \tag{3.2}$$

since $\text{cost}(T) \leq k' = k + \frac{\text{Perimeter}(P)}{2} + s_2|C|$,

$$\sum_{i=1}^{p} (r_i + c_i) \leq k + \frac{\text{Perimeter}(P)}{2} \tag{3.3}$$

$$\implies \sum_{i=1}^{p} \text{Perimeter}(T_i) \leq 2 \times k + \text{Perimeter}(P) \tag{3.4}$$

Since the sum of perimeters of all the tables T_i counts the boundary of P exactly once, and the edge length partition of P exactly twice, the partition of the spreadsheet $T = \{T_1, T_2, \ldots, T_p\}$ corresponds to an edge-length partitioning of the given rectilinear polygon P with edge-length less than k.

$\Leftarrow$ Let us assume that the given rectilinear polygon P has a minimum edge length partition of length at most k. We have to show that there exists a decomposition of the spreadsheet whose storage cost is at most $k' = k + \frac{\text{Perimeter}(P)}{2} + s_2|C|$. Let us represent the set of rectangles that corresponds to an edge length partition of P of at most k as $T = \{T_1, T_2, \ldots, T_p\}$. We shall use the partition T of P as the decomposition of the spreadsheet itself:

$$\text{cost}(T) = \sum_{i=1}^{p} s_1 + s_2 \cdot (r_i \times c_i) + s_3 \cdot c_i + s_4 \cdot r_i \tag{3.5}$$

$$\text{cost}(T) = \sum_{i=1}^{p} s_1 + s_2 \sum_{i=1}^{p} \cdot (r_i \times c_i) + s_3 \sum_{i=1}^{p} c_i + s_4 \sum_{i=1}^{p} r_i \tag{3.6}$$

substituting $s_1 = 0, s_2 = 2|C| + 1, s_3 = s_4 = 1$, we get:

$$\text{cost}(T) = \sum_{i=1}^{p} 0 + s_2|C| + 1 \cdot \left(\sum_{i=1}^{p} c_i + \sum_{i=1}^{p} r_i \right) \tag{3.7}$$

$$\text{cost}(T) = s_2|C| + \sum_{i=1}^{p} (r_i + c_i) \tag{3.8}$$

$$\text{cost}(T) = s_2|C| + \sum_{i=1}^{p} \frac{\text{Perimeter}(T_i)}{2} \tag{3.9}$$

since $\sum_{i=1}^{p} \text{Perimeter}(T_i) = 2 \times k + \text{Perimeter}(P)$, we have:

$$\text{cost}(T) = s_2|C| + k + \frac{\text{Perimeter}(P)}{2} = k' \tag{3.10}$$

$$\implies \text{cost}(T) = k' \tag{3.11}$$

Therefore, the decomposition of the spreadsheet using T corresponds to a decomposition whose storage cost equals k'. Note that our reduction can be done in polynomial time. Therefore we can solve the minimum length partitioning problem in polynomial time, if we have a polynomial time solution to the optimal storage problem. However, since the minimum length partitioning problem is NP-Hard [34], the optimal hybrid data model problem is NP-Hard. This completes our proof.

3.4.2 Optimal Recursive Decomposition

Instead of directly solving Problem 1, which is intractable, we instead aim to make it tractable, by reducing the search space of solutions. In particular, we focus on hybrid data models that can be obtained by *recursive decomposition.* Recursive decomposition is a process where we repeatedly subdivide the spreadsheet area from $[1 \ldots r_{max}, 1 \ldots c_{max}]$ by using a vertical cut between two columns or a horizontal cut between two rows, and then recurse on the resulting areas. For example, in Figure 3.9, we cut along line 1 horizontally, giving us two regions from rows 1 to 4 and rows 5 to 6. We then cut the top portion along line 2 vertically, followed by line 3, separating out one table B1:D4. By cutting the bottom portion along line 4 and line 5, we can separate out the table D5:G7. Further cuts can help us carve out tables out of H1 or I2, not depicted here.

As the example illustrates, recursive decomposition captures a broad exponential space of hybrid models; basically, anything that can be obtained via recursive cuts along the x and y axis. Unfortunately, there is an exponential number of such models. Now, a natural question is: what sorts of hybrid data models cannot be composed via recursive decomposition?

Observation 3.1 (Counterexample)**.** *In Figure 3.11(a), the tables: A1:B4, D1:I2, A6:F7, and H4:I7 cannot be obtained via recursive decomposition.*

To see this, note that any vertical or horizontal cut that one would make at the start would cut through one of the four tables, making the decomposition impossible. Nevertheless, we expect this form of construction to not be frequent, whereby the hybrid data models obtained via recursive decomposition form a natural class of data models.

Despite the space of recursively decomposed hybrid data models being exponential, as it turns out, identifying the optimal data model in this space to Problem 3.2 is PTIME using dynamic programming. Our algorithm makes the most optimal "cut" horizontally or vertically at every step, and proceeds recursively; details below.

Consider a rectangular area formed from x_1 to x_2 as the top and bottom row numbers respectively, both inclusive, and from y_1 to y_2 as the left and right column numbers respectively, both inclusive, for some x_1, x_2, y_1, y_2. Now, the optimal cost of representing this rectangular area, which we represent as $\text{Opt}((x_1, y_1), (x_2, y_2))$ is the minimum of the

	A	B	C	D	E	F	G	H	I
1	×	×		×	×	×	×	×	×
2	×	×		×	×	×	×	×	×
3	×	×							
4	×	×						×	×
5								×	×
6	×	×	×	×	×	×		×	×
7	×	×	×	×	×	×		×	×

		2	1	3	1	2
		A	C	D	G	H
2	1	×		×	×	×
1	3	×				
1	4	×				×
1	5					×
2	6	×	×	×		×

Figure 3.11: (a) Counterexample. (b) Weighted Representation.

following possibilities:

- If there is no filled cell in the area $(x_1, y_1), (x_2, y_2)$, then we do not use any data model, and the cost is 0.
- Do not split, *i.e.*, store as a ROM model (romCost()):

$$\text{romCost}((x_1, y_1), (x_2, y_2)) = s_1 + s_2 \cdot (r_{12} \times c_{12}) + s_3 \cdot c_{12} + s_4 \cdot r_{12}, \tag{3.12}$$

  where number of rows $r_{12} = (x_2 - x_1 + 1)$, and the number of columns $c_{12} = (y_2 - y_1 + 1)$.
- Perform a horizontal cut (C_H):

$$C_H = \min_{i \in \{x_1, \ldots, x_2\}} \text{Opt}((x_1, y_1), (i, y_2)) + \text{Opt}((i+1, y_1), (x_2, y_2)) \tag{3.13}$$

- Perform a vertical cut (C_V):

$$C_V = \min_{j \in \{y_1, \ldots, y_2\}} \text{Opt}((x_1, y_1), (x_2, j)) + \text{Opt}((x_1, j+1), (x_2, y_2)) \tag{3.14}$$

Therefore, when there are filled cells in the rectangle,

$$\text{Opt}((x_1, y_1), (x_2, y_2)) = \min\big(\text{romCost}((x_1, y_1), (x_2, y_2)), C_H, C_V\big). \tag{3.15}$$

The base case is when the rectangular area is of dimension 1×1. Here, we store the area as a ROM table if it is filled. That is, $\text{Opt}((x_1, y_1), (x_1, y_1)) = c_1 + c_2 + c_3 + c_4$, if filled, and 0 if not.

We have the following theorem:

Theorem 3.3 (Dynamic Programming Optimality). *For the exponential space of ROM-based hybrid data models based on recursive decomposition, we can obtain the optimal solution via dynamic programming in* PTIME.

Time Complexity. Our dynamic programming algorithm runs in polynomial time with respect to the size of the spreadsheet. Let the length of the larger side of the minimum enclosing rectangle of the spreadsheet be of size n. Then, the number of candidate rectangles is $\mathcal{O}(n^4)$. For each rectangle, we have $\mathcal{O}(n)$ ways to perform the cut. Therefore, the running time of our algorithm is $\mathcal{O}(n^5)$. However, this number could be very large if the spreadsheet is massive–which is typical of the use-cases we aim to tackle.

Approximation Bound. Even though our dynamic programming algorithm only identifies the best recursive decomposition based hybrid data model, we can derive a bound for its cost relative to the best hybrid data model overall.

Theorem 3.4 (Approximation Bound). *Say there are k rectangles in the optimal decomposition with cost c. Then, the recursive decomposition algorithm identifies a decomposition with cost at most $c+s_1\times\frac{k(k-1)}{2}$, where s_1 is the cost of storing a new table as in Equation 3.1.*

Proof. Let the optimal decomposition consist of a set of five rectangles R as in Figure 3.12. Starting from R, we will construct a recursive decomposition solution with cost $c+s_1\times\frac{k(k-1)}{2}$, denoted as R', using the following steps. Sort the rectangles from the optimal solution in the increasing order of their bottom edge. Pick the first rectangle, and use a line through its bottom edge to cut or partition the remaining rectangles. This is the first "partitioning" step, denoted as 1 in Figure 3.12. This partitioning step leads to two portions. We handle the top portion with vertical partitions, while for the bottom portion we recurse. This partition introduces at most $k-1$ new rectangles in the top half and eliminates one rectangle.

Thus, at every step, we have $k-1$ new rectangles and reduce the total number of rectangles by 1. That is, the next partition will introduce at most $k-2$ rectangles; and so on. So, we in total we $(k-1)+(k-2)+\ldots+1=\frac{k(k-1)}{2}$ new rectangles. Since the dynamic programming algorithm explores the entire space of recursive decomposition based

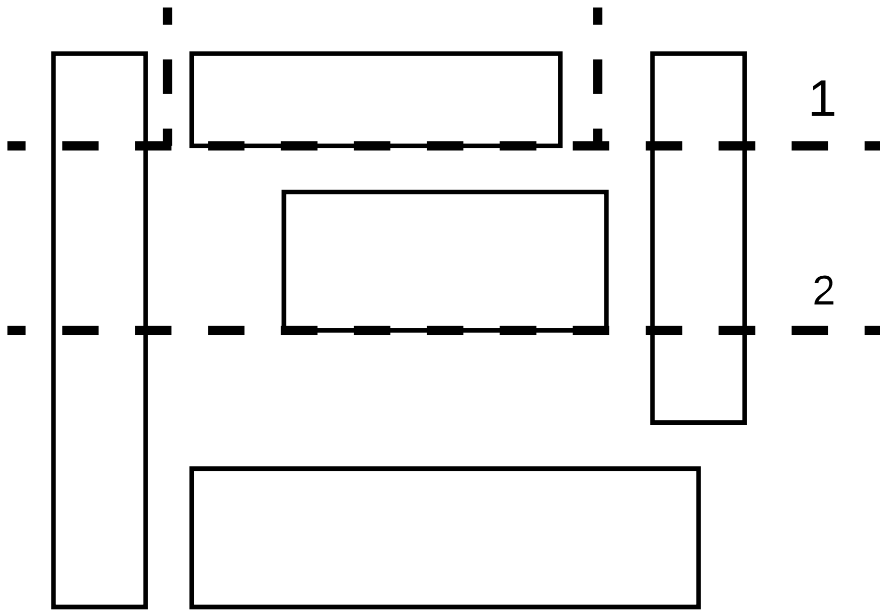


Figure 3.12: Obtaining a recursive decomposition from the optimal solution.

data models, it also considers R' as one of the candidates. Thus; its solution must be at least as good. Hence proved.

Typically k is small, so this is a small additive approximation. In our experiments, we compare the solution obtained from recursive decomposition with a lower bound of the optimal solution (Section 3.7.2), and show that it is near-optimal. As observed from Figures 3.2 and 3.3, typically spreadsheets have a small number of highly dense connected components. By deriving an upper bound for the number of tables in the optimal solution for each connected component, we can get an upper bound for k. The following theorem implies that the high density of the connected components makes it sub-optimal to split them in the optimal decomposition, thereby suggesting that the number of tables in the optimal decomposition, *i.e.*, k, is small.

Theorem 3.5 (Connected Component Solution)**.** *The optimal solution to Problem 3.2 for a minimum bounding rectangle of a connected component will have at most* $\left\lfloor \frac{e \times s_2}{s_1} + 1 \right\rfloor$ *rectangles, where* e *is the number of empty cells in the bounding rectangle.*

Proof. Let the optimal decomposition for a minimum bounding rectangle of a connected component C have k' tables. Therefore, we have the cost representing the minimum bounding rectangle using a single table is higher than the optimal decomposition into k' tables, *i.e.*,

$$\sum_{i=1}^{k'} s_1 + s_2 \cdot (r_i \times c_i) + s_3 \cdot c_i + s_4 \cdot r_i \leq s_1 + s_2 \cdot (r_0 \times c_0) + s_3 \cdot c_0 + s_4 \cdot r_0, \tag{3.16}$$

where $r_1, .., r_{k'}$ and $c_1, .., c_{k'}$ are the number of rows and columns respectively for each of the tables in the optimal decomposition and r_0 and c_0 are number of rows and columns for the minimum bounding rectangle.

Since our region of focus is a minimum bounding rectangle encapsulating a connected component, we do not have any empty rows or columns. Thus, each row and column should be captured by at least one rectangle in the optimal decomposition. Hence, we have,

$$\sum_{i=1}^{k} s_3 \cdot c_i + s_4 \cdot r_i \geq s_3 \cdot c_0 + s_4 \cdot r_0 \tag{3.17}$$

Subtracting Equation 3.17 from 3.16 we have,

$$\sum_{i=1}^{k} s_1 + s_2 \cdot (r_i \times c_i) \leq s_1 + s_2 \cdot (r_0 \times c_0). \tag{3.18}$$

Since the optimal solution should represent all the filled-in cells at least once, we have $\sum_{i=1}^{k}(r_i \times c_i) \geq r_0 \times c_0 - e$, where e is the number of empty cells in the bounding box. Subtracting this from Equation 3.18 and simplifying, we get

$$k \cdot s_1 \leq s_1 + e \times s_2. \tag{3.19}$$

$$k \leq 1 + \frac{e \times s_2}{s_1}. \tag{3.20}$$

Since k is an integer, we have

$$k \leq \left\lfloor 1 + \frac{e \times s_2}{s_1} \right\rfloor. \tag{3.21}$$

Hence proved.

We empirically show in Section 3.7.2 that the upper bound of k is small by obtaining the distribution of $\sum \left\lfloor \frac{e \times s_2}{s_1} + 1 \right\rfloor$ across our datasets. Additionally, we compare the solution obtained from our recursive decomposition with a lower bound of the optimal solution (Section 3.7.2), and demonstrate that it is in fact, near-optimal.

Weighted Representation. Notice that in many real spreadsheets, there are many rows and columns that are very similar to each other in structure, *i.e.*, they have the same set of filled cells. We exploit this property to reduce the effective size n of the spreadsheet. Essentially, we collapse rows that have identical structure down to a single weighted row, and similarly collapse columns. Consider Figure 3.11(b) which shows the weighted version of Figure 3.11(a). Here, we can collapse column B down into column A, which is now associated with weight 2; similarly, we can collapse row 2 into row 1, which is now associated with weight 2. The effective area of the spreadsheet now becomes 5×5 as opposed to 7×9. Now, we apply the same dynamic programming algorithm to the weighted representation: in essence, we are avoiding making cuts "in-between" the weighted edges, thereby reducing the search space. This does not sacrifice optimality.

Theorem 3.6 (Weighted Optimality)**.** *The optimal hybrid data model obtained by recursive decomposition on the weighted spreadsheet is no worse than the optimal hybrid data model obtained by recursive decomposition on the original spreadsheet.*

3.4.3 Greedy Decomposition Algorithms

Greedy Decomposition. To improve the running time even further, we propose a greedy heuristic that avoids the high complexity but sacrifices somewhat on optimality. The greedy algorithm essentially repeatedly splits the spreadsheet area in a top-down manner identifying the operation that results in the lowest local cost. We have three alternatives for an area $(x_1, y_1), (x_2, y_2)$: Either we do not split, with cost from Equation 3.12, *i.e.*, romCost$((x_1, y_1), (x_2, y_2))$. Or we split horizontally (vertically), with cost C_H (C_V) from Equation 3.13 (Equation 3.14), but with Opt() replaced with romCost(), since we are making a locally optimal decision. The smallest cost decision is followed, and then we continue recursively decomposing using the same rule on the new areas, if any.

Complexity. This algorithm has a complexity of $\mathcal{O}(n^2)$, since each step takes $\mathcal{O}(n)$ and there are $\mathcal{O}(n)$ steps. While the greedy algorithm is sub-optimal, its local decision is *optimal assuming the worst case about the decomposed areas*, *i.e.*, with no further information about the decomposed areas this is the best decision to make at each step.

Aggressive Greedy Decomposition. Since it is based on the worst case, the greedy algorithm may halt prematurely, even though further decompositions may have helped to reduce cost. An alternative, with the same complexity as greedy, is one where we don't stop subdividing, *i.e.*, we always choose to use the best horizontal or vertical cut, until we end up with rectangular areas where all of the cells are non-empty. After this point, we backtrack up the tree of decompositions, assembling the best solution that was discovered, considering whether to not split, or perform a horizontal or vertical split.

Complexity. The aggressive greedy approach also has complexity $\mathcal{O}(n^2)$, but takes longer since it considers a larger space of data models than the greedy approach.

3.4.4 Extensions: Maintenance, Cost, Models

In this section, we discuss a number of extensions to the cost model of the hybrid data model. We will describe these extensions to the cost model, and then describe the changes to the basic dynamic programming algorithm; modifications to the greedy and aggressive greedy decomposition algorithms are straightforward.

RCV, COM and TOM. The cost model can be extended in a straightforward manner to allow each rectangular area to be a ROM, COM, or an RCV table. (We deal with the TOM case later.) First, note that it doesn't benefit us to have multiple RCV tables—we can simply combine all of these tables into one, and assume that we're paying a fixed up-front cost to have one RCV table. Then, the cost for a table T_i, if it is stored as a COM table is:

$$\text{comCost}(T_i) = s_1 + s_2 \cdot (r_i \times c_i) + s_4 \cdot c_i + s_3 \cdot r_i.$$

This equation is the same as Equation 3.1, but with the last two constants transposed. And the cost for a table T_i, if it is stored as an RCV table is simply:

$$\text{rcvCost}(T_i) = s_5 \times \#\text{cells}.$$

where s_5 is the cost incurred per tuple. Once we have this cost model set up, it is straightforward to apply dynamic programming once again to identify the optimal hybrid data model encompassing ROM, COM, and RCV. The only step that changes in the dynamic programming equations is Equation 3.12, where we have to consider the COM and RCV alternatives in addition to ROM. To handle TOM tables, we assume that the corresponding cells are empty; while also setting the romCost() and comCost() of any tables overlapping with these cells as ∞. We have the following theorem.

Theorem 3.7 (ROM, COM, TOM, and RCV)**.** *The optimal ROM, COM, TOM, and RCV-based hybrid data model based on recursive decomposition can be determined in* PTIME*.*

Incremental Decomposition. So far, we have focused on finding an optimal decomposition given a static spreadsheet. We now consider how we can support incremental maintenance of the decomposition across updates. Here, along with the storage cost, we also consider the cost of migrating cells from an existing decomposition T_o to a new decomposition T. We define the migration cost as $\text{migCost}((x_1, y_1), (x_2, y_2)) = \#\text{cells}$, where #cells denotes the number of populated cells in the rectangular region defined by $(x_1, y_1), (x_2, y_2)$. To migrate a region of a spreadsheet into a new decomposition, we assume that we only use an existing table if it exactly covers the region; for all other cases we migrate all of the populated cells within the region to the new decomposition. In other words, we do not consider the cases when an existing table needs to be modified either to accommodate or eliminate rows or columns. We introduce a factor η to enable users to balance the trade-off between the migration cost and storage cost; our objective is thus find a data model T such that $\text{cost}(T) + \eta \cdot \text{migCost}()$ is minimized.

For incremental decomposition, we update the dynamic programming formulation by adding an additional case that retains the decomposition as is and updates the romCost() to include the migration cost in terms of the number of cells that need to be migrated from the existing model into a new model. As the migration cost for a region is defined as the number of populated cells, the migration cost of a region can be computed independently of the remaining regions. This enables us to employ dynamic programming once again.

- Keep the decomposition as-is. This is permissible only if the there exists a ROM model at $(x_1, y_1), (x_2, y_2)$ in T_o.

$$\text{romCost}((x_1, y_1), (x_2, y_2)) = s_1 + s_2 \cdot (r_{12} \times c_{12}) + s_3 \cdot c_{12} + \\ s_4 \cdot r_{12}, \quad (3.22)$$

- Store the area as ROM by migrating the non-empty cells into the new model.

$$\text{romCost}((x_1, y_1), (x_2, y_2)) = s_1 + s_2 \cdot (r_{12} \times c_{12}) + s_3 \cdot c_{12} + s_4 \cdot r_{12} + \\ \eta \cdot \text{migCost}((x_1, y_1), (x_2, y_2)) \quad (3.23)$$

Note that we may be able to migrate a region more efficiently by leveraging existing tables that partially cover the region; however, this will lead to complications in leveraging tables that span more than one region, and any reorganization costs involved. For simplicity, we do not consider such migrations. As we will demonstrate in Section 3.7, this still leads to adequate performance.

Access Cost. So far, we have only been focusing on storage. Our cost model can be extended in a straightforward manner to handle access cost—both scrolling-based operations, and formulae, and our dynamic programming algorithms can similarly be extended to handle access cost without any substantial changes. We focus on formulae since they are often the more substantial cost of the two; scrolling-based operations can be similarly handled. For formulae, there are multiple aspects that contribute to the time for access: the number of tables accessed, and within each table, since data is retrieved at a tuple level, the number of tuples that need to be accessed, and the size of these tuples. Once again, each of these aspects can be captured within the cost model via constants similar to $s_1, \ldots, s_5$, and can be seamlessly incorporated into the dynamic programming algorithm. Thus, we have:

Theorem 3.8 (Optimality with Access Cost). *The optimal ROM, COM, and RCV-based hybrid data model based on recursive decomposition, across both storage and access cost, can be determined via dynamic programming.*

Even though in this thesis we focus on relational row stores for hybrid data models, our techniques are general—making as few assumptions on the underlying physical storage as possible. Thus, our techniques can be easily extended to work with all kind of data stores, including column stores and key-value stores.

Size Limitations of Present Databases. Current databases impose limitations on the number of columns within a relation[1]; since spreadsheets often have an arbitrarily large number of rows and columns (sometimes 10s of thousands each), we need to be careful when trying to capture a spreadsheet area within a collection of tables that are represented in a database.

This is relatively straightforward to capture in our context: in the case where we don't split (Equation 3.12), if the number of columns is too large to be acceptable, we simply return ∞ as the cost.

Theorem 3.9 (Optimality with Size Constraints). *The storage optimal ROM, COM, and RCV-based hybrid data model, with the constraint that no tables violate size constraints, based on recursive decomposition, can be determined via dynamic programming.*

Incorporating the Costs of Indexes. Within our cost model, it is straightforward to incorporate the costs associated with storage of indexes, since the size of the indexes are typically proportional to the number of tuples for a given table, and the cost of instantiating an index is another fixed constant cost. Since our cost model is general, by suitably re-weighting one or more of s_1, s_2, s_3, s_4, we can capture this aspect within our cost model, and apply the same dynamic programming algorithm.

Theorem 3.10 (Optimality with Indexes). *The storage optimal ROM-based hybrid data model, with the costs of indexes included, based on recursive decomposition, can be determined via dynamic programming.*

[1]Oracle column number limitations: https://docs.oracle.com/cd/B19306_01/server.102/b14237/limits003.htm\#i288032; MySQL column limitations: https://dev.mysql.com/doc/mysql-reslimits-excerpt/5.5/en/column-count-limit.html; PostgreSQL column limitations: https://www.postgresql.org/about/

3.5 PRESENTATIONAL ACCESS FOR UPDATES

For all of our data models, storing the row and/or column numbers may result in substantial overheads due to cascading updates—this makes working with large spreadsheets infeasible. To eliminate the overhead of cascading updates, we introduce *positional mapping*. For our discussion we focus on row numbers; the techniques can be analogously applied to columns—we use the term *position* to refer to this number. In addition, row and column numbers can be dealt with independently.

Problem. We require a data structure to capture a specific ordering among the items (here, tuples) and efficiently support: *(i) fetch* items based on a position, *(ii) insert* items at a position, and *(iii) delete* items from a position. The insert and delete operations require updating the positions of the subsequent items. For example, inserting an item at the n^{th} position requires us to first increment by one the positions of all the items that have a position greater than or equal to n, and then add the new item at the n^{th} position. Due to the interactive nature of DataSpread, our goal is to perform these operations within a few hundred milliseconds.

Operation	RCV	ROM
Insert	87,821 ms	1,531 ms
Fetch	312 ms	244 ms

Table 3.2: The performance of storing Position-as-is.

Naïve Solution: Position as-is. The simplest approach is to store the position along with each tuple: this makes fetch efficient at the expense of insert/delete operations. With a traditional index, *e.g.*, a B+ tree, the complexity to access an arbitrary row identified by a position is $\mathcal{O}(\log N)$. Insert and delete operations require updating the positions of subsequent tuples, which need to be propagated in the index, and therefore it results in a worst case complexity of $\mathcal{O}(N \log N)$. To illustrate the impact of these complexities in practice, in Table 3.2, we display the performance of storing the positions as-is for two operations—fetch and insert—on a spreadsheet containing 10^6 cells. We note that irrespective of the data model used, the performance of inserts is beyond our acceptable threshold whereas that of the fetch operation is acceptable.

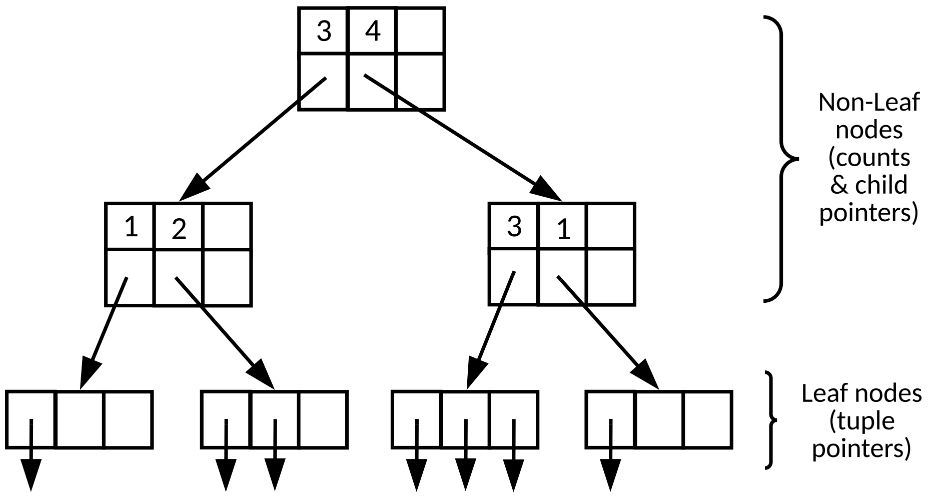

3
4
1
2
3
1
Non-Leaf nodes (counts & child pointers)
Leaf nodes (tuple pointers)

Figure 3.13: Hierarchical Positional Mapping.

Hierarchical Positional Mapping. To improve the performance of inserts and deletes for ordered items, we introduce the idea of *positional mapping.* At its core, the idea is simple: we do not store positions explicitly but instead obtain them on the fly. Formally, positional mapping $\mathbb{M}$ is a bijective function that maintains the relationship between the position r and tuple pointers p, *i.e.*, $\mathbb{M}(r) \rightarrow p$.

We now describe *hierarchical positional mapping*, which is an indexing structure that adapts classical work on *order-statistic trees* [35]. Just like a typical B+ tree is used to capture the mapping from keys to records, we can use the same structure to map positions to tuple pointers. Here, instead of storing a key we store the count of elements stored within the entire sub-tree. The leaf nodes store tuple pointers, while the remaining store children pointers along with counts. We show an example hierarchical positional mapping structure in Figure 3.13. Similar to a B+ tree of order m, our structure satisfies the following invariants. *(i)* Every node has at most m children. *(ii)* Every non-leaf node (except root) as at-least $\lceil {}^{m}/_{2} \rceil$ children. *(iii)* All leaf nodes appear at the same level. Again similar to B+ tree, we ensure the invariants by either splitting or merging nodes, ensuring that the height of the tree is at most $\log_{\lceil m/2 \rceil} N$.

Our hierarchical mapping structure makes accessing an item at the n^{th} position efficient, by starting from the root node with $n' = n$, and traversing downwards; at each node, given our current count n', we subtract the counts of as many of the children nodes from left-to-right (representing counts of sub-trees) as long as n' stays positive, and then follow the pointer to that child node, and repeat the process until we reach a leaf node and access a pointer to a tuple. Overall, the complexity of this operation is $\mathcal{O}(\log N)$.

Insert/delete are similar, where we start at the appropriate leaf node (as before), insert or delete appropriate tuple pointers, and then update the counts of all nodes on the path to the modified leaf node. Once again, the complexity of this operation is $\mathcal{O}(\log N)$.

Overall, we find that the complexity of the hierarchical positional mapping is $\mathcal{O}(\log N)$ for all operations, while the Position-as-is approach has $\mathcal{O}(\log N)$ for access, but $\mathcal{O}(N \log N)$ for insert/delete. We empirically evaluate the impact of the difference in complexities in Section 3.7.

3.6 DATASPREAD'S STORAGE ARCHITECTURE

To support interactive, scalable data access by integrating relational databases and spreadsheets, we have implemented a fully functional DataSpread prototype as a web-based tool using the open-source ZK Spreadsheet frontend [20] on top of a PostgreSQL database. Along with standard spreadsheet features, the prototype supports all the spreadsheet-like and database-like operations discussed in Section 3.2. Screenshots of DataSpread in action can be found in Section 3.7.4.

Figure 3.14 illustrates DataSpread's architecture, which at a high level can be divided into three layers, *(i)* user interface, *(ii)* execution engine, and *(iii)* storage engine. The *user interface* is a *spreadsheet widget*, which presents a spreadsheet on a web-based interface and handles the interactions on it. The *execution engine* is a Java web application residing on an application server. The *controller* accepts user interactions in the form of events and identifies the corresponding actions. For example, a formula update is sent to the formula parser and a cell update to the cell cache. The *positional mapper* translates the row and column numbers into the corresponding stored identifiers. The *ROM/TOM, COM, RCV, and hybrid translators* use their corresponding spreadsheet representations and provide a "collection of cells" abstraction to the upper layers. ROM/TOM, COM, and RCV translators service getCells by using the tuple pointers, obtained from the *positional mapper*, to

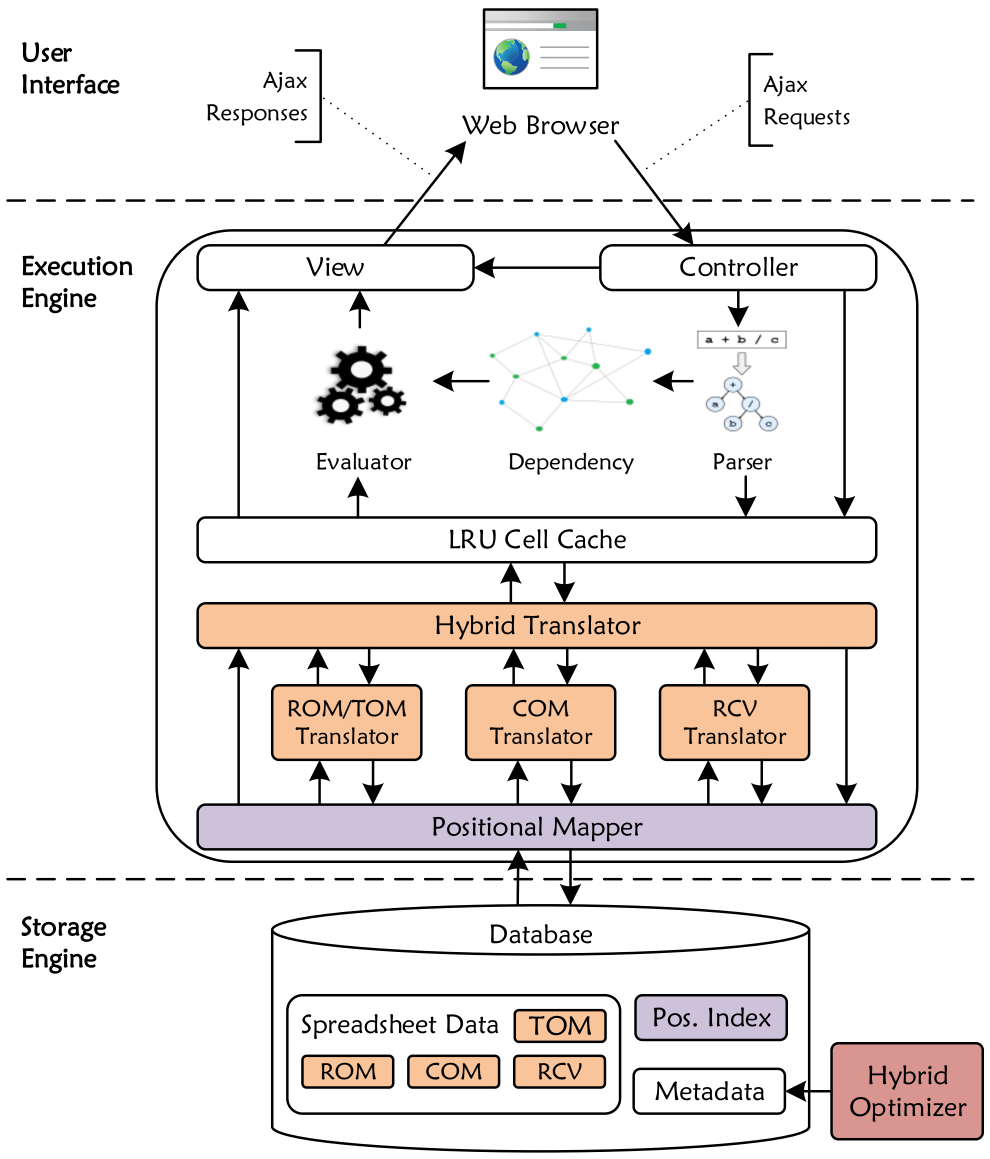


Figure 3.14: DataSpread's Storage Architecture.

fetch required tuples. For a hybrid model, the mapping from a range to model is stored as *metadata*. A region requested by the getCells operation on hybrid data model might span one or more primitive data models, in which case the hybrid translator delegates the call to all relevant primitive data models and aggregates their output. The returned cells are then cached in memory via the *LRU cell cache*. The storage engine is a relational database responsible for persisting data using a combination of *ROM, COM, RCV,* and *TOM* (Section 3.3) along with *positional mapping indexes*, which map row/column numbers to tuple pointers

(Section 3.5), and *metadata*, which records information about the hybrid data model. The *hybrid optimizer* determines the optimal hybrid data model and migrates data across data models.

Two-way synchronization. The two-way synchronization setup by the linkTable operation is captured as *metadata* in the database. The updates on the linked region are propagated to the underlying table in a write-through manner by the *controller*. The linkTable operation also creates a *database trigger* to monitor updates to the underlying linked table. Whenever the trigger fires, the *controller* invalidates the updated records from the cache; thereby signaling the user interface to fetch the updated cells from the underlying layers.

SQL and Relational Operations. In addition to supporting standard spreadsheet functions, DataSpread leverages the SQL engine of the underlying database to seamlessly supports SQL queries and relational operators on the front-end spreadsheet interface.

DataSpread supports executing of SQL queries via a spreadsheet function sql(query, [param1], ...), which takes a SQL statement along with parameters values as arguments. The query parameter is a single SQL SELECT statement, possibly containing '?'s. When one or more '?'s exists in the query, DataSpread treats the query like a SQL prepared statement, where each '?' is substituted by the values param1, ... in order. The number of parameters must match the number of '?'s in the query. Each parameter must evaluate to a single value, *i.e.*, it cannot refer to a range.

The sql function and the other functions that we discuss in this section return a single composite table value; to retrieve the individual rows and columns within that composite table value, we have an index(table, row, [column]) function that looks up the (row, column)th cell in the composite table value in location table, and places it in the current location.

In addition, DataSpread supports relational operators via the following spreadsheet functions: union(table1, table2), difference(table1, table2), intersection(table1, table2), crossproduct(table1, table2), select(table, filter), join(table1, table2, [filter]), project(table, attribute1, [attribute2], ...), and rename(table, oldAttribute, newAttribute). Since the input and output of all these functions is a table, they can be arbitrarily nested to obtain complex expressions.

The arguments table1 and table2 can either refer to a composite table value or a (contiguous) range of non-table values, which is treated as a table. The filter argument must be a Boolean expression which may utilize standard spreadsheet functions and can refer to attributes in tables.

3.7 EXPERIMENTAL EVALUATION

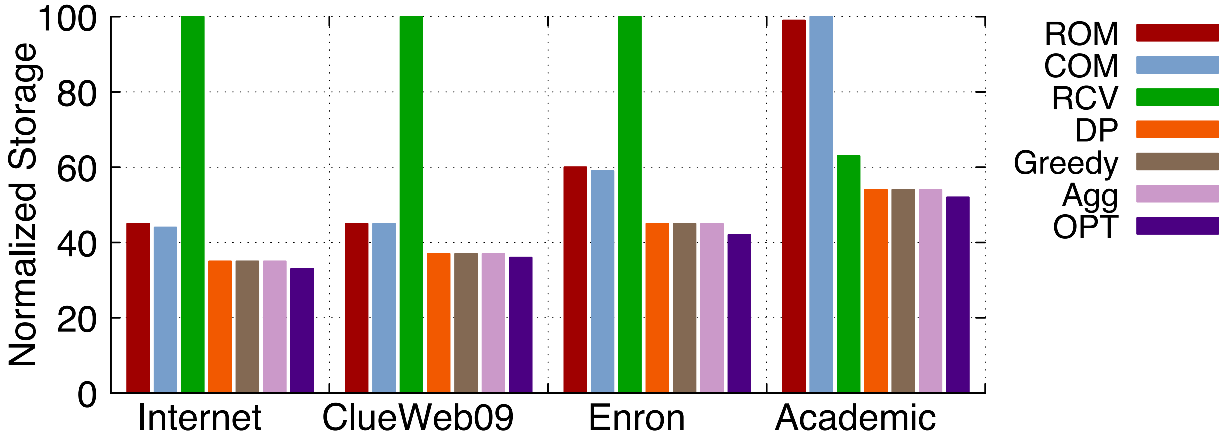
In this section, we present an evaluation of the storage engine of DataSpread.

3.7.1 Experimental Setup

Environment. We have implemented the data models and positional mapping techniques using PostgreSQL 9.6, configured with default parameters. We run all of our experiments on a workstation running Windows 10 on an Intel Core i7-4790K 4.0 GHz with 16 GB RAM. Our test scripts are single-threaded applications developed in Java. While we have a fully functional prototype, our test scripts are independent of it, so that we can isolate the back-end performance implications. We ensured fairness by clearing the appropriate cache(s) before every run.

Datasets. We evaluate our algorithms on a variety of real and synthetic datasets. Our real datasets are the ones listed in Table 3.1. To test scalability we constructed large synthetic spreadsheet datasets. We identify several goals for our experimental evaluation:

Goal 1: Presentational Awareness and Access on Real and Synthetic Datasets. We evaluate the hybrid data models selected by our algorithms against the primitive data models, when the cost model is optimized for storage. We compare our algorithms: *DP* (Section 3.4.2), and *Greedy* and *Agg* (greedy and aggressive-greedy from Section 3.4.3) against *ROM, COM, and RCV*, which represent our best current database approach. We evaluate these data models on both *storage*, as well as *formulae access cost*, based on the spreadsheet formulae. In addition, we evaluate the *running time* of the algorithms for of *DP, Greedy, and Agg.* We additionally evaluate hybrid data models optimized for formula accesses (Agg-formulae) and contrast it with storage-optimized ones. Finally, we drill-down into the performance of hybrid data models and investigate incremental maintenance of



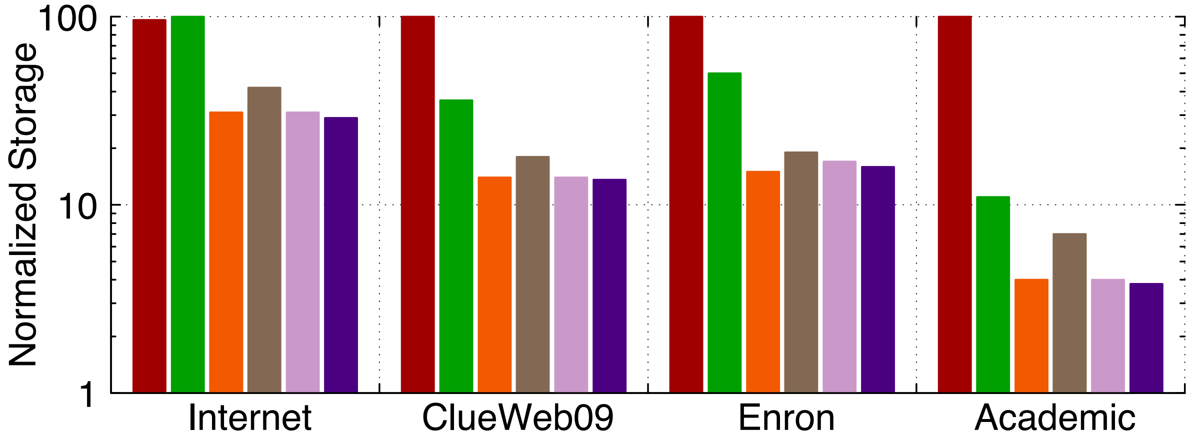


Figure 3.15: (a) Storage Comparison for PostgreSQL. (b) Storage Comparison on an Ideal Database.

hybrid data models.

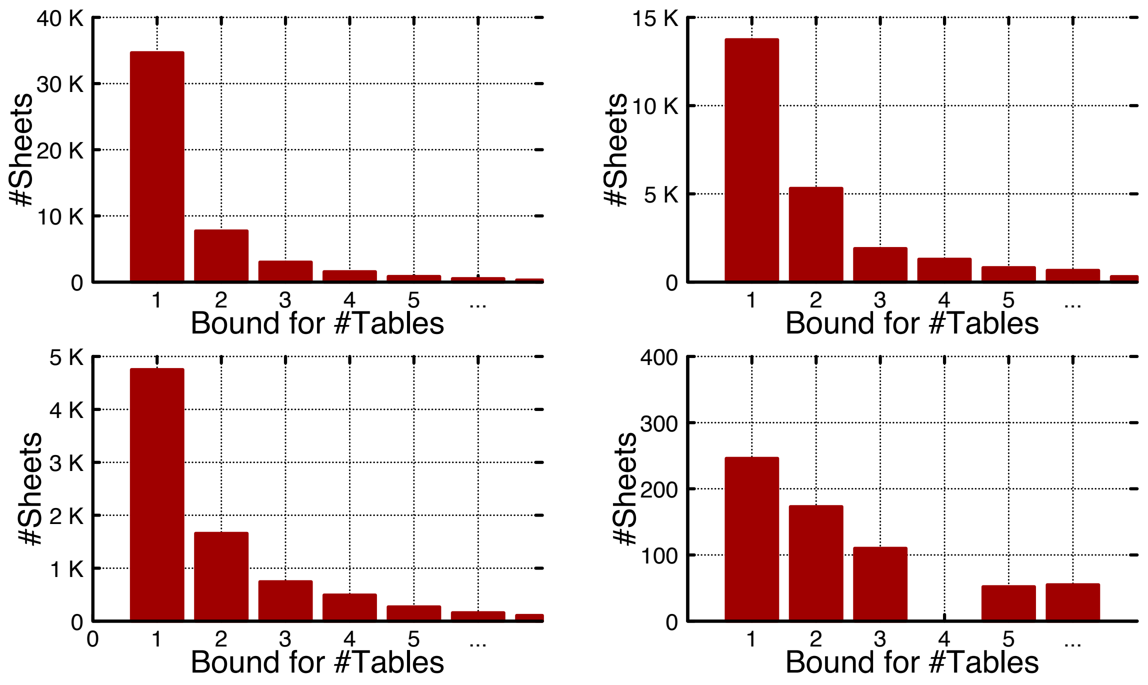


Figure 3.16: Upper bound for #Tables in the optimal decomposition—(a) Internet. (b) ClueWeb09. (c) Enron. (d) Academic.

Goal 2: Presentational Access (With Updates) on Synthetic Datasets. We evaluate the impact of our positional mapping schemes in aiding access on the spreadsheet. We focus on *Position-as-is, Monotonic, and Hierarchical* positional mapping schemes (introduced later) applied on the ROM primitive model, and evaluate the performance of *fetch, insert, and delete* operations on varying the *number of rows*. We additionally study the impact of varying parameters of synthetic spreadsheets on positional mapping.

Goal 3: Qualitative Evaluation. We evaluate the user experience of DataSpread relative to Excel, and study whether it's storage engine enables users to effectively work with large datasets in two different scenarios.

3.7.2 Presentational Awareness and Access

Takeaway 3.7. *Hybrid data models provide substantial benefits over primitive data models, with up to* ***20% reductions in storage, and up to 50% reduction in formula evaluation time*** *on PostgreSQL on real and synthetic spreadsheet datasets, compared to the best primitive data model. While DP has better performance on storage than Greedy and Agg, it* *suffers from high running time;* ***Agg bridges the gap between Greedy and DP****, while taking only marginally more running time than Greedy; both Agg and Greedy are within 10% of the optimal storage. Lastly, if we were to design a database storage engine from scratch, the hybrid data models would provide* ***up to 50% reductions in storage*** *compared to the best primitive data model. By optimizing for formula accesses, hybrid data models provide on average a* ***85% reduction in formula evaluation*** *time compared to the best primitive data model. Overall,* ***our hybrid data models bring scalability to spreadsheets: efficiently support storage across a range of spreadsheet structures, and access data via position in an efficient manner.***

The goal of this section is to evaluate spatial access and awareness (without updates) by evaluating our data models—on real and synthetic datasets.

a. Real Dataset: Storage Evaluation on PostgreSQL. We begin with an evaluation of storage for different data models on PostgreSQL. The schemas for the different models are as described in Section 3.3.2. The costs for storage on PostgreSQL as measured by us is as follows: s_1 is 8 KB, s_2 is 1 bit, s_3 is 40 bytes, s_4 is 50 bytes, and s_5 (RCV's tuple cost) is 52 bytes. We plot the results in Figure 3.15(a): here, we depict the average normalized storage across sheets; in addition to the aforementioned data models, we also plot a lower bound for the optimal hybrid data model (denoted OPT)—the cost of storing only non-empty cells in a single ROM, *i.e.*, the cost ignoring the overhead of extra tables and empty cells. For Internet, ClueWeb09, and Enron, we found RCV to have the worst performance, and hence normalized it to a cost of 100, and scaled the others accordingly; for Academic, we found COM to have the worst performance, and hence normalized it to a cost of 100, and scaled the others. The first three datasets are primarily used for data sharing, and as a result are quite *dense*. As a result, ROM and COM do well, using about 40% of the storage of RCV. At the same time, DP, Greedy and Agg perform roughly similarly, and better than the primitive data models, providing an additional reduction of 15–20%. On the other hand, since the last dataset (primarily used for computation) is very sparse, RCV does better than ROM and COM, while DP, Greedy, and Agg once again provide additional benefits. We finally observe that DP, Greedy, and Agg are all very close (within 10%) of OPT. From this we conclude that the solution give by Agg is close to the optimal in terms of cost.

We next show that the error bound of using a recursive decomposition based algorithms (DP, Greedy, and Agg) is small as compared to the optimal solution. For this we plot the upper bound for the number of tables in the optimal solution, *i.e.*, $\sum \left\lfloor \frac{e \times s_2}{s_1} + 1 \right\rfloor$, for the four data sets in Figure 3.16. Here, we observe the the number of tables in the optimal solution is typically small—90% of spreadsheets have fewer than 10 tables in the optimal decomposition. From the above observation and Theorem 3.4, we conclude that the error bound of using the search space of recursive decomposition for practical purposes is small.

b. Real Dataset: Storage Evaluation on an Ideal Database. The reason why RCV does so poorly for the first three datasets is because PostgreSQL imposes a high overhead per tuple, of 50 bytes, considerably larger than the amount of storage per cell. So, to explore this further, we investigated the scenario if we could redesign our storage engine from scratch. We consider a theoretical "ideal" cost model, where the cost of a ROM or COM table is equal to the number of cells, plus the length and breadth of the table (to store the data, the schema, as well as position), while the cost of an RCV row is simply 3 units (to store the data, as well as the row and column number). We plot the results in Figure 3.15(b) in log scale for each of the datasets—we exclude COM for this chart since it is identical to ROM. Here, we find that ROM has the worst cost since it no longer leverages benefits from minimizing the number of tuples. For Internet, ROM and RCV are similar, but RCV is slightly worse. As before, we normalize the cost of the worst model to 100 for each sheet, and scaled the others accordingly. As an example, we find that for the ClueWeb09 corpus, RCV, DP, Greedy and Agg have normalized costs of about 36, 14, 18, and 14 respectively—with the hybrid data models more than halving the cost of RCV, and getting $1/7^{th}$ the cost of ROM. Furthermore, DP provides additional benefits relative to Greedy, and Agg ends up bringing us close to DP performance; finally, we find that Agg and DP are both very close to OPT (within 10%).

c. Real Dataset: Running Time of Hybrid Optimization Algorithm. Our next question is how long our hybrid data model optimization algorithms for DP, Greedy, and Agg, take on real datasets. In Figure 3.17(a), we depict the average running time for the algorithms. The results for all datasets are similar. For example, for Enron, DP took 6.3s on

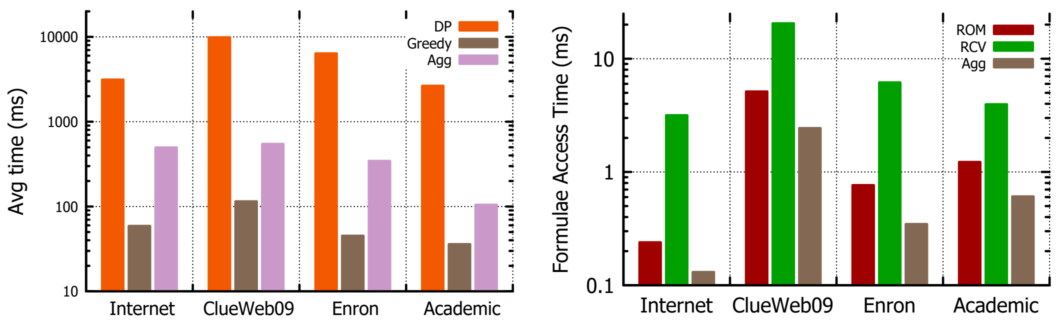
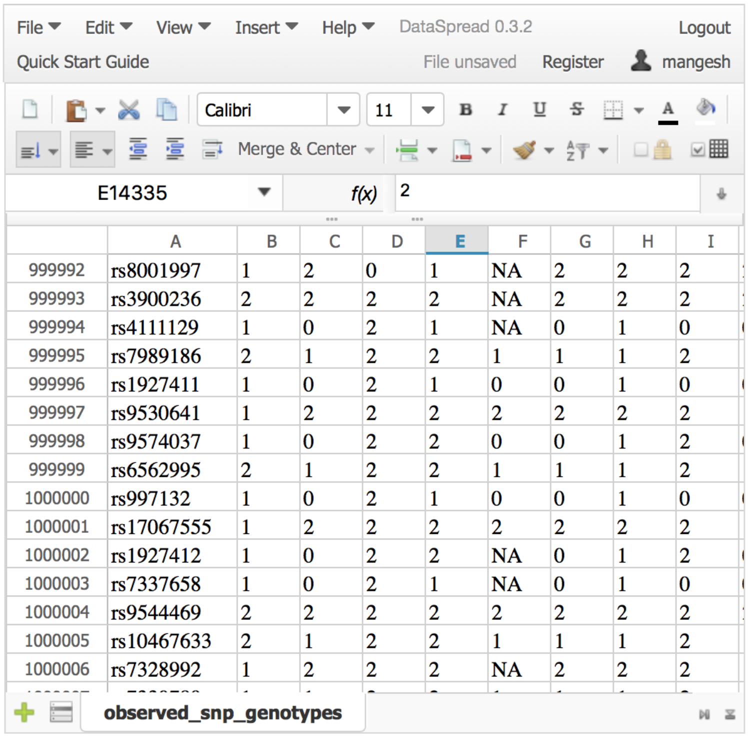


Figure 3.17: (a) Hybrid optimization algorithms: Running time. (b) Average access time for formulae.

average, Greedy took 45ms (a 140× reduction), while Agg took 345ms (a 20× reduction). Thus DP has the highest running time for all datasets, since it explores the entire space of models that can be obtained by recursive partitioning. Between Greedy and Agg, Greedy turns out to take less time. Note that these observations are consistent with our complexity analyses from Section 3.4.3. That said, Agg allows us to trade off running time for improved performance on storage (as we saw earlier). We note that for the cases where the spreadsheets were large, we terminated DP after about 10 minutes, since we want our optimization to be relatively fast. (Note that using a similar criterion for termination, Agg and Greedy did not have to be terminated for any of the real datasets.) To be fair across all the algorithms, we excluded all of these spreadsheets from this chart—if we had included them, the difference between DP and the other algorithms would be even more stark.

d. Real Dataset: Formulae Access Evaluation on PostgreSQL. We next evaluate if our hybrid data models, optimized only on storage, have any impact on the access cost for spreadsheet formulae. Our hope is that spreadsheet formulae focus on "tightly coupled" tabular areas, which our hybrid data models are able to capture and store in separate tables. For this evaluation, we focus on Agg, since it provided the best trade-off between running time and storage costs. Given a sheet in a dataset, for each data model, we measured the time taken to evaluate all the formulae in that sheet, and averaged this time across all sheets and all formulae. Thus, the workload simply comprise all the formulae that are present in a sheet. We plot the results in Figure 3.17(b) in log scale in ms. As a concrete example, on



E14335 f(x) 2

	A	B	C	D	E	F	G	H	I
999992	rs8001997	1	2	0	1	NA	2	2	2
999993	rs3900236	2	2	2	2	NA	2	2	2
999994	rs4111129	1	0	2	1	NA	0	1	0
999995	rs7989186	2	1	2	2	1	1	1	2
999996	rs1927411	1	0	2	1	0	0	1	0
999997	rs9530641	1	2	2	2	2	2	2	2
999998	rs9574037	1	0	2	2	0	0	1	2
999999	rs6562995	2	1	2	2	1	1	1	2
1000000	rs997132	1	0	2	1	0	0	1	0
1000001	rs17067555	1	2	2	2	2	2	2	2
1000002	rs1927412	1	0	2	2	NA	0	1	2
1000003	rs7337658	1	0	2	1	NA	0	1	0
1000004	rs9544469	2	2	2	2	2	2	2	2
1000005	rs10467633	2	1	2	2	1	1	1	2
1000006	rs7328992	1	2	2	2	NA	2	2	2

Figure 3.18: Genomics Use Case: VCFs in DataSpread.

Internet, ROM has a formula access time of 0.23, RCV has 3.17, and Agg has 0.13. Thus, Agg provides a substantial reduction of 96% over RCV and 45% over ROM—*even though Agg was optimized for storage and not for formula access.* Even though a region access in Agg can span multiple primitive data models, for real datasets the access is generally confined to a single data model—thereby leading to a significant benefit for Agg. Specifically, for Internet, about 98.47% of formulae access only access a single primitive data model, 1.43% between 2 and 10, and only 0.1% formulae access more than 10 primitive data models. When the number of accessed tables is more then one, ROM has an edge as compared to Agg—concretely for one such formula the access times (in ms) are 0.62, 0.31, and 0.99 for RCV, ROM, and Agg respectively. This validates our design of hybrid data models to store spreadsheet data. While the performance numbers for real spreadsheet datasets are small for all data models (due to the size limitations in present spreadsheets), when scaling up to large datasets, and formulae on them, these numbers will increase proportionally, at which point it is even more important to opt for hybrid data models, as we will see next.

To better understand the gains of the hybrid data model, we plot the access times of

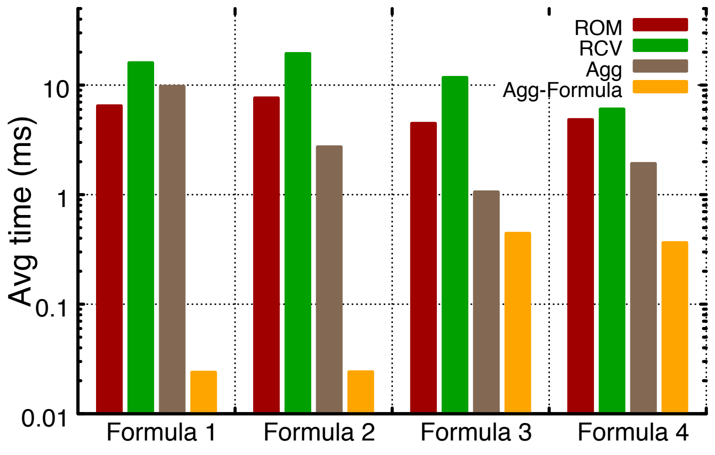


Figure 3.19: Access time for individual formulae.

some representative formulae from Internet in Figure 3.19. Here, in addition to plotting the access times for ROM, RCV, and Agg, we plot Agg-formula, which is the hybrid data model optimized for formula accesses, as discussed in Section 3.4.4. The Agg-formula is able to substantially reduce the access time across all the formulae. For example, for Formula 4, the access time for Agg-formula is 7% of ROM and 15% of Agg. For a sample of forty spreadsheets from Internet, the formula access times for ROM, Agg, and Agg-formula were 67% of RCV, 52% of RCV, and 10% of RCV respectively.

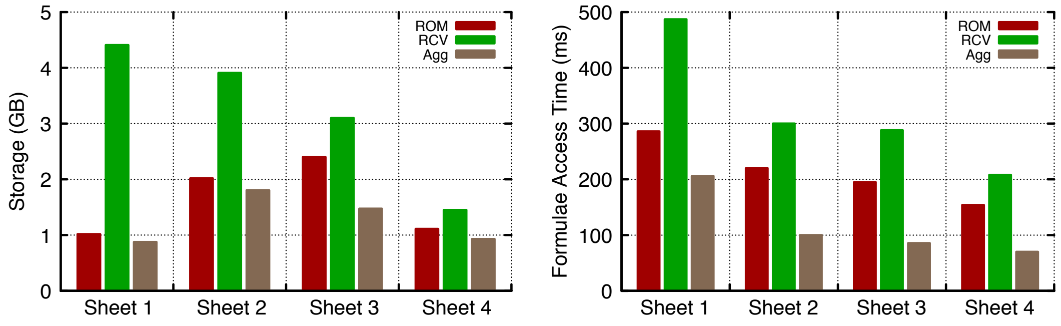


Figure 3.20: Synthetic sheets (a) Storage. (b) Access time.

e. Synthetic Dataset: Storage and Formula Access Evaluation We now run our tests on large synthetic spreadsheets with 100+ million cells to evaluate our techniques in large dataset scenarios. We create synthetic spreadsheets by populating an empty sheet with twenty dense rectangular regions to simulate randomly placed tables. We add 100 randomly generated formulae that access rectangular ranges of these tables. Figures 3.20(a)

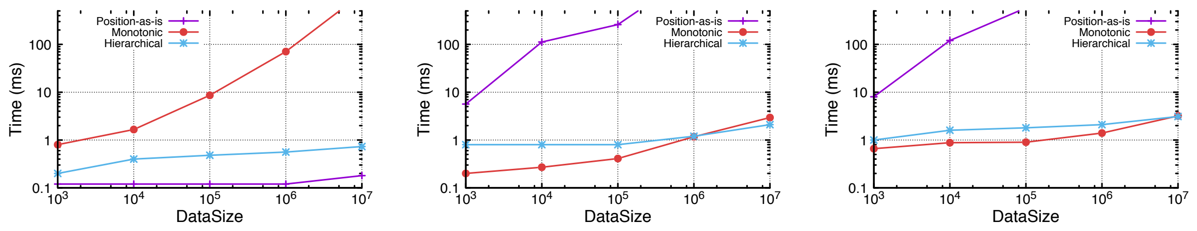


Figure 3.21: Positional mapping performance for (a) Select. (b) Insert. (c) Delete.

and 3.20(b) depict the storage requirements and the formulae access time respectively for four synthetic spreadsheets, which are in the decreasing order of density (the fraction of cells that are filled-in in the minimum bounding rectangle). For both storage and access, we find that Agg is better than ROM, which is better than RCV; as density is decreased, RCV's performance becomes closer to ROM. Agg performs the best, providing substantial reductions of up to 50-75% of the time taken for access with ROM or RCV.

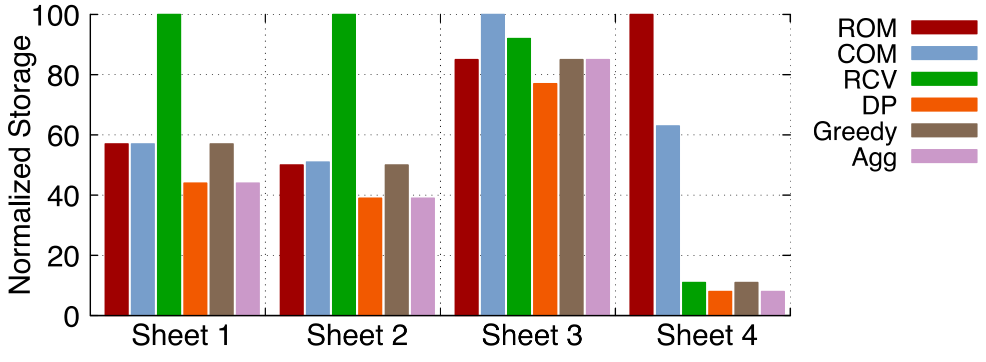


Figure 3.22: Storage comparison for sample spreadsheets.

f. Drill-Down of storage optimization algorithms We now drill deeper into the storage optimization algorithms and understand their behavior with respect to the characteristics of spreadsheets. We selected four sample sheets from our dataset to represent variations in terms of data density and layout of data, which is either horizontal for most part or vertical for most part. For these sheets, we contrast their storage requirements for the different data models. We plot the results in Figure 3.22, where we depict the normalized storage across sheets. For each sheet we have normalized the data model that performs the worst to 100, and scaled the others accordingly.

The four spreadsheets show the variation among the different models in terms of storage requirements. Sheets 1 and 2 have substantial storage savings for ROM and COM when

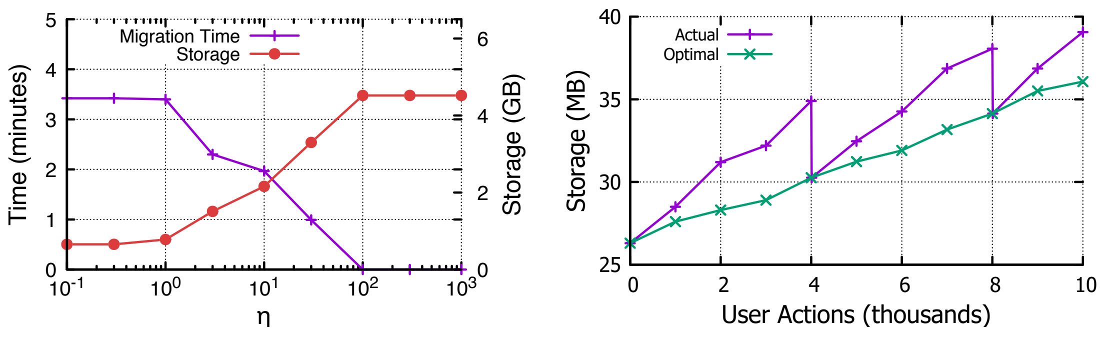


Figure 3.23: Incremental Hybrid Decomposition: (a) Trade-off with respect to η. (b) User operations vs. Storage.

compared with RCV since they are relatively high density. On the other hand, Sheet 4 has has substantial storage savings for RCV as compared to ROM and COM due to its relatively low density. For Sheet 3 (4), ROM's (COM's) storage requirement is less then that of COM (ROM). This is due to the distribution of the cells, which span for the most part in the vertical direction for Sheet 3 and in the horizontal direction for Sheet 4. Except for Sheet 3, for all other sheets, the solution provided by Agg is close to DP. For Sheet 3, the optimization algorithms are not able to perform much better in terms of cost saving than the primitive data models. This is due to the fact that the sheet has both dense and sparse regions.

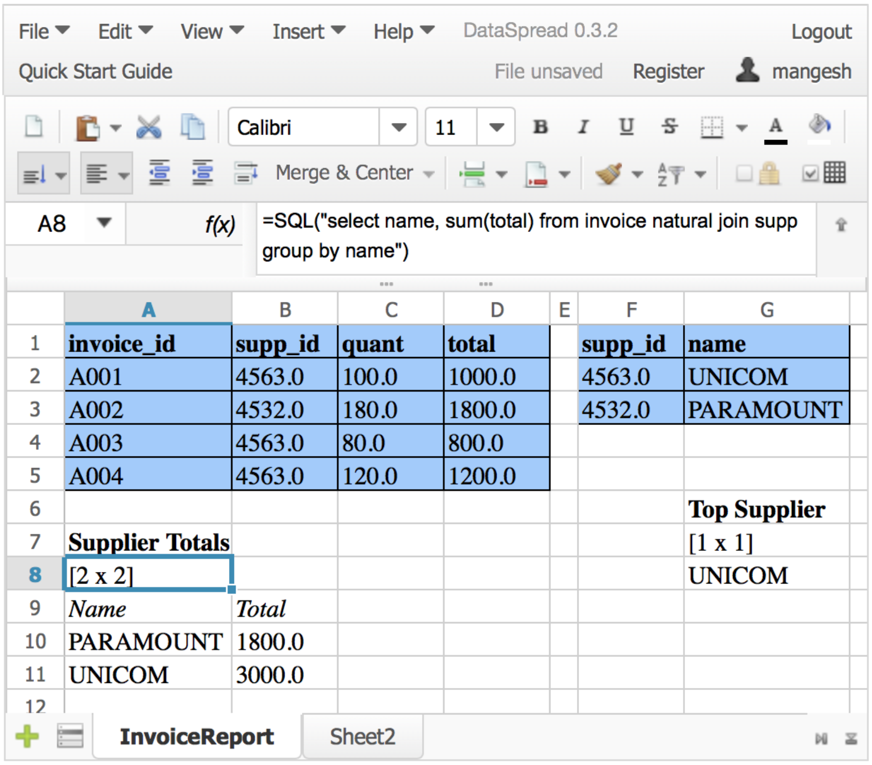
g. Incremental Maintenance We now evaluate whether our representation schemes can be maintained efficiently in the face of edits. Note that in practice there will be periods where the DataSpread is idle, and so we can run the hybrid optimization algorithms then, but it is still valuable to ensure that the choice of data model is aware of the existing layout. To illustrate our incremental decomposition approach from Section 3.4.4, For this, we consider a synthetic spreadsheet as described in part e. of Section 3.7.2. We store the spreadsheet using the Agg-based hybrid data model. In the absence of user operation traces, we develop a generative model for update operations. We consider the following four operations. *(i)* Change the value of an existing cell. *(ii)* Add a new cell at an arbitrary location. *(iii)* Add a new row. *(iv)* Add a new column. Motivated from our user study, we consider that the above four operations are performed with probabilities 0.6, 0.2, 0.1999, and 0.0001 respectively. We fix the value of η (the trade-off factor between migration and storage) to 1.0 and run incremental maintenance with Agg after each batch of 1000 user updates are performed. We plot the storage requirement against the number of user updates in Figure 3.23(b). The *actual* line in the graph indicates the storage requirement, which has a saw-tooth like behavior. The drop in the graph correspond to the points where the incremental maintenance algorithm chose a new decomposition and migrated to it: thus, there was no migration performed at batch 1, 2, 3, but there was one at batch 4. We also plot the storage for the non-incremental variant of Agg, which we obtain by running incremental decomposition and setting η to 0. Overall, we find that a policy of this form (with $\eta = 1$) only performs migration when the structure within the spreadsheet has substantially changed.

To study the impact of η, we consider one such point where the spreadsheet has diverged from its original Agg-based data model. We run the Agg variant of incremental maintenance algorithm on varying η. We plot η's impact of the time taken to migrate and the storage requirement of the final decomposition in Figure 3.23(a). Here, as we increase the value of η we observe that the migration time decreases and the storage requirement increases. At lower values of η, the algorithm gives preference to finding the optimal solution while ignoring the migration cost. We can observe this from the low storage cost, and the high time required to migrate the data in to the new decomposition. When $\eta > 100$, the optimization aims at minimizing the migration cost at the expense of sacrificing the optimality of storage. Here, we observe a zero migration time, as the algorithm returns the original decomposition, and has the worst storage requirement.

3.7.3 Presentational Access with Updates

Takeaway 3.8. *Hierarchical positional mapping retains the rapid fetch benefits of position-as-is, while also providing rapid inserts and updates. Thus,* ***hierarchical positional mapping is able to perform positional operations within a few milliseconds****, while the other schemes often take seconds on large datasets. Overall,* ***our hierarchical positional mapping schemes support spatial access with updates, validating the fact that our storage engine can support interactivity.***

We now evaluate spatial access (with updates) by studying our positional mapping meth-



A8 f(x) =SQL("select name, sum(total) from invoice natural join supp group by name")

	A	B	C	D	E	F	G
1	**invoice_id**	**supp_id**	**quant**	**total**		**supp_id**	**name**
2	A001	4563.0	100.0	1000.0		4563.0	UNICOM
3	A002	4532.0	180.0	1800.0		4532.0	PARAMOUNT
4	A003	4563.0	80.0	800.0			
5	A004	4563.0	120.0	1200.0			
6							**Top Supplier**
7	**Supplier Totals**						[1 x 1]
8	[2 x 2]						UNICOM
9	*Name*	*Total*					
10	PARAMOUNT	1800.0					
11	UNICOM	3000.0					
12							

Figure 3.24: Customer Management in DataSpread.

ods (Section 3.5) on synthetic datasets. We compare our hierarchical positional mapping (denoted hierarchical), with position as-is (denoted position-as-is): this is the approach a traditional database with a B+ tree would use. In addition, motivated by the online dynamic reordering technique [36, 37], we consider another baseline (denoted monotonic), where we store a monotonically increasing sequence of identifiers (with gaps) to capture the position. Using this sequence we dynamically order the tuples at run-time (by sorting); whereas the gaps in the sequence enable efficient insert/delete operations. The dynamic reordering sacrifices the performance of the fetch operation as it needs to discard $n-1$ tuples to fetch n^{th} tuple.

We operate on a dense synthetic dataset ranging from 10^3 to 10^7 rows, with 100 columns with all cells filled; and repeat this 1000 times. We evaluate the performance of a single ROM table with all of the data; evaluations for other data models are similar. Figure 3.21 displays the average time taken to perform a fetch, insert, and delete of a single (random) row.

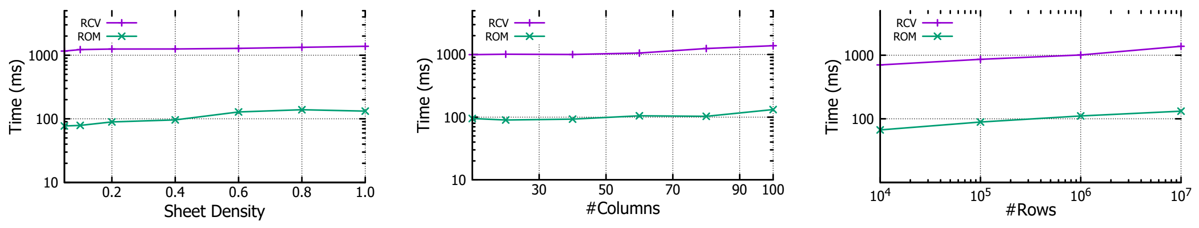


Figure 3.25: Update range performance vs (a) Sheet Density (b) Column Count (c) Row Count

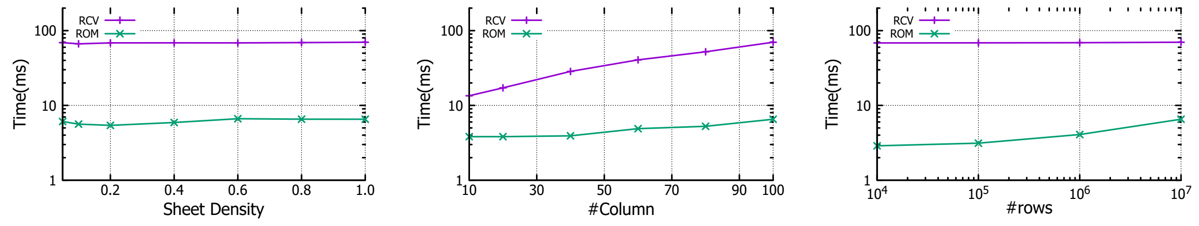


Figure 3.26: Insert row performance vs (a) Sheet Density (b) Column Count (c) Row Count

We see that position-as-is performs well for fetch. However, the insert and delete time increases rapidly with the data size, due to cascading updates; thus, beyond a data size of 10^5, position-as-is is no longer interactive ($> 500ms$) for insert and delete. Conversely, the response time of monotonic for fetch increases rapidly with data size. This is again expected, as we need to linearly search through the monotonic keys to retrieve the required records—making it infeasible for large datasets. Lastly, we find that hierarchical performs well *for all operations* and the performance does not degrade even with data sizes of 10^9 tuples. In comparison with the other schemes, hierarchical performs all of the three aforementioned operations in few milliseconds, which makes it the practical choice for spatial access with updates.

Finally, we perform an evaluation of spatial access with updates on varying various parameters of the synthetic spreadsheets. For this evaluation, we focus on the two primitive data models, *i.e.*, ROM and RCV with the spreadsheet being represented as a single table in these data models. Since we use synthetic datasets where cells are "filled in" with a certain probability, we did not involve hybrid data models, since they would (in this artificial context) typically end up preferring the ROM data model. These primitive data models are augmented with hierarchical positional mapping. We consider the performance on varying several parameters of these datasets: the density (*i.e.*, the number of filled in cells), the

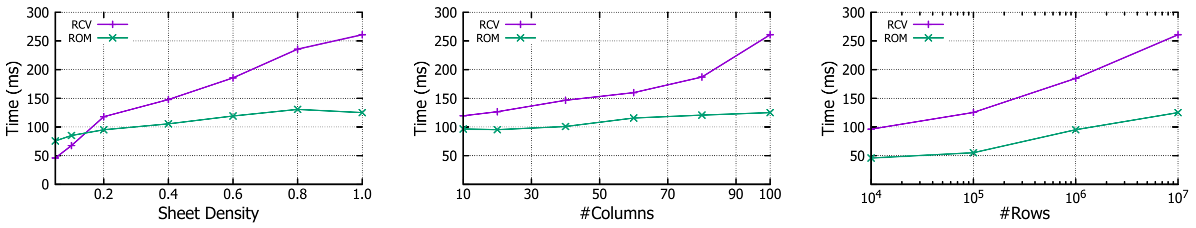


Figure 3.27: Select performance vs — (a) Sheet Density (b) Column Count (c) Row Count

number of rows, and the number of columns. The default values of these parameters are 1, 10^7 and 100 respectively. We repeat each operation 500 times and report the averages.

In Figure 3.27, we depict the charts corresponding to average time to perform a random select operation on a region of 1000 rows and 20 columns. This is, for example, the operation that would correspond to a user scrolling to a certain position on our spreadsheet. As can be seen in Figure 3.27(a), ROM starts dominating RCV beyond a certain density, at which point it makes more sense to store the data in as tuples that span rows instead of incurring the penalty of creating a tuple for every cell. Nevertheless, the best of these two models takes less than 150ms across sheets of varying densities. In Figure 3.27(b)(c), since the spreadsheet is very dense (density=1), ROM takes less time than RCV. Overall, in all cases, even on spreadsheets with 100 columns and 10^7 rows and a density of 1, the average time to select a region is well within 500ms.

Figures 3.25 and 3.26 depict the corresponding charts for updating a region of 100 rows and 20 columns, and inserting one row of 100 columns for the primitive data models. In Figure 3.25, we find that the update time taken for RCV is a lot higher than the time for inserts or selects. This is because in this benchmark, DataSpread assumes that the entire region update happens at once, and fires $100 \times 20 = 2000$ update queries one at a time. In practice, users may only update a small number of cells at a time; and further, we may be able to batch these queries or issue them in parallel to further save time. In Figure 3.26, we find that like in Figure 3.25, the time taken for updates on ROM is faster than RCV since it only needs to issue one query, while RCV needs to issue multiple queries. However, in this case, since the number of queries issued is small, the response time is always within 100ms.

Overall, for both RCV and ROM, for inserting a row, the time is well below 500ms for all of the charts; for updates of a large region, while ROM is still highly interactive, RCV ends up taking longer since 1000s of queries need to be issued to the database. In practice, users won't update such a large region at a time, and we can batch these queries.

3.7.4 Qualitative Evaluation

Takeaway 3.9. DataSpread *enables users to* ***interactively work on large spreadsheets****. The direct manipulation and database-oriented features of* DataSpread *enable* ***interactive management of data via database tables on a spreadsheet interface****.*

We now evaluate DataSpread to see how it can handle the use cases described earlier. With our genomics use case, we evaluate the *scalability* of DataSpread, and with our customer management use case, we evaluate the *functionality*.

a. Evaluating Scale for Genomics: For this evaluation, we used a VCF file provided by our biology collaborators, as described in Example 1, and used it to perform basic exploration. We contrast the performance of DataSpread with Excel. Our VCF file has 1.3M rows and 284 columns. Unfortunately, we were unable to load this file in Excel since it exceeds Excel's limits. Importing the file in DataSpread takes about a minute. On the other hand, even after reducing the VCF file to 1M rows, Excel is unable to import the file within an hour. After substantially reducing the file size to 130K rows, we were able to import it into Excel in about 10 minutes. After loading the 1.3M VCF file, we were able to take advantage of DataSpread's efficient positional access to scroll up and down to explore the data with interactive (sub-second) response times. Figure 3.18 shows a screenshot of the file in DataSpread, having scrolled to the millionth row.

b. Evaluating Functionality for Customer Management: For evaluating functionality, as described in Example 2, we leverage the database-oriented operations discussed in Section 3.2. Using linkTable, we first establish a two-way synchronization between the spreadsheet regions and the invoice and supp tables in the database (Figure 3.24). These linked regions enable us to directly manipulate the underlying tables via spreadsheet operations such as cell updates; this is not possible in spreadsheet tools that only allow one-way import of data from a backend database to a spreadsheet. We used the sql function in cell A8 to join the two tables and perform grouping and aggregation; this is less cumbersome and more efficient compared to Excel's vlookup and pivot tables, and indexed into the composite value in A8 to display the results in A9:B11. Finally, we use the project and select functions to get the top supplier in cell G8; any updates to the underlying tables are automatically reflected in the function's output.

3.8 RELATED WORK

Our work draws on related work from multiple areas: *(i)* order or array-based database management systems, and *(ii)* hybrid storage schemes.

1. Order-aware database systems. Some limited aspect of spatial awareness, in particular, order, has been studied. The early work of online dynamic reordering [36] supports data reordering based on user preference, citing a spreadsheet-like interface [37] as an application. More recently, there has been work on array-based databases, but most of these systems do not support edits, like SciDB [38] which supports an append-only, no-overwrite data model or TileDB [39], which supports only restricted forms of edits where values in cells are modifiable but new rows or columns cannot be added. Our position-aware access efficiently supports general updates as required by our spatial interface.

2. Hybrid, access-optimized storage schemes. Utilizing query workloads to select appropriate physical designs has been a long-standing research problem, with work on auto tuning [40, 41], cracking [42], and materialized views [43] targeting the selection and organization of indexes and views to match queries. Other work examines the use of hybrid row-column stores for mixed OLAP-OLTP workloads [44, 45]. Some work focuses on partitioning—vertical [46, 47], horizontal [48], or both [49, 50]. While we similarly consider vertical and horizontal partitioning, in addition to transposing the data (COM) and storing the data in a key-value fashion (RCV), in contrast, our work emphasizes the structure and skew of data to determine appropriate models for positional access and update, a first-class citizen in a spatial interface.

3.9 CONCLUSIONS

In this chapter, we focused on developing a storage engine for our prototype DataSpread, characterizing key requirements in the form of spatial awareness and access. We addressed spatial awareness by proposing three primitive data models for representing spreadsheet data, along with algorithms for identifying optimal hybrid data models from recursive decomposition. Our hybrid data models provide substantial reductions in terms of storage (up to 20–50%) and formula evaluation (up to 50%) over the primitive data models. For spatial access, we couple our hybrid data models with a hierarchical positional mapping scheme, making working with large spreadsheets interactive.

CHAPTER 4: ASYNCHRONOUS FORMULA COMPUTATION

Formula computation enables end-users with little programming experience to be able to interrogate their data, and compute derived statistics. However, the sheer volume of data available for analysis in a host of domains exposes the limitations of traditional formula computation. A recent study exploring Microsoft Excel forum posts on Reddit describes several instances of Excel becoming unresponsive while computing formulae [12]. One user posted[1] that complex calculations on Excel can take as long as four hours to finish, during which time the user interface is unresponsive:

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

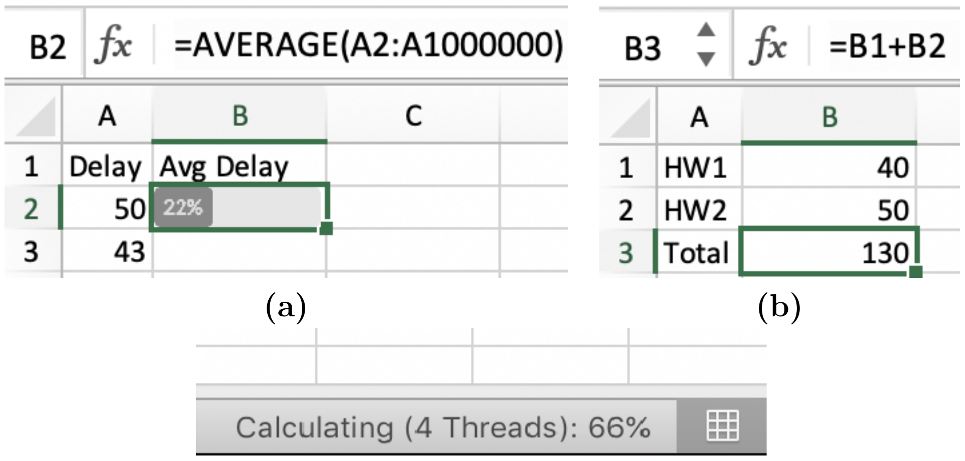
Another user reported using spreadsheets to track their entire life, and periodically cull data to keep the size manageable, but they still have trouble with formula computation:

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

The chief culprit for this unresponsiveness is that in traditional spreadsheet systems, every change, be it changing values or formulae, triggers a sequence of computation of dependent formulae. This sequence could take minutes to complete, depending on the size of the data and complexity of the formulae. Since these systems aim to present a "consistent" view after any update, *i.e.*, one with no stale values, they forbid users from interacting with the spreadsheet while the computation is being performed, limiting interactivity. They *only return control to the user after the computation is complete*: the only indication to the user is a bar at the bottom, as in Figure 4.1(c), with no viewing, scrolling, or edits allowed. Recent studies have shown that even delays of 0.5s can lead to fewer hypotheses explored and insights generated [51], so this *synchronous computation* approach is not desirable.

One workaround that traditional spreadsheet systems provide is a *manual computation* approach, wherein computation of dependent formulae is performed only when triggered

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



B2 fx =AVERAGE(A2:A1000000)

	A	B	C
1	Delay	Avg Delay	
2	50	22%	
3	43		

(a)

B3 fx =B1+B2

	A	B
1	HW1	40
2	HW2	50
3	Total	130

(b)

Calculating (4 Threads): 66%

(c)

Figure 4.1: (a) Our proposed asynchronous computation maintains interactivity and consistency by showing computation status instead of a stale value. (b) Manual computation mode in traditional spreadsheets achieve interactivity but violate consistency. (c) Automatic calculation mode in traditional spreadsheets achieve consistency but keep user interface non-responsive for the duration of the computation.

manually by users. This method breaks consistency, as stale values are visible to the users, as in Figure 4.1(b), potentially leading to users drawing incorrect conclusions.

Towards Interactivity and Consistency. As we introduced in Section 2.3, we describe 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 4.1(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 formulae are computed in the background.

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

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

Dependencies: Challenges and Approach. Since we need to preserve both interactivity and consistency, once a change is made, we need to quickly identify cells dependent on that change, and therefore must be "blurred out", or made unavailable, as in B2 in Figure 4.1(a). One simple approach is to traverse the network or graph of formula dependencies to find all dependent cells, and then make them unavailable. However, during this period, the entire spreadsheet is unavailable, so we aim to minimize the time spent in identifying dependent cells. Unfortunately, for computationally heavy spreadsheets, a traditional dependency graph that captures formula dependencies at the cell level [52] can be quite large, so identifying dependencies can be computationally intensive and cannot be done in a bounded time. Ideally, we would like to do this within interactive timescales (less than 500 ms [51]), without sacrificing consistency.

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

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

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

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

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

We incorporate our asynchronous computation model in DataSpread. As discussed earlier DataSpread achieves scalability by utilizing a two-tiered memory model, where data resides in an underlying relational database and is fetched on-demand into main-memory, which is limited in size. This introduces additional challenges that go beyond those found in traditional spreadsheets which are completely main-memory resident. Note, however, that our techniques for decreasing unavailability apply equally well to traditional spreadsheets as well as DataSpread. We additionally discuss how we support this two-tiered memory model in this chapter. For the two-tiered memory model, the computation schedule impacts not only the unavailability metric but also the total computation time significantly.

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

1. Asynchronous Computation. In Section 4.1, we introduce the asynchronous computation model ensuring interactivity and consistency. Additionally, we propose a novel unavailability metric to quantitatively evaluate our model.

2. Fast Dependency Identification. In Section 4.2, we propose the idea of lossily compressing the dependency graph to identify dependencies in a bounded time. We show that the problem is NP-Hard, and develop techniques for compression and maintenance of this graph.

3. Computation Scheduling. In Section 4.3, we discuss the importance of finding an efficient schedule for computing formulae. Since, not only is finding the optimal schedule NP-Hard but also obtaining a schedule expensive, we propose an on-the-fly scheduling technique.

4. Supporting Asynchronous Computation in DataSpread. In Section 4.4, we describe the incorporation of the asynchronous formula computation model within DataSpread.

5. Supporting Asynchronous Computation at Scale. In Section 4.4.2, we describe supporting the asynchronous computation at scale.

6. Experimental Evaluation. Throughout the chapter, we provide illustrative experiments to demonstrate individual ideas. In Section 4.5, we discuss our experimental setup and provide evaluation with real-world spreadsheets.

4.1 ASYNCHRONOUS COMPUTATION

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

For simplicity, we explain the concepts and techniques in the context of standard spreadsheet tools, which are main-memory-based, where once loaded the cost of data retrieval is negligible compared to the cost of formula evaluation. The techniques, as described for main-memory systems, are beneficial even if used in systems with different memory settings. In Section 4.4.2, we extend our techniques to two-tier memory systems wherein data retrieval cost is significant.

While the techniques discussed in this chapter 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.

4.1.1 Standard Spreadsheet Terminology

We now formally introduce spreadsheet terminology that we utilize throughout the chapter; some content is from Chapter 3 is repeated to keep this chapter relatively self contained.

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

A cell may contain *content* that is either a *value* or a *formula*. A value is a constant belonging to some fixed type. For example, in Figure 4.1(b), cell A1 (column A, row 1) contains the value HW1. In contrast, a formula is a mathematical expression that contains values and/or cell/range references as arguments to be manipulated by operators or functions. A formula has an *evaluated value*, which is the result of evaluating the expression, with cell references substituted by their values or evaluated values. For the rest of the chapter, we shall use the term "value" to refer to either the value or the evaluated value of a cell, depending on what the cell contains. In addition to a value or a formula, a cell could also additionally have formatting associated with it, *e.g.*, width, or font. For the purpose of this chapter, we focus only on computation.

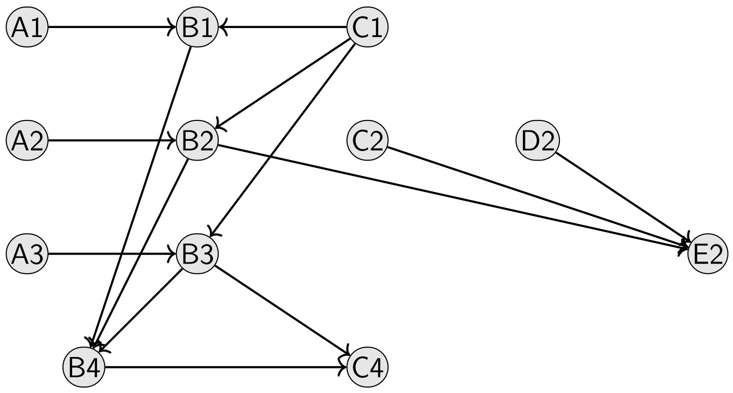


Figure 4.2: A dependency graph that captures the dependencies of Example 4.1 at the granularity of cells.

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

Example 4.1. *A spreadsheet with the following formulae: (i) B1=A1*C1, (ii) B2=A2*C1, (iii) B3=A3*C1, (iv) B4=SUM(B1:B3), (v) C4=B3+B4, and (vi) E2=SUM(B2:D2).*

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

Definition 4.1 (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 4.2 (Dependency). *For two cells u and v, $u \Rightarrow v$ is a* dependency *if there is a sequence $w_0, w_1, \ldots, w_n$ of cells where $w_0 = u$, $w_n = v$, and for all $i \in [n]$, $w_{i-1} \rightarrow w_i$ is a direct dependency. Here, u is called a* precedent *of v, and v is called a* dependent *of u. We denote the set of dependents of a cell u as Δ_u.*

One can construct a conventional *dependency graph* of direct dependencies [52]. Figure 4.2 depicts the graph for the formulae in Example 4.1 at the granularity of cells. Here, each vertex corresponds to a single cell, *e.g.*, A1. The edges in the graph indicate direct dependencies. For example, the directed edge from A1 to B1 indicates a direct dependency due to formula A1*C1 in cell B1. The dependencies of a cell u are therefore the vertices that are reachable from u in the dependency graph. For example, cell B1 has B4 and C4 as dependents. As this dependency graph captures dependencies at the granularity of cells, this graph grows quickly when the ranges mentioned in the formulae are large [52]. For example, a formula SUM(A1:A1000) in cell F2 will require 1,001 vertices and 1,000 edges to capture the dependencies.

4.1.2 Design Principles

We introduce consistency and interactivity as two fundamental principles that any system should maintain during formula computation. Spreadsheets should be *consistent*, *i.e.*, they should not display stale values. For example, if a cell B2 contains the formula SUM(A1:A225500) and the user updates the value in cell A1, the user should not see the stale value in B2 until the corresponding formula is recomputed. Along with consistency, spreadsheet systems must ensure *interactivity*, meaning they should react to user events, such as cell updates, rapidly, and provide users with results as soon as possible—this is crucial for the usability of any interactive exploration systems [51]. Thus, we introduce the following two design principles by which our solution must abide.

Principle 4.1 (Consistency)**.** *Never display an outdated or incorrect value on the user interface.*

Principle 4.2 (Interactivity)**.** *Return control to users within a bounded time after any cell update user event.*

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

Synchronous Computation Model. Traditional spreadsheet systems adopt a *synchronous computation model*, where, upon updating u, the entire spreadsheet becomes unavailable during the evaluation of cells that are dependent on u. The spreadsheet system waits for all of the computation to complete before providing updated values and returning control back to the user—thereby adhering to the consistency principle. However, the waiting time can be substantial for computationally intensive spreadsheets. According to our recent Reddit study [12], waiting for formula computation is one of the primary sources of poor interactivity. In other words, when the number of cells dependent on u is large, this model sacrifices interactivity, with often minutes to hours of unresponsiveness.

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

One naïve asynchronous approach could be to merely modify the synchronous model to return control to the user immediately after an update, even before all the dependent cells are computed. However, similar to the *manual computation* option found in traditional spreadsheet systems, this approach violates consistency. Consider Figure 4.1(b), where even after updating the value of cell `B1` from `80` to `40`, the cell `B3` is not updated to the correct value of `90` unless a full computation of the spreadsheet is triggered manually—violating consistency, cell `B3` shows `130`, a stale value.

To satisfy the consistency principle within the asynchronous computation model, we instead provide users with the cells that the system can ensure to have correct values in a short time, while notifying users of cells that have stale values—see Figure 4.1(a), where upon updating `A1` the computation of cell `B2` is performed in the background and the computation progress is depicted by a progress bar. Our solution is to add a "dependency identification" step before computation of any dependent formulae. The goal of this step is to efficiently identify the cells that do not depend on an updated cell, so that they can be quickly marked clean and "control" of them can be returned to the user.

4.1.3 New Concepts

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

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

Definition 4.3 (Clean Cell). *We consider a cell u to be* clean *if and only if (i) all of u's precedents are clean and (ii) u's evaluated value is determined to be up-to-date.*

Definition 4.4 (Dirty Cell). *We consider a cell u to be* dirty *if and only if (i) at least one of u's precedents are dirty or (ii) the u's evaluated value is determined to be not up-to-date.*

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

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

Definition 4.5 (Unavailable Cell). *We consider a cell c to be* unavailable *if and only if a user cannot act on c either because (i) c is determined to be dirty or (ii) the system has not yet determined if c is in the clean or dirty state or (iii) the user interface is unresponsive.*

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

Unavailable and Dirty Time. Quantifying the time a cell is unavailable to the user to act upon is an important factor for understanding the usability of the spreadsheet. Similarly, the

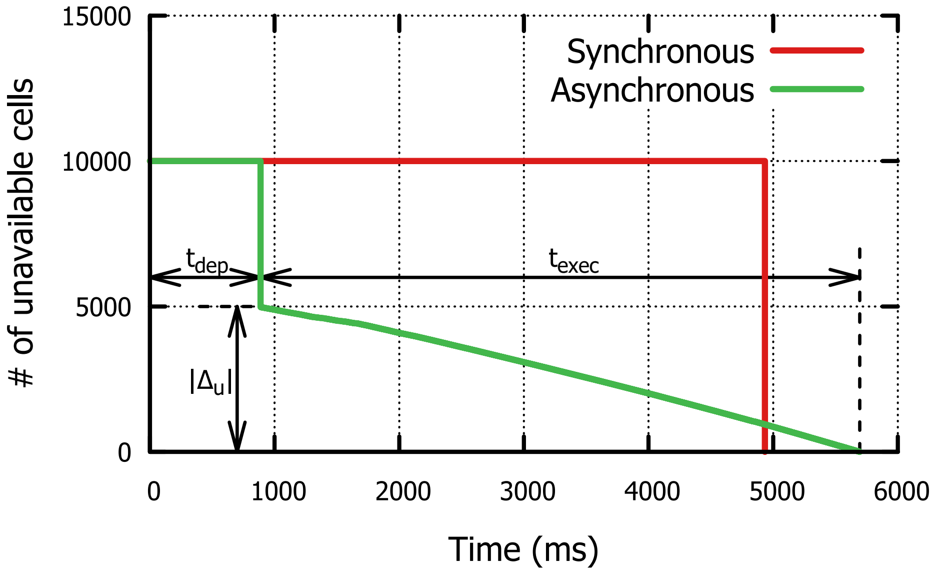


Figure 4.3: Unavailability comparing synchronous and asynchronous models. For the asynchronous model, t_{dep} denotes dependency identification time, Δ_u is the set of cells that are determined to be dependent on u and therefore need computation, and t_{exec} denotes computation time for these cells.

dirty time is the time a cell spends in the dirty state. We formalize the notion of unavailable and dirty time as below.

Definition 4.6 (Unavailable Time)**.** *The unavailable time of a cell c, denoted as* unavailable(c)*, is the amount of time that c remains in the unavailable state after an update.*

Definition 4.7 (Dirty Time)**.** *The dirty time of a cell c, denoted as* dirty(c)*, is the amount of time that c remains in the dirty state after an update.*

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

Definition 4.8 (Unavailability)**.** *The unavailability U_M for a computation model M is given by $U_M = \int_0^t D(t)\,dt = \sum_{c \in S} \text{unavailable}(c)$, where $D(t)$ denotes the number of unavailable cells at time t and S is the set of all spreadsheet cells.*

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

Illustrative Experiment 1: Asynchronous vs. Synchronous Computation. The goal of this experiment is to quantitatively compare the asynchronous and synchronous computation models using unavailability. We describe the experimental setup later in Section 4.5. Here, we adopt a conventional dependency identification mechanism as described in Section 4.1.1 and a naïve schedule for computing cells—we will build on this and develop better variants later. We use a synthetic spreadsheet with a total of 10,000 cells out of which 5,000 cells are formulae dependent on a cell u. We update the value of u and plot the number of unavailable cells on the y-axis with respect to time on the x-axis for both computation models—see Figure 4.3. The synchronous computation model (in red) performs poorly under unavailability, since it keeps the interface unresponsive for the entire duration of computation of all of the dependent cells. The asynchronous computation model (in green) performs better in terms of unavailability, since it allows users to interact with most of the spreadsheet cells while performing calculations asynchronously in the background, with the cells incrementally returned to the user interface as they are complete.

We now describe how the computations proceed with respect to time for both models—refer to Figure 4.3. Upon updating u (at $time = 0$), the asynchronous model identifies dependents of u, as is marked by t_{dep} on the graph. For both models, all 10,000 cells in the sheet are unavailable for the first 890 ms, as the sheet is unresponsive. After this point, the asynchronous model has determined which cells are clean and which cells are dirty, and it returns the clean cells to the user. Thus, the number of unavailable cells drops down to 5,000 from 10,000 after 890 ms. However, under the synchronous model, control has not been returned to the user, and thus all cells are still unavailable. Under the asynchronous model, at the 5,700 ms mark, all of the cells have been computed and marked clean—this is slightly after the 4,900 ms mark, which is when the synchronous model returns control of all of the cells to the user. This time difference is due to the fact that the asynchronous model takes some time to identify dependent cells in a separate step from computing them, while the synchronous model does not have to have this separate step. Note that the area under the green curve is greater than that under the red curve, and therefore the asynchronous model performs better under unavailability.

Takeaway 4.1. *The asynchronous computation model improves usability of spreadsheets, without forgoing correctness, by (i) quickly returning control to the user and (ii) incrementally making cells available.*

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

4.2 FAST DEPENDENCY IDENTIFICATION

In this section, we propose our first technique for decreasing unavailability: *identifying dependencies in a bounded time*. Upon updating u, our strategy is to quickly identify u's dependencies, ideally, within a bounded time—this enables us to promptly identify the cells that do not depend on u as clean and return their control to the user. Our strategy, the *dependency table*, aims to reduce t_{dep} in Figure 4.3, which is the time during which the user interface is non-responsive for the asynchronous computation model. Reducing t_{dep} is particularly crucial when the update affects a small number of cells relative to the size of the spreadsheet. We propose compression to accelerate dependency identification by grouping a large number of dependent cells into a smaller number of regions. We then discuss construction and maintenance of the compressed dependency table.

4.2.1 Motivation and Problem Statement

After a user updates a cell u in a spreadsheet, to minimize the number of unavailable cells over time, we need to quickly identify the cells that depend on u. Until we can determine that a cell c is independent of u or not, we cannot designate c as clean and return its control to the user. For example, within the asynchronous computation model in Figure 4.3, we return the control to the user in 890 ms, which corresponds to the time it takes dependency identification to finish.

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

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

Problem 4.1 (Dependency Identification)**.** *Design a data structure that, upon updating u, quickly (preferably in bounded time) identifies u's dependencies. Additionally, modifications to the data structure, i.e., inserts and deletes, should be quick (again, preferably in bounded time).*

Our proposed method of capturing dependencies is to maintain a *dependency graph*. Rather than the conventional method of recording dependencies between individual cells (Figure 4.2), we capture dependencies between regions—this substantially reduces the size of the dependency graph. Figure 4.4 shows the dependency graph for Example 4.1. Our dependency graph has the following four components. *(i)* A *cell vertex* corresponding to each cell, in gray, *e.g.*, A1, B1. *(ii)* A *range vertex* corresponding to each range that appears in at least one formula, in red, *e.g.*, B1:B3. *(iii)* A *formula edge* from u to v if u is an operand in the formula of cell v, *e.g.*, the edge from A1 to B1. *(iv)* An *inherent edge* from u to v if cell u is contained in range v, *e.g.*, the edge from B1 to B1:B3.

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

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

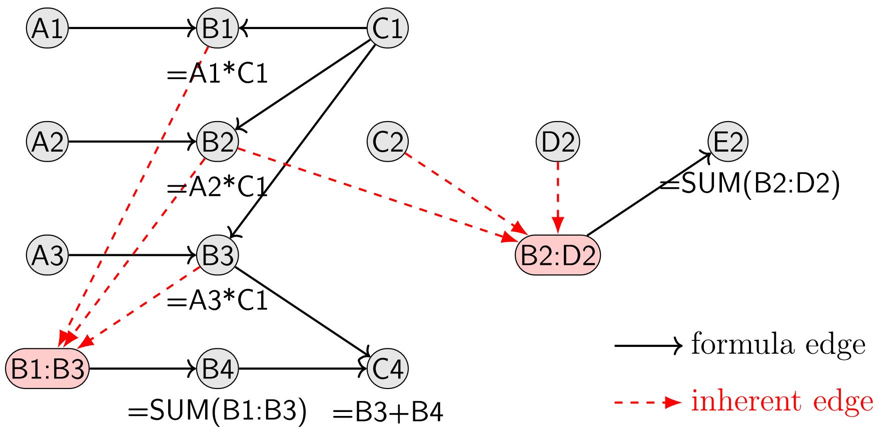


Figure 4.4: Dependency graph capturing dependencies between regions thus reducing the graph size.

B1 and *(ii)* from C1 to B1. Rather than storing the inherent edges explicitly, which can be expensive, we can infer these edges from the cell and ranges they represent. To enable efficient lookups for inherent edges, we can use a spatial index, such as R-tree [53]. To find outgoing edges from a cell c, we can issue a query to the R-tree to find all ranges containing c. For example, to infer the outgoing edges from B2, we can search for all the nodes that overlap with B2—for B2 we have B1:B3 and B2:D2.

Challenges With Dependency Traversal. The lookup of dependencies by traversing a full dependency graph takes time proportional to the number of dependencies, which is inefficient when the number of dependencies is large. Consider the scenario depicted in Figure 4.5—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 4.3 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.

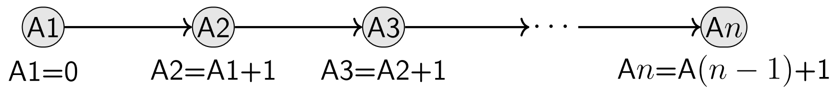


Figure 4.5: Long Dependency Chain

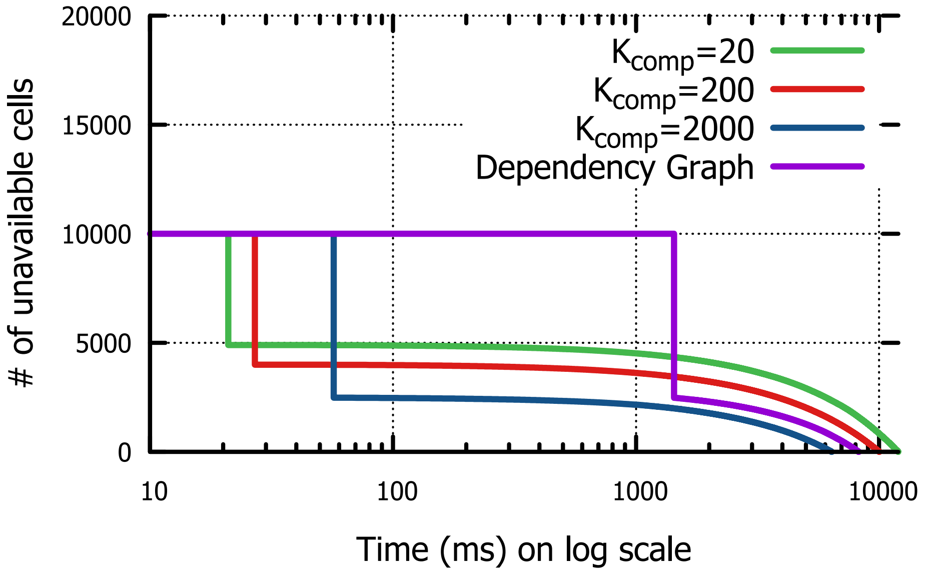


Figure 4.6: Comparing unavailability for by using dependency graph vs dependency table with varying K_{comp}.

4.2.2 Compressed Dependency Table

To overcome the aforementioned challenge, we propose an alternate manner to capture dependencies. In addition to the dependency graph, we maintain a "cache" of dependents for each cell, in a *dependency table*—see Figure 4.7(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.

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

(a)

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

(b)

Figure 4.7: Compressing dependency table to bound the number of dependents: (a) original before compression (b) after compression with $K_{\text{comp}} = 2$.

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 a recalculation of c_{FP}, whose value will remain the same. In other words, the dependency table is *false positive tolerant*—the presence of a false positive does not affect correctness, but can cause unnecessary calculations. On the other hand, a "false negative", a cell c_{FN} that is an actual dependency of u but is missing from the table, is unacceptable, because a update to u would not trigger a recalculation of c_{FN}, leading to a possibly incorrect value for c_{FN}.

A *compressed dependency table*, or CDT for short, is a variation of a dependency table that enables identifying dependencies in $O(1)$ time—see Figure 4.7(b). As ranges naturally represent a group of cells, we express the dependents in a compressed dependency table as ranges. For example, dependents of C1 can be expressed as B1:B3, B4:C4, E2 with no false positives, or as B1:C4, E2 with three false positives (C1, C2, C3). For a set of cells C to be expressed as a set of regions R, we require that the regions in R can collectively "cover" the set C. We formalize the notion of a cover as follows.

Definition 4.9 (Cover)**.** *For a set C of cells, a set $R = \{R_1, \ldots, R_m\}$ of ranges is a* cover *of C if $C \subseteq R^{\cup}$, where $R^{\cup}$ denotes the set of cells that are in at least one of the ranges $R_1, \ldots, R_m$. The size of the cover R, denoted by* $\mathrm{size}(R)$*, is $|R|$. The cost of the cover R, denoted by* $\mathrm{cost}(R)$*, is $|R^{\cup}|$.*

To ensure that dependents of a cell u can be retrieved and reported in constant time, we limit the size of the cover to a constant K_{comp}. In Figure 4.7(b), the K_{comp} is 2. Varying the value of K_{comp} can significantly impact the unavailability, due to the following: there is a trade-off between the time it takes to perform dependency identification (t_{dep} in Figure 4.3) and the number of cells that remain when dependency identification is complete (Δ_u in Figure 4.3). This trade-off is because the less time we spend identifying dependencies, the smaller the K_{comp} and the more false positives we introduce into the dependency table. This increase in false positives causes the cost of the cover, and therefore the total number of dirty cells at the time immediately following dependency identification, to increase. Ultimately, we need a value of K_{comp} that minimizes unavailability.

Illustrative Experiment 2: Impact of K_{comp}. In this experiment we quantitatively demonstrate the benefit of using a dependency table instead of a traditional dependency graph using unavailability. Additionally, we also demonstrate the impact of varying K_{comp} for the dependency table. We consider a synthetic spreadsheet having 10,000 cells out of which 5,000 cells contain formulae. Out of the 5,000 formulae cells, 50% of the cells are dependent on a cell u, which we intersperse with cells that are independent of u. For this synthetic spreadsheet, we update the value of u and plot the number of unavailable cells on the y-axis with respect to time on the x-axis for the asynchronous computation model—see Figure 4.6. Due to a large number of dependencies, dependency identification using the dependency graph (in purple) takes a significant time of 1.4 seconds. The three remaining curves show the benefit of using a dependency table—here, we vary K_{comp} and observe its impact on the time for identifying dependencies. At one extreme, we have the blue curve where K_{comp} is 2,000—the dependency identification takes around 60 ms. On the other hand, the green curve, when K_{comp} is 20, remains in the dependency identification step for very little time (20 ms). However, to compress all of the dependents of a cell into 20 regions, the number of false positives grow to 2,400 cells. Therefore, even though the green curve returns control to the user in a few milliseconds, it takes more time to clean all the dirty cells. In this example, $K_{\text{comp}} = 2{,}000$ (in blue) performs the best under the unavailability metric, as its curve encloses the least area.

Takeaway 4.2. *Dependency table with lossy compression of dependencies bounds the time for which user interface is unresponsive.*

4.2.3 Construction of the Compressed Dependency Table

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

Problem 4.2 (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 4.2 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 4.1.

Theorem 4.1. *The decision version of* Dependents Compression *is* NP-Hard.

We shall prove Theorem 4.1 by providing polynomial-time reduction from the Polygon Exact Cover problem, a known NP-Hard problem, defined as follows [54].

Problem 4.3 (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. Given a simple and holeless orthogonal polygon P and an integer k in a Polygon Exact Cover instance, perform a rank-space reduction on the coordinates of P; that is, change the actual coordinates into values in $\{1, \ldots, n\}$, where n is the size (number of vertices) in P, such that the coordinates are in the same order. Translate the polygon in the new coordinates into a set C of cells. (Note that the representation size goes up quadratically.) There is a set of at most k axis-aligned rectangles whose union is precisely P if and only if C has a cover whose size does not exceed k and cost is at most $|C|$ (has no false positives), following the natural coordinate mapping between rectangles and ranges.

Greedy Heuristic. Since efficiently finding the best compression is hard, we propose a greedy algorithm for graph compression: while the number of ranges representing dependents of a cell exceeds K_{comp}, two of those ranges are selected and replaced by the smallest range enclosing them; repeat until the number of ranges reduce to K_{comp}. We can use various heuristics for selecting the two ranges to combine. One such simple heuristic is to select two ranges such that replacing them with their enclosing range introduces the fewest false

Algorithm 4.1 Incremental Greedy Compression

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

Output: a cover R' of the union of rectangular regions in R, where $|R'| \leq k$

```
1: $R' \leftarrow R$
2: while $|R'| > k$ do
3:     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.
4:     Let $r$ be the smallest of such a bounding box.
5:     $R' \leftarrow (R' - \{r_1, r_2\}) \cup r$
6: end while
7: return $R'$
```

positives, which, as we will see, does well in practice. Note that due to the incremental nature of our compression algorithm, we can use it for the maintenance of the dependency table when we add a new dependency, as we will see next. The pseudocode for the greedy compression algorithm is given as Algorithm 4.1.

4.2.4 Maintenance of the Compressed Dependency Table

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

Deleting dependencies. Deleting a dependent from the dependency table can potentially introduce false negatives. To illustrate this, consider the example provided in Figures 4.4 and 4.7. Here, C4 is a dependent of B3. Suppose the formula in C4 is changed to =B4+3, and thus the direct dependency B3 $\rightarrow$ C4 is deleted. However, we cannot remove C4 from the dependent list of B3, because C4 remains a dependent of B3, albeit no longer a direct one. In other words, the dependency between C4 and B3 is due to more than one formula. Another issue is deleting a single dependent cell from one represented by a range, which is difficult to do efficiently without leading to a highly fragmented, inefficient R-tree.

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

Adding dependencies. Adding a direct dependency can be quite time-consuming in the worst case. Consider the example in Figure 4.8, where the formula of B1 is changed from =0 to =A3+1, and thus a new direct dependency A3 $\rightarrow$ B1 is added. Because of this change, A3 and its precedents must have their entries changed in the dependency table by adding B1, B2, B3 as their dependents, which is quite time consuming.

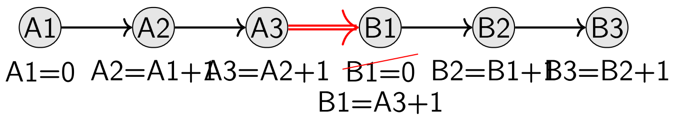


Figure 4.8: Adding a new direct dependency in a dependency chain.

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

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

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

(a)

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

(b)

Figure 4.9: Adding dependencies to dependency table: (a) naïve method (b) lazy dependency propagation (must-expand dependencies are marked by asterisks)

For example, instead of updating all entries as in Figure 4.9(a), B1 is added as a must-expand dependent of A3, as in Figure 4.9(b). At this unstable state, to identify dependents of A1, it is insufficient to just report A2, A3 as dependents, even if neither of the cells *are* must-expand dependents. Since A3 *has* a must-expand dependent B1, the recursive lookup continues, to include B2 and B3. Since neither of the dependents of B1 (which are B2 and B3) has a must-expand dependent, recursion can stop there. Eventually, the must-expand dependent is resolved by a background thread and the dependency table becomes similar to that shown in Figure 4.9(a).

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

Note that adding dependents to a cell can push the number of dependents beyond the K_{comp} limit. To ensure constant lookup time when the dependency table leaves the unstable state, we reduce the number of ranges representing the dependents down to K_{comp} using the method of repeated merging of ranges described in Section 4.2.3.

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

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

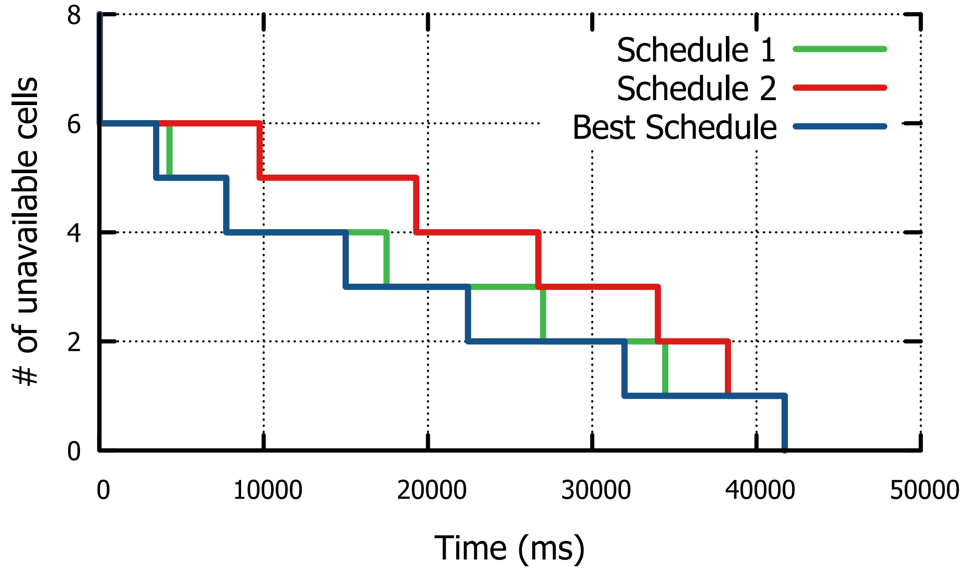


Figure 4.10: Unavailability varying the computation schedule (dependency identification time of 20 ms is not pictured on the graph)

Illustrative Experiment 3: Computation Schedule. The goal of this experiment is to demonstrate the impact of scheduling. Here, we consider a synthetic spreadsheet with six formula cells. The formulae perform summation using the SUM function of varying sized ranges to simulate varying complexities. In this case, as the complexity of a formula in increases, the time to compute it increases as well. These formula cells are independent of each other but dependent on a cell u. For this sample spreadsheet, we update the value of u and plot the number of formula cells that are unavailable on the y-axis with respect to time on the x-axis—see Figure 4.10. Even though the total time required to complete cleaning all the cells is the same across all possible schedules (around 40,000 ms), the time spent by each cell in the dirty state varies, which impacts unavailability. Schedule 1 and 2 adopt a random schedule, and thus differ in terms of unavailability. The best schedule computes the cells in the increasing order of complexity, thereby minimizing unavailability.

Takeaway 4.3. *Computation scheduling is important within the asynchronous computation model and impacts the number of cells that are available to users over time.*

4.3.1 Motivation and Problem Statement

The computation scheduling problem naturally arises from the idea of partial results (Section 4.1.3): if we are displaying the computed cell values to the user as we finish computing them, in what order should we compute cells? For our computation scheduling problem, we define $\text{cost}(c)$ to quantify the time taken for computing a cell c. For now we assume a simple independent computation model where we ignore the impact of caching cells ; we will discuss its impact later and relax this assumption. Their formal definitions are as follows.

Definition 4.10 (Cost)**.** *The cost of a cell c, denoted by* $\text{cost}(c)$*, is the amount of time needed to compute the evaluated value of c, assuming the values of its precedents are already computed.*

Assumption 4.1 (Independent Computation Model)**.** *We assume that the cost of computing a cell c, i.e.,* $\text{cost}(c)$*, is independent of the computation schedule. In other words, c takes the same time to evaluate, regardless of when we compute c.*

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

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

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

Definition 4.11 (Dependency Constraint). *A computation order $c_1, \ldots, c_n$ of cells is valid only if the following holds: if $i < j$, then c_i is not a dependent of c_j.*

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

Definition 4.12 (Dirty Time with respect to a Schedule). *In a computation order $c_1, \ldots, c_n$, the dirty time for the cell c_i is $\text{dirty}(c_i) = \sum_{j=1}^{i} \text{cost}(c_j) = \text{dirty}(c_{i-1}) + \text{cost}(c_i)$.*

We formalize our scheduling problem as follows, which is shown as NP-Hard by Lawler [55].

Problem 4.4 (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.*

4.3.2 On-the-fly Scheduling

In addition to the fact that Computation Scheduling is NP-Hard, upon updating u, merely obtaining a schedule can be prohibitively expensive. The dirty time defined in the previous subsection (Definition 4.12) only takes into account computation time, but not the time to perform the scheduling itself. If there are n dirty cells in Δ_u, then the amount of time to obtain any complete schedule satisfying the dependency constraints is $\Omega(n)$, as each of the n cells must be examined at least once to determine dependency and the cost of computation. If the scheduling algorithm takes time t_s, then performing scheduling up front increases the dirty time of each cell in Δ_u by t_s, and no progress towards their computation is made during that time. Such an effect is undesirable and potentially negates any gains from incrementally computing and showing results to the users.

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

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

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

The on-the-fly scheduling heuristic attempts to postpone computing high cost cells for as long as possible, because computing low cost cells first allows for more results to be quickly

Algorithm 4.2 On-the-fly Scheduling algorithm

```
Input: a set dirty cells $\Delta_u$, and two integers $k$ and $m$
Output: a computation schedule of the cells in $\Delta_u$
 1: $D \leftarrow \Delta_u$
 2: $P \leftarrow \emptyset$
 3: while $|D| > 0$ do
 4:     $P' \leftarrow$ subset of $k - |P|$ cells drawn from $D$
 5:     $D \leftarrow D - P'$
 6:     $P \leftarrow P \cup P'$
 7:     Compute cost of each element in $P$.
 8:     $M \leftarrow$ the $m$ elements of $P$ with lowest cost
 9:     $P \leftarrow P - M$
10:     Let $M'$ be the union of the dirty precedents of $c$ for $c \in M$
11:     Append $M \cup M'$, in topological order, to $S$
12: end while
13: return $S$
```

shown to the user. In fact, without dependency requirements, scheduling computation in increasing order of cost yields the optimal schedule [56]. Our heuristic is based on the same principle, but adapted to obey the dependency constraint and to make decisions without looking at the entire workload. The pseudocode for the on-the-fly scheduling procedure is given as Algorithm 4.2.

4.3.3 Weighted Computation Scheduling

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

Since users can only view the cells that are within the viewport, it is desirable to prioritize the computation of cells that the user is currently viewing—for this purpose we introduce a weighted variation of unavailability. Here, each cell c is given a *weight*, denoted as weight(c). The more important a cell is, the higher its weight. For example, we can prioritize computation of cells in the viewport by assigning a high weight, $w \gg 1$, to cells within the viewport and a low weight, 1, to other cells. It may also be desirable to assign a medium weight to cells just outside the viewport, as scrolling to these cells is likely. The following formalization of a *weighted unavailability* modifies our previously defined unavailability (see Definition 4.8) such that if a high-weight cell is left dirty, and therefore unavailable, for an extended period, the metric's value is much higher.

Definition 4.13 (Weighted Unavailability)**.** *The weighted unavailability W_M for a computation model M over a spreadsheet S is*
$W_M = \sum_{c \in S}(\text{weight}(c) \cdot \text{unavailable}(c))$, *where* $\text{weight}(c)$ *is the weight of c,* $\text{unavailable}(c)$ *is the unavailable time for c, and S is the set of all cells within the spreadsheet.*

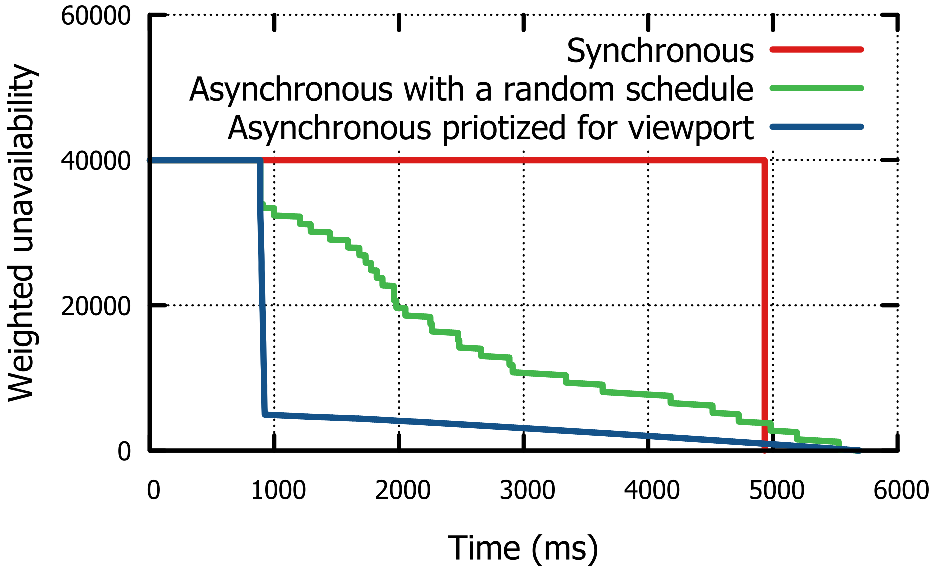


Figure 4.11: Weighted unavailability comparing synchronous computation models and asynchronous computation model with and without viewport prioritization.

4.3.4 Weighted Computation Scheduling

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

Problem 4.5 (Weighted Computation Scheduling)**.** *Given a set of dirty cells (Δ) along with their weights and the dependencies among them, determine an order $c_1, \dots, c_n$ of all the* *cells in* Δ *that minimizes weighted unavailability, i.e.,* $\sum_{c_i \in \Delta}(\text{weight}(c_i) \cdot \text{dirty}(c_i))$*, where* $\text{dirty}(c_i) = \sum_{j=1}^{i} \text{cost}(c_j) = \text{dirty}(c_{i-1}) + \text{cost}(c_i)$ *and* $\text{weight}(c)$ *is the weight of* c*, under the* DEPENDENCY *constraint.*

WEIGHTED COMPUTATION SCHEDULING is trivially NP-HARD, since it is a generalization of COMPUTATION SCHEDULING discussed in Section 4.3.1, which is NP-HARD.

Illustrative Experiment 4: Weighted Scheduling. This experiment demonstrates a weighted variation of Experiment 1, with Figure 4.11 showing a weighted variation of Figure 4.3. Here, we assign a weight of 1,000 for 30 formula cells within the user's viewport and 1 for the remaining. We plot time on the x-axis and weighted unavailability (the product of the number of unavailable cells and their weights) on the y-axis. Past the 890 ms mark, the red curve, which represents the synchronous model, maintains the same level of weighted unavailability until all of the cells have been computed and marked clean at around 5,000 ms. For the asynchronous model (in blue) that prioritizes cells in the viewport when scheduling, the weighted unavailability drops off very quickly between 890 ms and 1,000 ms, and then slowly decreases to 0 afterwards. This sharp decline represents the time when the system is computing the highly-weighted cells within the viewport. The remaining, lower-weighted cells outside the viewport are computed afterwards. On the other hand, the asynchronous computation model that uses random scheduling, slowly decreases over time, as high-weighted cells are left in the dirty state due to randomized scheduling. As can be clearly seen in Figure 4.11, the model which prioritizes cells in the viewport when scheduling performs the best under weighed availability.

Takeaway 4.4. *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 4.3.2 by updating the cost calculation to additionally consider the weight of the cell. Intuitively, we would like to prioritize cells that have a higher weight but a lower cost. Thus, in Algorithm 4.2, we sort the cells by ${}^{cost(c)}/_{weight(c)}$, where $cost(c)$ is the computation cost for c and $weight(c)$ is the weight that

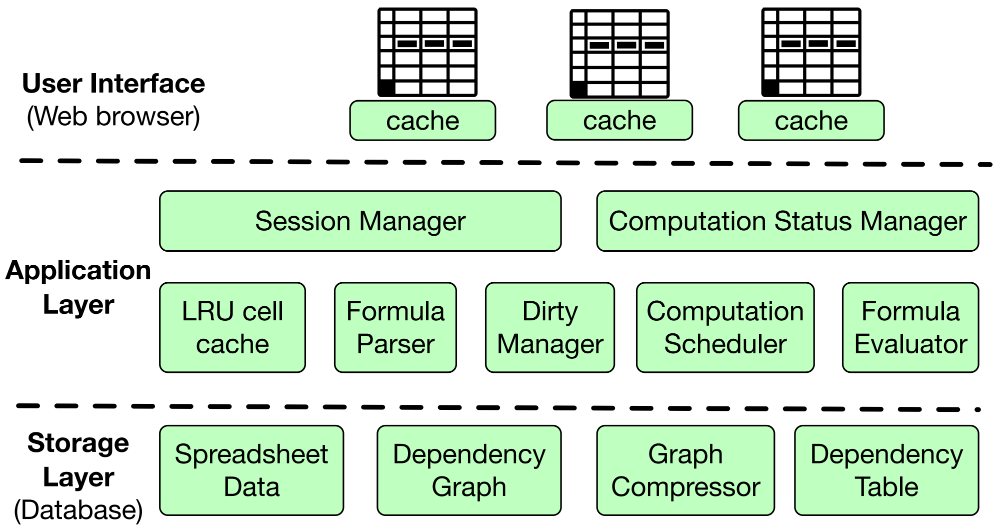


Figure 4.12: DataSpread's Formula Computation Architecture

we assign to c. Additionally, we dynamically update the cell weights when the user changes their viewport by scrolling. Further more we can also modify Algorithm 4.2 to first pick up the cells that are within the viewport.

4.4 SUPPORTING ASYNCHRONOUS COMPUTATION IN DATASPREAD

In this section, we discuss the incorporation of our asynchronous computation model within DataSpread. In this chapter, until this point, our focus was on main-memory setting for compatibility with existing spreadsheet tools. However, as we discussed in Chapter 3, to handle spreadsheets at scale, DataSpread adopts a two-tiered memory model where it persists the spreadsheet data in the back-end database and fetches it *on-demand* when triggered by a user action (like scrolling) or a system action (like calculating a formula). In Section 4.4.2, we discuss how the problems discussed in this chapter change when we consider memory model such as one used by DataSpread, a *two-tiered memory model*, and how the techniques can be adapted to account for different cost considerations.

4.4.1 Formula Computation Architecture

Figure 4.12 illustrates DataSpread's formula computation architecture. The front-end back-end communication designed using the Spring [57] 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 such as cell computation.

Partial Result Presentation Components. As discussed in Section 4.1, 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 4.3.3.

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

Formula Evaluation Components. The *dependency graph* maintains dependencies between cells and regions. The compressed *dependency table*, maintained by the *dependency table compressor*, allows the system to support fast dependency identification, as discussed in Section 4.2.

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

The *computation scheduler* coordinates the formula computation. It retrieves dirty cells from the *dirty manager* and schedules their computation as discussed in Section 4.3. 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.

4.4.2 Handling Scale

While working with large datasets, the main-memory based design of current spreadsheet systems fundamentally limits their scalability [58]. Thus, to achieve interactivity at scale, systems must operate beyond main memory limits. In this section, we discuss a *two-tier memory model*, wherein the cost of data retrieval from storage factors into the unavailable time. Unlike the previous chapter where two tiers were handlied by the backend database, here we need to explicitly decide how and where to do computation.

We discussed the asynchronous computation techniques in the earlier sections in the context of main-memory systems, wherein computation time is the dominant concern. These techniques still provide a significant improvement in a two-tier setting, but can be further improved if fetching costs are taken into account. This section discusses how we adapt the techniques described in the earlier sections to work with these additional cost concerns.

Two-tier Memory Model

We define the two-tier memory model as follows:

Definition 4.14. *The* two-tier memory model *contains two tiers of memory:*

- *the* main-memory*, which is limited in size, but allows fast data access; the application interfaces with this tier;*

- *the* storage*, which is large, but data access is slow; and the application does not directly interact with this tier.*

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

Techniques under Fetching Cost

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

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

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

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

Problem 4.6 (Computation Scheduling with Fetching Costs)**.** *Given a set of dirty cells (Δ), their direct precedents*
$P = \{p \mid$ the direct dependency $p \to c$ 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., $\sum_{c_i \in \Delta} \mathrm{dirty}(c_i)$, where $\mathrm{dirty}(c_i) = \sum_{j=1}^{i} \mathrm{cost}(c_j) = \mathrm{dirty}(c_{i-1}) + \mathrm{cost}(c_i)$, under the Dependency *constraint.*

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

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

Theorem 4.2. Computation Scheduling with Fetching Costs *is* NP-Hard.

Problem 4.6 is a generalization of the Scheduling With Supporting Tasks problem [59], which is NP-Hard, defined as follows.

Problem 4.7 (Scheduling With Supporting Tasks)**.** *Let $A = \{a_1, \ldots, a_m\}$ and $B = \{b_1, \ldots, b_n\}$ be sets of tasks, and $R \subseteq A \times B$ be a relation. All tasks take a unit time to complete. The tasks in A and B are to be scheduled on a single machine that can perform one task at a time, under the restriction that if $(a_i, b_j) \in R$, then task a_i must be completed before task b_j. Tasks in A are* supporting tasks *and are not required to be completed (unless required by other tasks in B). Let $\mathrm{cost}(b_j)$ 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 $\sum \mathrm{cost}(b_j)$ is at most c.*

Proof of Theorem 4.2. To prove the theorem, we provide a polynomial-time reduction from Scheduling With Supporting Tasks. For each task t in $A \cup B$, create a corresponding cell cell(t). For each $(a_i, b_j) \in R$, make cell(b_j) dependent on cell(a_i); in other words, if $a_{i_1}, \ldots, a_{i_\ell}$ are the elements of $\{a \mid (a, b_j) \in R\}$, create a formula cell($b_j$) = cell($a_{i_1}$) + $\cdots$ + cell(a_{i_ℓ}). 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.

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

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

4.5 ADDITIONAL EXPERIMENTS

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

Environment. We have implemented the asynchronous model along with graph compression and computation scheduling within our scalable spreadsheet system, DataSpread, which uses PostgreSQL 10.5 as a backend data store. We run all of our experiments on a

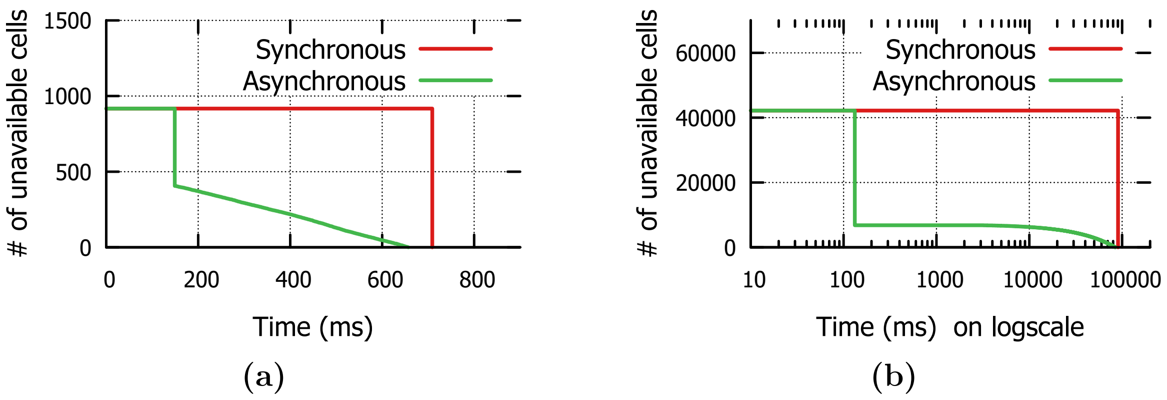


Figure 4.13: Unavailability comparing synchronous and asynchronous computation models for two real world spreadsheets of different sizes. (a) small (b) large

workstation running Windows 10 on an AMD Phenom II X6 2.7 GHz CPU with 16 GB RAM. While we have a functional prototype, to eliminate the impact of communication between front-end and back-end, we design our test scripts as single threaded independent applications that directly utilize DataSpread's back-end.

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

Illustrative Experiment 5: Real-world Spreadsheets. In this experiment, we use two real-world spreadsheets to compare the two computation models. For both spreadsheets, we find a cell u that has the highest number of dependents. The first spreadsheet has complex financial calculations, with a total of 917 cells, out of which 406 formula cells are dependent on u. The second spreadsheet is targeted towards inventory management and had a total of 42,181 cells, out of which 6,803 formula cells are dependent on u. We use a naïve synchronous computational model as described in Experiment 1. For the asynchronous model, we use a compressed dependency table with $K_{\text{comp}} = 5000$ (Section 4.2.2) and schedule computations on-the-fly (Section 4.3.2). We update u and plot the number of unavailable cells on the y-axis with respect to time on the x-axis—see Figure 4.13. Observations are similar to Experiment 1. In terms of unavailability, for the asynchronous model, we see an improvement of 2x and 12x over the synchronous model for the first and second spreadsheet respectively, thus confirming the applicability of our ideas to real-world spreadsheets.

Takeaway 4.5. *Our proposed asynchronous computation model maintains interactivity and consistency thus improving usability of spreadsheet systems for large and complex spreadsheets.*

4.6 RELATED WORK

The asynchronous formula computation model presented in this chapter is a novel alternative to the synchronous model adopted by traditional spreadsheet systems. Problems similar to graph compression and scheduling are studied in various contexts with different goals and constraints. We now discuss in more detail each of these categories of related work.

Computation Models. Asynchrony has been used in operations with delayed actors, such as in crowdsourcing [60, 61] and web search [62] but never for spreadsheets—their concerns and objectives are very different. The synchronous model of traditional spreadsheets utilize the idea of dependency graph [52, 63, 64] to avoid unnecessary computations, but it does not avoid the performance degradation due to large and complex dependencies [65]. Our proposed asynchronous computation model along with compressed dependency table alleviate such issues as discussed in Section 4.2.

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 [66], those that obtain a concise representation of graphs to improve query performance are related to our work. Graph compression methods, surveyed by Liu et al. [67], have different focuses, such as compactness with bounded errors [68], pattern matching queries [69], and dynamic graphs [70]. Our setting is different from these works because of *(i)* our goal of quickly obtaining a representation of dependents of a cell, *(ii)* the one-sided (false positive only) tolerance, and *(iii)* the spatial nature of cell ranges.

Scheduling. Scheduling under precedence constraints is a thoroughly studied problem, especially in operations research, with various settings and metrics, including ones similar to the unavailability metric [55]. Similar scheduling problems arise in this chapter, and some hardness results are drawn from previous work. However, as discussed in Section 4.3, in the prior work, schedules are built up front, whereas obtaining a complete schedule up front is prohibitive in our setting. For this reason, we introduce on-the-fly scheduling.

4.7 CONCLUSIONS

Our proposed asynchronous computation model improves the interactivity of spreadsheets without violating the consistency while working with large datasets. To support asynchrony without violating consistency, we introduced the idea of partial results, which blurs out the formulae that are being computed in the background. We ensured interactivity by proposing a compressed dependency table to identify dependent cells after a cell update in a bounded amount of time. For usability, we developed an on-the-fly scheduling technique to minimize the number of cells that are pending computation. We have implemented the aforementioned ideas in DataSpread and demonstrated improved interactivity compared to traditional spreadsheet systems. Thus, our new computation model's improved interactivity allows the use of spreadsheet systems in data analysis situations where it was once inconceivable.

CHAPTER 5: RELATIONAL SCHEMA DESIGN

In this chapter, we focus on the problem of quantitatively designing an optimal relational schema for a workload. The discussion in Chapter 3 focused on representation of *presentation data* within a relational database, which had a unique challenge of capturing the positional information along with the data. In this chapter, we complement the discussion by focusing on optimizing the schema of relational tables stored in the database. The challenges here are orthogonal to that of spatial data, specifically in addition to the exponential search space, we need to develop means to describe a schema's requirements precisely.

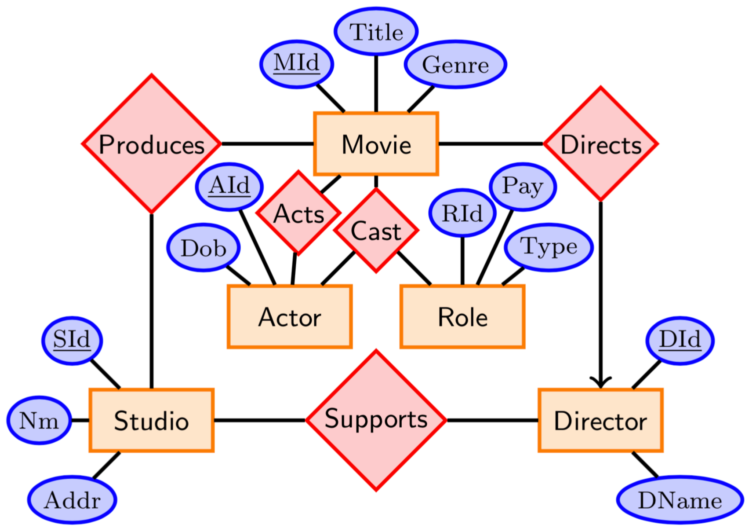
Relational schema design is an essential problem for DataSpread to represent tabular data within a database efficiently. As discussed in Chapter 2, in DataSpread we enable users to persist tabular data on the spreadsheet interface as relational tables within the underlying database. On a spreadsheet interface, a user embedded formulae in the form of queries along with tabular and non-tabular data. To ensure the interactivity of the interface, efficient execution of the queries is a necessity. Furthermore, the spreadsheet interface acts as an abstraction isolating the users from the underlying database. Thus, it is possible to optimize the schema of the relational tables stored in the underlying database in a transparent manner while still being able to service the queries and formula. Additionally, schema design relives users from the burdon of obtaining a schema based on an application's requirements. Data organized on a spreadsheet in the form of tables [71, 72] enables us to identify the associations and constraints—thus enabling us to obtain the application's requirements. The formulae on spreadsheets serves as workload for quantitatively designing an optimal schema.

Quantitatively designing an optimal relational schema is also important in a generic database setting. With the advent of interactive data-driven applications everywhere, our databases, which were originally designed for ad-hoc querying, are now facing a new scenario. Rather than writing queries, users satisfy their information needs by interacting with websites, *e.g.*, IMDb [73]. For such user-facing, interactive, and high volume websites response time is crucial for its usability [74] and success [75]. Since the response time of such data-driven websites usually depends on the underlying database, improving the database's performance provides an instant boost for such applications.

Among all the factors that determine the performance of a database system, a *schema* is an important one in physical design—which has not been fully explored and quite contrast to traditional metrics. A schema impacts a database's response time as it influences the effort required answer queries. To motivate this, we consider query Q_1 (Figure 5.4), where the database performs a *join*, to gather the needed attributes, for each execution. If we avoid the join, say by merging the tables, it usually improves the query's performance. However, such a merge might not always be beneficial as a query that accesses the merged table has to deal with duplicated tuples, and increased record size resulted due to the merge. While query performance is crucial, most traditional schema design works focus on obtaining an intuitive schema [76] and reducing redundancy [77].

The trend of databases as invisible "back-end" demands high performance from a schema much more than the normalization or intuitiveness as traditionally emphasized. In the now ubiquitous website and application-driven scenarios, a database's workload as a "back-end" is rather well-defined—each page in a website is rendered using a fixed set of queries. Further, based on a user's purpose, he follows a series of "prescribed" actions [78]. Such navigation patterns lead to a well-defined workload that characterizes the site's usage. Meanwhile, such well-defined workloads also mean much less ad-hoc interaction with the databases (*e.g.*, by command-line SQL), and thus schemas are much less directly exposed to database users. We see an opportunity to boost the performance of a database by designing its schema optimized for its application workload.

Thus, *can we design a schema automatically*—from only a *conceptual model* such as ER or a "schema description", to search for a *quantitatively optimal* schema? Later in this chapter we advocate and formalize how we faithfully describe a schema. While the notion of schemas has been essential from the dawn of relational databases [77], and the study of schemas has been vast, this question is largely open. As Section 5.6 will review, most schema normalization (for normal forms) or optimization (for query performance) techniques require an initial schema as input—and thus solve a fundamentally different *transformation* problem, instead of a *design* problem. They start with a physical schema, which already captures the "requirements" of information storage, and transform it to make it better for



- MId $\rightarrow$ Title
- MId $\rightarrow$ Genre
- SId $\rightarrow$ Nm, Addr
- DId $\rightarrow$ DName
- AId $\rightarrow$ Dob
- RId, Type $\rightarrow$ Pay
- DId $\rightarrow$ Genre

Figure 5.1: Conceptual data modeling—ER model $\mathcal{M}$.

some criteria. The reliance of an initial schema is a major shortcoming—Stonebraker in his lecture notes [79] points out that the output of normalization is limited by the starting schema, and asks *how to come up with the initial set of tables?* The reliance on such initial schemas significantly limits the search space—since we only "transform" them—for finding a quantitatively optimal design. This paper aims to qualitatively *design* a physical schema, given a conceptual model—which to the best of our knowledge, is a novel and first attempt—Section 5.1 formulates the problem.

Quantitative schema design is challenging due to many unaddressed issues. *What does a conceptual ER model convey, for a physical schema to carry out?* We identify the notion of *essential associations* as intended by a conceptual model, as the requirements for schema design. *What is a* correct *schema that we must consider, given an ER model?* We identify a correct schema as one that preserves *all* and *only* the "essential associations" intended by the conceptual model, and define the search space accordingly. Based on these findings, we can develop a framework that searches for best schemas in a space induced by only a conceptual model.

Challenge. Quantitative schema design is challenging due to its EXPTIME complexity compounded with a layer of query optimizer to determine the cost. Furthermore, existing schema modeling techniques do not faithfully capture a schema's requirements; this is crucial

- $A_1 : \{\text{MId}, \text{Title}, \text{Genre}\}$
- $A_2 : \{\text{SId}, \text{Nm}, \text{Addr}\}$
- $A_3 : \{\text{DId}, \text{DName}\}$
- $A_4 : \{\text{AId}, \text{Dob}\}$
- $A_5 : \{\text{RId}, \text{Type}, \text{Pay}\}$
- $A_6 : \{\text{MId}, \text{DId}\}$
- $A_7 : \{\text{MId}, \text{SId}\}$
- $A_8 : \{\text{SId}, \text{DId}\}$
- $A_9 : \{\text{MId}, \text{AId}, \text{RId}\}$
- $A_{10} : \{\text{MId}, \text{AId}\}$
- $A_{11} : \{\text{DId}, \text{Genre}\}$

Figure 5.2: Associations from conceptual data modeling $\mathcal{A}$.

Schema S_0	*Schema* S_1	*Schema* S_2
• Movie($\underline{\text{MId}}$, Title, Genre, DId) • Director($\underline{\text{DId}}$, DName) • Studio($\underline{\text{SId}}$,Nm, Addr) • Actor($\underline{\text{AId}}$, Dob) • Role($\underline{\text{RId}}$, Type) • Produces($\underline{\text{SId}}$, $\underline{\text{MId}}$) • Supports($\underline{\text{SId}}$, $\underline{\text{DId}}$) • Cast($\underline{\text{AId}}$, $\underline{\text{RId}}$, $\underline{\text{MId}}$)	• Movie($\underline{\text{MId}}$, Title, Genre, DId) • Director($\underline{\text{DId}}$, DName) • Studio($\underline{\text{SId}}$, Nm, Addr) • Actor($\underline{\text{AId}}$, Dob) • Role($\underline{\text{RId}}$, Type) • Produces($\underline{\text{SId}}$, $\underline{\text{MId}}$) • Cast($\underline{\text{AId}}$, $\underline{\text{RId}}$, $\underline{\text{MId}}$)	• MovieDir($\underline{\text{MId}}$, Title, Genre, DId, DName) • Studio($\underline{\text{SId}}$, Nm, Addr) • Actor($\underline{\text{AId}}$, Dob) • Role($\underline{\text{RId}}$, Type) • Produces($\underline{\text{SId}}$, $\underline{\text{MId}}$) • Cast($\underline{\text{AId}}$, $\underline{\text{RId}}$, $\underline{\text{MId}}$)

Figure 5.3: Some possible schemas for input ($\mathcal{M}$, $\mathcal{F}$).

for understanding the search space of schema design as for correctness, we need to evaluate every candidate from the space because by counter argument an unevaluated one can be optimal. The traditional query cost models [80], which estimate a query's cost for candidate execution plans, cannot obtain a schema's cost without an execution plan making them unsuitable for cost speculation, without which every schema needs to be evaluated.

Contribution. Our contributions are: *(i)* We propose a novel concept of attribute association along with relevant theory to describe a schema's functional requirements. *(ii)* We provide a significant contribution for schema design by encompassing the major performance related schema transformations, *i.e.*, partitioning and merging. *(iii)* We develop a schema-centric cost model, which is independent of a query optimizer's implementation but works in conjunction with it to predict its behavior while merging tables.

Insight. A query optimizer does not determine a query's cost arbitrarily but rather chooses an execution plan from possible alternatives to minimize the cost of execution. We leverage

Q_1	SELECT Title, DName FROM Movie NATURAL JOIN Director WHERE DName = "Larry"
$E(S_1)$	3. Nested Loop(cost=0.0..34894.0 rows=10 width=41) 2. Seq Scan on director(cost=0.0..2084.00 rows=1 width=4) Filter: (fn = 'Larry'::text) 1. Seq Scan on movie(cost=0.0..20310.0 rows=1000000 width=45)
$E(S_2)$	1. Seq Scan on MovieDir (cost=0.00..26786.00 rows=11 width=41) Filter: (fn = 'Larry'::text)

Figure 5.4: Query Q_1 and its execution plans on S_1 and S_2.

this to speculate, *i.e.*, estimate without evaluation, a query's cost and develop an anti-monotonic property to prune the sub-optimal schemas efficiently. Intuitively, if eliminating a group of joins is helpful then eliminating *some* of those joins should also be useful. This property is similar to the one used by pruning algorithms such as Apriori.

Evaluation. We experimentally evaluate our solution on three datasets with different content types to validate the performance gains. We show: *(i)* a speedup of 1.72, 1.27, and 2.22 across datasets by employing a workload in the design process, *(ii)* an improvement of 40% and 60% over two baselines, *(iii)* the ability of our method to work in conjunction with indexes, and *(iv)* the impact of workload characteristics on performance gains.

5.1 QUANTITATIVE SCHEMA DESIGN

We now concretize our problem of *quantitative schema design.* Defining the problem is *novel*: While the notion of a schema is essential for databases since their inception, the *quantitative schema design* problem—which starts from only conceptual requirements and constructs a concrete schema and finds the best schema with a quantitative, instead of qualitative, objective—has not been formalized and studied.

Defining the problem is also *challenging*, which must consider the rising trends and needs against traditional wisdom. As discussed earlier, our problem is motivated from the new settings where the departure from ad-hoc querying (of arbitrary combinations of attributes)

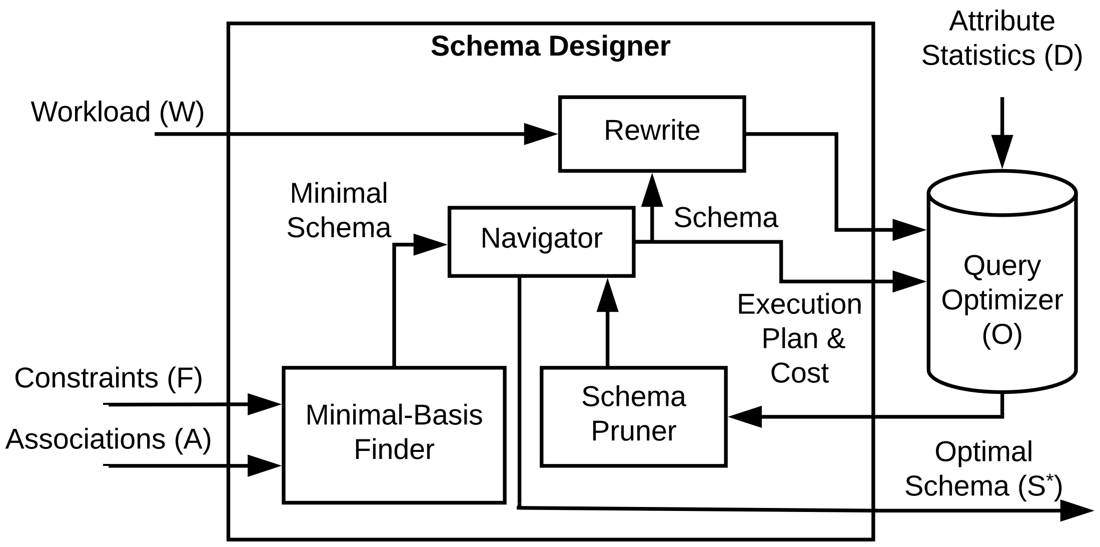


Figure 5.5: Schema design framework.

is typical, the demand for high efficiency is crucial, and the burden of manual schema design is hindering usability.

Schema design for a database application is the process of creating a schema that satisfies the database's requirements and has an optimal performance with respect to its workload. We thus model the problem as a *cost-based optimization* framework: to find, for a database D and workload W, a schema S^* that has the minimum cost, where cost indicates the workload's expected execution time.

Workload W. A workload abstracts a collection of queries that capture the usage of the database to optimize for. We represent a workload as a weighted set of queries, $W = \{(Q_1, w_1), \ldots, (Q_n, w_n)\}$, where weight w_i indicates the importance (frequencies or preferences) of query Q_i. A workload can be collected over time (*e.g.*, for web applications) or in a particular configuration (*e.g.*, when queries are embedded with data such as in a spreadsheet of tables).

Input: Schema Description I**.** As input, users give some *schema description* (including the set of attributes $A_1, \ldots, A_n$ and other characteristics) that constrains what a "correct" schema should be. For example, a *possible* form of description is an ER model of the database application, as Figure 5.1 shows. We advocate and formalize what the input should be in our study.

Output: Optimal Schema S^*. We construct a *schema*, which is the structure of a database that organizes data by use of tables. We formalize the notion of *schema* S as a set of tables $\{T_1, \ldots, T_m\}$, where each *table* T_i is a set of *attributes* A_{ij}, written as $T_i(\underline{A_{i1}}, A_{i2}, \ldots, A_{ik_i})$, with underlined attributes indicating keys. To illustrate, Figure 5.3 shows S_0, a possible schema for our running example (Figure 5.1), which structures information in eight tables. Note that S_0 is a direct translation from the ER model $\mathcal{M}$, following the well-known textbook rules [81, 82]. For example, tables Movie, Director, Studio, Actors, Roles represent the corresponding entities in $\mathcal{M}$, and Produces, Supports, and Cast the relationships (while "Directs" is merged to Movie). As there are numerous possible schemas that satisfy the input (*e.g.*, S_1 and S_2 in addition), our goal of quantitative schema design is to find an *optimal* schema S^* in terms of its *cost* among those *correct* schemas.

With recent observations, we propose to define the new quantitative schema design problem with the following distinct characteristics—which are also challenges we must cope—as distinguished from existing work.

- **C1:** *Schema description* should be *conceptual* and *minimal.* The mandatory input, schema description, should be *conceptual*, without burdening users to design an "initial" physical schema, and *minimal*, capturing only the "absolute" requirements, to maximize the choices of schemas.

- **C2:** *Search space* should be *comprehensive.* To ensure optimality, we should consider a comprehensive search space of schemas, with a formal guarantee of its completeness.

- **C3:** *Objective criteria* should be *quantitative.* To ensure preciseness, we should establish a quantitative objective to measure the performance *w.r.t.* desired workloads.

C1 Schema Description: Associations as Information Units. *What essential schema description I should be required—to be conceptual and minimal?* As the purpose of a database is to store data for applications, the question is then how to describe desired "capacity" in terms of the basic "information units" to store. We propose *associations* of—or mappings between—attributes as such units, and develop a systematic *representation* and *inferencing* framework for their theoretical treatment. While few existing works (*e.g.*, [21, 83]) also identified using associations, surprisingly, the existing mechanisms are limited to *functional* dependencies (*e.g.*, MId and DId is a many-to-one mapping) and largely ignore general *non-functional* associations (*e.g.*, MId and AId is many-to-many)—thus, the following questions are still open: *Given a set $\mathcal{A}$ of associations, can a schema S store it? What is a "minimal basis" $\hat{\mathcal{A}}$ that can imply all other associations in $\mathcal{A}$?*

As our *first* key result (Section 5.2), our study fills in this void: We identify the dual components $I = (\mathcal{A}, \mathcal{F})$ with *target associations* $\mathcal{A}$ with respect to FDs $\mathcal{F}$, as the input to quantitative schema design—conceptual and minimal, where the former describes the information units to be stored, and the latter provides sufficient and necessary constraints to infer equivalence of such storage. We propose a general representation framework in Section 5.2 and complete it with the inferencing mechanism in Section 5.3.

C2 Search Space: Minimal-Basis Spanning. *How to span a correct and complete search space $\mathcal{X}$ from a basic schema description ($\mathcal{A}$, $\mathcal{F}$)?* We need a way to enumerate every correct schema (which is able to store any instances of associations in $\mathcal{A}$ that satisfy FDs $\mathcal{F}$).

As our *second* key result (Section 5.3), we develop a precise condition for a correct schema which preserves desired associations: *Any correct schema must store* at least *a minimal basis $\hat{\mathcal{A}}$ of target associations $\mathcal{A}$.* Adding the no-redundancy requirement (no extra association should be stored), a correct schema must store *exactly* a minimal basis $\hat{\mathcal{A}}$—which leads to a simple algorithm for spanning a correct and complete search space, by "losslessly" joining the associations in such a minimal basis.

C3 Objective Criteria: Optimizer Driven. *How do we estimate the cost of a schema S for a workload W with respect to a database D?* We do not wish to actually execute queries in W over data in D, as it can be costly. There are several interesting issues: First, while we are optimizing schemas, a database itself performs sophisticated query optimization to determine a query plan and estimate its cost. As a *query optimizer* $\mathcal{O}$ dictates what query plans will be run, our schema optimizer should interact with $\mathcal{O}$ as an autonomous black box and *respect* its cost estimation—so we can deploy quantitative schema design to any database systems. Second, at schema design time, depending on applications, the database may not be fully populated. Fortunately, a query optimizer only needs data statistics instead of actually materialized tables. Thus, we represent a database $D = (\mathcal{D}, \mathcal{O})$ with its query optimizer $\mathcal{O}$ and statistics $\mathcal{D}$, and we assume it will return a cost estimate $\mathrm{Cost}_{\mathcal{D},\mathcal{O}}(Q_i, S)$ for each query Q_i, and the workload cost is then $\mathrm{Cost}_{\mathcal{D},\mathcal{O}}(W, S) = \sum_{i=1}^{n} w_i \cdot \mathrm{Cost}_{\mathcal{D},\mathcal{O}}(Q_i, S)$.

However, using an optimizer $\mathcal{O}$ of a target database system as the quantitative estimator is non-trivial: It is prohibitively expensive to invoke $\mathcal{O}$ for every schema S in $\mathcal{X}$ and every query Q_i in W to estimate $\mathrm{Cost}_{\mathcal{D},\mathcal{O}}(Q_i, S)$—the search space is exponentially combinatorial, and the optimizer invocation is expensive.

As our *third* key result (Section 5.4), we develop *cost speculation*—based on a set of *assertions* that a reasonable query optimizer must conform to—to predict relative estimations that the optimizer will make, for pruning the search space. With such speculation, we can develop effective search-space pruning by an *anti-monotonicity* property of joining associations in navigating the search space, which intuitively means: If merging an attribute A with some attributes X (*i.e.*, table AX) increases the cost, then merging A with X' (table AX'), a superset of X, will increase the cost too.

Putting together, we now summarize our problem as follows, and present an overview our framework.

Problem. Given an *input* I that describes a target schema, over a *search space* $\mathcal{X}$ of candidate schemas S, find an *output* schema S^* that is *optimal* with respect to some cost estimate $\mathrm{Cost}_{\mathcal{D},\mathcal{O}}(W, S)$ over a database with data statistics $\mathcal{D}$ and query optimizer $\mathcal{O}$ and *workload* W of queries, *i.e.*,

$$S^* = \arg\min_{S \in \mathcal{X}} \mathrm{Cost}_{\mathcal{D},\mathcal{O}}(W, S). \tag{5.1}$$

Framework (Figure 5.5). The *navigator* coordinates the process. The *schema designer* takes associations and constraints (in the form of functional dependencies) as input. After eliminating the redundant associations, the minimal schema along with the input workload is passed to the *query optimizer* $\mathcal{O}$, which obtains the costs for each query from the workload. The costs are used by the *schema pruner* to validate an anti-monotonic property, which if not violated is used to prune the sub-optimal schemas. The leftover schemas are returned to the navigator, which continues generating candidates by merging tables. Eventually, when all the schemas are either evaluated or pruned, the navigator returns the optimal one.

5.2 CONCEPTUAL SCHEMA DESCRIPTION

To automate the design of schemas, what do we need to describe schemas that will satisfy our database applications? For our objective to design a schema, as the *physical* table organization of a database, we need a description that is *conceptual* in capturing the essential requirements that all physical schemas must meet. In this section, we develop such a specification ($\mathcal{A}$, $\mathcal{F}$) with two components: *(i)* target associations $\mathcal{A}$ and *(ii)* functional dependencies $\mathcal{F}$. As the "basic description" of a schema, it describes what a database can store, *i.e.*, information capacity [84].

5.2.1 Our Proposal

The purpose of relations and their organization as a schema is to store "associations" of attributes. In an application, a set of attributes $X = \{A_1, \ldots, A_n\}$ may form a (many-to-many) mapping that should be stored, or an *association*, which we denote $[A_1, \ldots, A_n]$. For example, for our Movie example, $X = \{\text{MId, Title, Genre, DId}\}$ forms an association between the attributes, which we denote $[X]$. A *tuple* x of an association $[X]$ is a joint assignment of its attributes, *i.e.*, $x \in \text{DOM}(A_1) \times \ldots \times \text{DOM}(A_n)$, *e.g.*, $x_1 =$ ("001", "Frozen", "animation", "d1") and $x_2 =$ ("120", "The Avengers", "action", "d2"). An *instance* $\boldsymbol{X}$ of an association $[X]$ is a set of X-*tuples* to be stored, *e.g.*, $\boldsymbol{X} = \{x_1, x_2, ...\}$.

Such associations are essentially what a database must "remember" by storing their instance values, and a correct schema must allow such storage of any instances in the database. Depending on its semantics and purposes, an application determines—through conceptual modeling— it's intended or "target associations", *i.e.*, what attributes are semantically related and thus their associations should be captured by the database. Different application semantics may determine different target associations among attributes A, B, and C, as Figure 5.6 summarizes.

Case	Asssociations $\mathcal{A}$			Functional Dependencies $\mathcal{F}$				Candidate Schemas S				
	$[AB]$	$[BC]$	$[AC]$	$A \to B$	$A \to C$	$B \to C$	$C \to B$	S_1: ABC	S_2: AB BC, AC	S_3: AB BC	S_4: AB AC	S_5: BC AC
1	***A*: MId, *B*: DId, *C*: DName**											
1a	*Movie has one director, and director has one name.*											
	✓	✓	✓	✓	✓	✓		■	●	■	■	
1b	*Movie has multiple directors, and director has one name.*											
	✓	✓	✓			✓		■	●	■		
1c	*Movie has one director, director has one distinct name.*											
	✓	✓	✓	✓	✓	✓	✓	■	●	■	■	■
2	***A*: MId, *B*: SId, *C*: AId**											
2a	*Movie belongs to multiple studios and has multiple actors.*											
	✓		✓								■	
2b	*Movie belongs to multiple studios and has multiple actors.*											
	✓	✓	✓						■			
2c	*Movie belongs to one studio and has multiple actors. Actor belongs to one studio.*											
	✓	✓	✓	✓			✓	■	●		■	■

● Correct schemas. ■ : Correct and non assoc-redundant schemas.

Figure 5.6: Candidate schemas for 3-attribute cases.

Example 5.1 (Target Associations). *Consider different cases in Figure 5.6.* **(Case 1)** *As a subset of the Movie application, consider A = MId, B = DId, and C = DName. This application* may *require to store associations* $[AB]$ *(a movie and its director),* $[AC]$ *(a movie and its director's name), and* $[BC]$ *(a director and her name), since these attributes are all semantically related.* **(Case 2a)** *As another example, for A = MId, B = SId, C = AId, we may need to store* $[AB]$ *(a movie its studios) and* $[AC]$ *(a movie and its actors) associations.* **(Case 2b, 2c)** *If, in addition, actors are affiliated with studios, we also need association* $[BC]$.

Our schema design must focus on schemas that satisfy the necessary storage capability for storing intended associations, among which to find an optimal one. To describe such capabilities and identify alternative schemas, we propose the input to be a *conceptual schema description* consisting of two components ($\mathcal{A}$, $\mathcal{F}$).

Target Associations $\mathcal{A}$, the set of associations that a schema must be able to store their instances, or "preserve". Applications must determine target associations and thus are given as an input to our framework. Example 5.1, our running example, shows the target associations are $\mathcal{A}_{1*} = \{[AB], [AC], [BC]\}$ for Case 1, $\mathcal{A}_{2a} = \{[AB], [AC]\}$ for Case 2a, and $\mathcal{A}_{2bc} = \{[AB], [AC], [BC]\}$ for 2b and 2c.

Functional Dependencies $\mathcal{F}$, the set of FDs $X \to Y$ where X functionally determines Y. By definition, an FD also indicates an association, *i.e.*, $X \to Y$ implies $[XY]$. FDs allow us to identify alternative or equivalent schemas, as we see later.

We note that our conceptual description ($\mathcal{A}$, $\mathcal{F}$) is necessary for capturing application semantics—and thus is readily available as virtually part of any conceptual modeling. For using a database, the process of *conceptual modeling* (such as ER or UML) determine and generate, among other things, what and how attributes are associated. In particular, an ER model (*e.g.*, Figure 5.1) by modeling entities and relationships, essentially describes attribute associations. In addition, as part of any conceptual modeling, FDs also imply associations. Since an FD $X \to Y$ specifies a *functional* (*i.e.*, many-to-one) mapping, $[XY]$ is thus an association by definition.

Example 5.2 (Conceptual Modeling)**.** *Consider Figure 5.1, what associations does it induce? An entity is an association among its attributes, and thus* $[\mathit{MId}, \mathit{Title}, \mathit{Genre}]$ *due to entity* Movie*. A relationship specifies an association among the (key) attributes of the involved entities; e.g.,* $[\mathit{MId}, \mathit{SId}]$ *due to the* Produces *relationship. Further, an FD associates attributes at the two sides. E.g.,* $\mathit{DId} \to \mathit{Genre}$ *indicates* $[\mathit{DId}, \mathit{Genre}]$*. Thus, the conceptual model in Figure 5.1 will induce target associations* $\mathcal{A} = \{A_1, \ldots, A_{11}\}$.

On the other hand, our conceptual description is quite *minimal* as input to capture application semantics and thus can be obtained even without a formal conceptual modeling process. In particular, the notion of association as simply a set of attributes is simpler than, say, entity and relationship. The knowledge of necessary associations is often implicit and embedded in any natural usage of data: We can observe this knowledge from users' *data* arrangement, such as in the common spreadsheet software [71, 72], where users naturally put associated attributes in the adjacent columns of tabular areas in a worksheet. This knowledge can also be observed from users' queries in a workload, *i.e.*, what attributes are used (joined) together in queries. Thus, in many natural scenarios, users need not explicitly model or specify target associations, which can be "observed" or "learned" simply from natural usage of data or queries.

Association Preservation. Given a conceptual description $\mathcal{A}$, our goal is to find schemas that preserve every association in $\mathcal{A}$ *w.r.t.* constraints in $\mathcal{F}$. We define a schema *preserves* an association $[X]$ *w.r.t.* $\mathcal{F}$, if it can store *any* instance $\boldsymbol{X}_i$ of $[X_i]$ that satisfies $\mathcal{F}$. There are two ways a schema can preserve an association.

On the one hand, an association $[X]$ can be *explicitly* preserved in a schema S, with all its attributes stored together in a table T. We can thus retrieve $[X]$ by accessing T. For example, in Case 1a, $[AB]$ is explicitly preserved in S_1 (or S_2), being part of table ABC (or table AB). Thus, those schemas that explicitly store every target association as part of a table is obviously correct. For example, schema $S_1 = \{ABC\}$ would be correct for all cases since the single table ABC can store all the combinations of attribute subsets and thus preserves all such associations.

On the other hand, $[X]$ can also be *implicitly* preserved in S, without being entirely contained in one table. In the presence of FDs, an association may be stored across multiple tables; *e.g.*, $[AC]$ is implicitly preserved in $S_3 = \{AB, BC\}$ although not entirely contained in a table, since it can be "recovered" from tables AB and BC, *i.e.*, $AC = \pi_{AC} AB \bowtie BC$. In other words, the table AB and AC can be losslessly joined due to FD $B \rightarrow C$. Similarly, if also $A \rightarrow B$, then AC can be recovered from AB and AC, thus $S_4 = \{AB, AC\}$ is correct for Cases 1a and 1c.

We note that, to determine association preservation, we need (and only need) FDs specified, which also justifies why they should be part of our conceptual description input.

Theorem 5.1. *A schema may implicitly preserve an association if and only if there are functional dependencies.*

(Sufficient) On the one hand, if FDs are specified, we can determine if a target association $[X]$ can be preserved implicitly by the *lossless join* property of some tables T_i that together preserves $[X]$, *i.e.*, $\exists\ U \subseteq \{1, \ldots, l\}$, such that $\cup_{i \in U} T_i \supseteq X$, and $\bowtie_{i \in U} T_i$ is *lossless* [85]. Intuitively, since their join is lossless, storing any instance $\boldsymbol{X}$ in these separate T_i tables is "as good as", or *equivalent* to, a table with attributes $\cup_{i \in U} T_i$, which explicitly preserves $[X]$.

(Necessary) On the other hand, if FDs are not specified, *i.e.*, there are no FDs, we cannot assert any other equivalent schemas for storing $[x]$ explicitly. Without FDs, according to the *Equivalence-implies-Equality* theorem [84], under any natural notion of "information capacity" equivalence, two relational schemata *with no dependencies* are equivalent if and only if they are identical (up to reordering of the attributes and relations)—thus, without FDs, there are no different schemas that can preserve $\mathcal{A}$.

Schema Correctness. Our conceptual description ($\mathcal{A}$, $\mathcal{F}$), as input, allows us to determine if a schema is "correct"—for the purpose of storing information required by applications. For this purpose, we define a schema *correct w.r.t.* target associations $\mathcal{A} = \{[X_1],\ldots,[X_m]\}$ if it can *preserve* every association $[X_i]$ *w.r.t.* $\mathcal{F}$. Figure 5.6 summarizes the correct schemas (marked with squares and dots) for each case. For example, to preserve $\mathcal{A}_{1*} = \{[AB],[AC],[BC]\}$, Case 1a, given FDs $\mathcal{F}_{1a} = \{A \rightarrow B, A \rightarrow C, B \rightarrow C\}$, has correct schemas $\{S_1, S_2, S_3, S_4\}$. However, Case 1b, with only $\mathcal{F}_{1b} = \{B \rightarrow C\}$, has fewer correct schemas $\{S_1, S_2, S_3\}$.

Equivalent Schemas. With respect to a conceptual description ($\mathcal{A}$, $\mathcal{F}$), the objective of the schema design problem is to find, among those candidate schemas that are "equivalent," the best in terms of other physical considerations such as performance in our setting. Thus, central in the schema design problem, we must determine the *equivalence* of schemas, to form the search space of equivalent schemas.

In our setting, we consider an application-centric sense of schema equivalence: With respect to ($\mathcal{A}$, $\mathcal{F}$), two schemas S and S' are equivalent if they are both correct. Since our conceptual description captures the storage capacity required by an application, this equivalence is *application-centric*—as intended by an application. That is, S and S' are equivalent (*e.g.*, Case 2a, $S_1 = \{ABC\}$ and S_4=$\{AB, AC\}$), if both can store the target associations ($[AB]$ and $[AC]$), even though one *can* store more (unintended) associations than the other (S_1 but not S_4 can also store $[BC]$). The equivalence also depends on the constraints of the application; *e.g.*, in this example, S_1 and S_4 are equivalent due to $A \rightarrow B$, which renders S_4 (like S_1) capable of storing BC.

5.2.2 Contrast with Literature

As schema design has been extensively studied—although mostly with an exclusive focus on normal forms—our framework for schema normalization contains several concepts that find parallels in the literature. In this section, we explain the need and novelty for our proposed notions from a comparative perspective.

Associations. Our notion of association aims at capturing the "basic units" of information that an application needs the database to store. Such a basic unit—being basic—should be primitive and common in every data model (*e.g.*, ER or UML), so that it does not depend on the actual data models and is available in any. Association is similar to a *table* in the relational model; we can consider associations as a partial table (or multiple partial tables). As discussed earlier, an association is more primitive than classic conceptual modeling concepts such as entity and relationship in the ER-model—and can be induced from them.

For modeling such basic information units, there are some similar attempts in the literature, but a complete solution seems to remain open—and thus our proposal of associations. To model information units, we need both a representation framework (which we proposed the association notion) and an inference mechanism to determine minimal representations (which Section 5.3 will discuss).

For *representation*, some early literature in schema design proposes to use *functional dependencies* (FDs) as such a basic description [25, 86], where each FD $X \rightarrow Y$ (*e.g.*, MId $\rightarrow$ Title) describes the (functional) relationships between attribute sets X and Y. However, not all associations among attributes are functional. Later, Bernstein [21] augmented the description with *θ-notation*: a nonfunctional relationship f among attributes $A_1, A_2, \ldots, A_n$ is represented by FD F: $A_1, \ldots, A_n \rightarrow \theta_F$, and θ_F is an imaginary attribute unique to F (it does not appear in any other FD).

For *inference*, the problem of finding a minimal representation of information units is mostly missing in the literature. While the θ-notation attempts to represent nonfunctional associations, there is no corresponding inference. The closest may be the inference mechanism of FDs (Armstrong Axioms [87] for finding minimal covers)—but it only deals with functional associations. Without proper representation and inference framework, classic algorithms that construct schemas (say 3NF) may result in undesirable tables, as we show below. As Example 5.3 shows, do we need all three relations to store the desired associations?

Example 5.3 (θ-Notation)**.** *Consider Case* 1*b, for attributes A:MId, B:DId, and C:DName and FD $B \rightarrow C$. In addition to association $[BC]$ as the FD indicates, suppose there are also $[AB]$ and $[AC]$, which are nonfunctional (movies and directors are associated many-to-many). Bernstein's 3NF schema-synthesis framework [21] will represent these nonfunctional associations by the theta-notation as $AB \rightarrow \theta_1$ and $AC \rightarrow \theta_2$. The three FDs will result in a 3NF schema with three tables $S_2 = \{AB, AC, BC\}$—but will miss other correct (and non-redundant) 3NF schemas like S_3. Similarly, for Cases 1a and 1c, it will also synthesize S_2 and miss all others (S_1, S_3, S_4, and S_5).*

In retrospect, the missing of a basic description of relational schemas seems inevitable, due to the exclusive attention on normal forms. Most scheme design study focuses on normalization to produce various normal forms, *e.g.*, 3NF [21], BCNF [88], 4NF [89], or PJ/NF [90]. As FDs fully determine normal forms, they are commonly assumed as a sufficient description of a database for schema design [21, 25, 86]. There are two different types of approaches to formal database design [91].

On the one hand, the *decomposition* approaches [92, 89] start with an *initial schema* (or a universal relation [83]) to stepwise decompose it into simpler and smaller tables—which thus avoids the need for explicitly describing basic requirements. On the other hand, the opposite *synthetic* approaches directly group attributes into tables—however, again, as dictated exclusively by functional relationships (*e.g.*, [25, 86]), including the addition θ-notation FDs to capture nonfunctional associations.

Correct Schemas. Note that our notion of schema correctness is orthogonal to the classic notions of *normal forms* such as 3NF [21] or BCNF [88]. This is expected since our correctness of a schema concerns only that the target information *can* be stored and not *how* it is stored—so that we can further choose a physical schema based on the performance. For Example, in Case 1, with $B \rightarrow C$, S_1 is not in BCNF, but we consider it correct since it can store all target associations. It may have a performance advantage over a BCNF (such as

S_3) for some workloads.

Schema Equivalence. Our notion of application-centric equivalence is novel, in contrast to the traditional senses of equivalence, which is essentially the opposite. Schema equivalence has been extensively explored in the literature, with multiple equivalence notions proposed, such as *representational equivalence* between a relational schema and its decomposition [93], *data equivalence* [83], θ-equivalence [94] or *query equivalence* [95], *calculus equivalence* and more [84]. These various senses of equivalence compare schemas by the "amount" of applications (data or query capabilities) each can accommodate. For example, two schemas are "data equivalent" if they can store the same set of instances [83], or "query equivalent" if there is a query that maps an instance of the first schema to that of the second and vice versa [95]. In other words, these equivalence senses measure the "absolute" capacity of a schema, while we are only interested in the capacity with respect to an application.

5.3 SEARCH SPACE

In this section, we describe the search space of quantitatively designing a schema. That is, starting from a conceptual description ($\mathcal{A}$, $\mathcal{F}$), we aim at finding a schema, among possible alternatives (equivalent schemas), that minimizes the workload cost. We define our search space as *a set of schemas that precisely preserve* $\mathcal{A}$ *with respect to* $\mathcal{F}$. That is, our search space consists of schemas that preserve ($\mathcal{A}$, $\mathcal{F}$) and do not preserve any "redundant" or "unintended" association. Thus, to eliminate "redundant" or "unintended" associations, we need inference rules to identify them. Starting from ($\mathcal{A}$, $\mathcal{F}$), after eliminating the redundant associations using inference rules, we are left with a *minimal basis*, *i.e.*, a minimal set of associations that if preserved also preserves $\mathcal{A}$. In other words, minimal basis helps us to understand what associations need to be explicitly preserved by a schema. Additionally, the minimal basis serves as a starting point, from which we navigate our search space.

We now discuss the two assumptions that we use to define the search space namely *(i)* no association redundancy and *(ii)* no unintended associations. Our redundancy notion is quite in the spirit of *strong redundancy* as Codd defines [77]: "*A set of relations is strongly redundant if it contains at least one relation that possesses a projection which is derivable* *from other projections of relations in the set.*"

No Association Redundancy. While a schema should preserve associations, it should not store more than what is necessary. Various non-redundancy criteria can be meaningful depending on specific applications—this has been studied by the literature of *schema normalization*. However, as we aim to minimize query cost, we adopt a very basic notion where no associations are repeated.

Formally, a schema S is *non associations-redundant* if there does not exist any attributes or tables that we can eliminate from S and the resulted schema is still association preserving. Consider S_2, for Case 1a in Figure 5.6—S_2 explicitly preserves $[BC]$. Preserving $[BC]$, which can be inferred from $[AB]$, $[AC]$, and $A \rightarrow B$, explicitly results in redundancy as $[BC]$ can be dropped from S_2 without loosing the any associations. On the other hand, an interesting case is S_1, which also preserves $[BC]$ explicitly—but in this case we cannot drop any attribute (or table) without losing any essential association. Thus, S_1 is non-associations-redundant.

No Unintended Associations. For correctness, a schema should also not introduce any unintended associations. We consider an association as *unintended* if it associates attributes in a way that is inconsistent with the input associations. Consider S_2, for Case 2a in Figure 5.6—S_2 explicitly preserves $[BC]$. We consider $[BC]$ as unintended as it can can store inconsistent mapping between values of SId and AId that do not correspond to a tuple given by $\{\text{MId}, \text{SId}\}$ and $\{\text{MId}, \text{AId}\}$ associations.

5.3.1 Inference Rules and Minimal Basis

In this section, we develop inference rules for associations and use them to develop the notion of "minimal basis".

Uniqueness. To enable inference, we assume that any two attributes can be associated with each other in at most one way. For example, the associations between MId and AId given by $[\text{MId}, \text{AId}]$ and $[\text{MId}, \text{AId}, \text{Rid}]$ is the same. Our assumption is in a similar sprite to earlier work [25, 21] and is crucial to develop inference rules. Although this assumption is quite strong, it can be easily circumvented by renaming or creating additional attributes. We formalize this as follows:

Assumption 5.1 (Uniqueness)**.** *Between any two sets of attributes, there is at most one association.*

Inferring Associations. We now develop rules for inferring associations. We use these rules in conjunction with the traditional interference rules for FDs to identify and eliminate the redundant associations.

For the rules that infer an association from a FD, we ensure that the left-hand side of the FD is irreducible. That is, the FD needs to be explicitly preserved by a schema and is not implicitly preserved as a result of preserving other FDs. Formally, for two attribute sets T and U, if $T \rightarrow U$, then $T' \nrightarrow U'$, where $T' \subset T, U' \subseteq U$.

To motivate our first rule, consider a FD $F_1 = \text{MId} \rightarrow \text{Title}, \text{Genre}$. For a schema to preserve F_1 it needs to preserve the association between MId, Title, and Genre. We formalize this as:

1. $T \rightarrow U \implies [TU]$

For the second rule, consider an association, $A_1 = [\text{MId}, \text{Sid}]$ and a FD, $F_2 = \text{MId} \rightarrow \text{Did}$. For a schema to preserve A_1 with respect to F_2 it requires: *(i)* a table $T_1 \supseteq \{\text{MId}, \text{Sid}\}$, and *(ii)* a table $T_2 \supseteq \{\text{MId}, \text{Did}\}$, with a constraint $\text{MId} \rightarrow \text{Did}$. Since $\text{MId} \rightarrow \text{Did}$, $\pi_{\text{MId,Sid}}(T_1)$ and $\pi_{\text{MId,Did}}(T_2)$ are lossless decompositions of $T = \{\text{MId}, \text{Did}, \text{Sid}\}$, which preserves $[\text{Sid}, \text{Did}]$, *i.e.*, the two tables implicitly preserve $[\text{Sid}, \text{Did}]$. Since $[\text{Sid}, \text{Did}]$ can be inferred from others, it is non-essential and can be eliminated. We formalize this as:

2. $[TV], T \rightarrow U \implies [TUV]$

By definition, if a schema preserves $[T]$, where $T = \{A_1, A_2, \ldots\}$, then it can store any combination of the attribute values permitted by the constraints. Thus, it can also store any combination of attribute values of U permitted by the constraints, where $U \subseteq T$, *e.g.*, $[\text{MId}, \text{AId}, \text{Rid}] \implies [\text{MId}, \text{AId}]$. Therefore, we have,

3. $[T], U \subseteq T \implies [U]$.

We have the following two theorems for the correctness and completeness of the above rules. To prove completeness, we show that, starting from a set of associations $\mathcal{A}$ and FDs $\mathcal{F}$, if an association between two attributes A and B, *i.e.*, $[AB]$, cannot be inferred based on the rules then there exists a schema S that preserves $\mathcal{A}$ and $\mathcal{F}$ but cannot store at least one tuple of $DOM(A) \times DOM(B)$ that satisfies $\mathcal{F}$ in a lossless manner.

Theorem 5.2 (Correctness). *For a set of associations $\mathcal{A}$ and FDs $\mathcal{F}$, if we can infer an association A using the above rules then any schema that preserves $\mathcal{A}$ and $\mathcal{F}$ must also preserve A.*

Theorem 5.3 (Completeness). *For a set of associations $\mathcal{A}$ and FDs $\mathcal{F}$, if we cannot infer an association $[T]$ using the above rules, then there exists at least one schema that preserves $\mathcal{A}$ and $\mathcal{F}$ but does not to preserve A or $\mathcal{F}$.*

Minimal Basis. By utilizing the interference rules, we establish equivalence between sets of associations. Informally, we consider two association sets as *equivalent (basis)*, with respect to a set of FDs (FDs are imperative for inferring associations—see Section 5.2.1), if they can be inferred from each other. Consider the associations, $\mathcal{A}_1 = \{[\text{MId}, \text{Sid}], [\text{MId}, \text{Did}]\}$ and $\mathcal{A}_2 = \{[\text{MId}, \text{Sid}, \text{Did}]\}$. Here, $\mathcal{A}_1$ and $\mathcal{A}_2$ are equivalent if we have MId $\rightarrow$ Did, and thus are basis of each other. We formalize the notion of basis as:

Definition 5.1 (Basis). *A set of associations $\mathcal{A}'$ is a basis of another set of associations $\mathcal{A}$, with respect to a set of FDs $\mathcal{F}$, if any schema S that preserves $\mathcal{A}$ with respect to $\mathcal{F}$ also preserves $\mathcal{A}'$ with respect to $\mathcal{F}$ and vice-versa.*

We now use the notion of basis to define "minimal basis" for a set of associations. Informally, a minimal basis is a set of associations such that none of the associations can be inferred from the remaining. Therefore, the associations described by a minimal basis are *essential building blocks* that describe a schema. We formalize this as below.

Definition 5.2 (Minimal Basis). *A minimal basis, denoted as $\hat{\mathcal{A}}$, for a set of associations, with respect to a set of FDs $\mathcal{F}$, is a basis where: If any association or an attribute within any association is removed then the result is no longer a basis.*

Finding Minimal Basis: We use the aforementioned rules to eliminate the redundant associations to obtain a minimal basis. For example, in Figure 5.6, for Case 1a, the minimal basis is either $\{[AB], [BC]\}$ or $\{[AB], [AC]\}$. One interesting aspect to note here is that there can be more than one minimal basis for the same $(\mathcal{A}, \mathcal{F})$.

5.3.2 Search Space and its Navigation

In this section, we formalize our search space and propose a way to navigate it such that it can be efficiently pruned.

Search Space. With respect to a set of associations $\mathcal{A}$ and FDs $\mathcal{F}$, we define our *search space* as a set of schemas that preserve essential associations $\hat{\mathcal{A}}$, and adhere to our assumptions, *i.e.*, *(i)* no attribute redundancy and *(ii)* no unintended associations. We limit our scope to schemas obtained by regrouping the attributes—this naturally encompasses the search spaces of the two major performance related schema transformations proposed in the literature: *(i) vertical partitioning*, where a table is split into multiple tables and *(ii) merging*, where multiple tables are merged.

Minimal Schema. We introduce "minimal schema", a correct schema that cannot be further decomposed, and use it as a starting point for navigating our search space. As the associations in minimal basis are non-redundant and non-decomposable, if we synthesize a schema from the minimal basis of $(\mathcal{A}, \mathcal{F})$ such that each table corresponds to one association in the minimal basis $\hat{\mathcal{A}}$ then we term it as a the minimal schema. We formalize the notion of minimal schema below:

Definition 5.3 (Minimal Schema (S^m)). *Given a set of associations $\mathcal{A}$ with respect to FDs $\mathcal{F}$, a schema S is minimal if it has a table corresponding to every association in the minimal basis of $\mathcal{A}$ with respect to FDs $\mathcal{F}$.*

We obtain the minimal schema for our example using the minimal basis of the associations. For example, for Case 1a in Figure 5.6, one minimal schema is:
$\{\mathsf{MovieDir}(\mathrm{MId}, \mathrm{DId}), \mathsf{DirName}(\mathrm{DId}, \mathrm{DName})\}$

Navigation. We use minimal schema as a starting point to obtain the candidates by using the primitive of "merging"—this enables us to efficiently prune the search space Section 5.4 will discuss. For correctness, we merge two tables T and U only if the merged table, *i.e.*, $T \bowtie U$, preserves the same set of associations as preserved by T and U. Thus, as discussed earlier not every merge is valid. Since merging two tables to obtain $T \bowtie U$ does not add any new constraints, the merged table always preserves the associations preserved by T and U.

However, since a natural join groups attributes together the resultant table can preserves additional associations as seen earlier. Hence, we only choose to combine two tables if their merge is valid as defined below.

Definition 5.4 (Valid Merge)**.** *We consider a merge of two tables T and U as valid if their natural join, i.e., $T \bowtie U$, does not preserve any additional associations as compared to T and U.*

We next obtain the condition to check if a merge is valid. For a merge between T and U to be valid, the two tables should be able to store all the tuples of the $T \bowtie U$ and vice-versa. That is, T and U should be lossless representation of $T \bowtie U$, *e.g.*, MovieX. Therefore, we use the idea of lossless join decomposition to determine that a merge is valid by checking if the common attributes, *i.e.*, $T \cap U$ functionally determine at least one of the two tables. Note that since merge of Produces and Cast does not satisfy the lossless join decomposition condition, it is not valid. We formalize this as the following theorem.

Theorem 5.4 (Merge Condition)**.** *A merge of two tables T and U is valid if and only if $T \cap U \rightarrow U$ or $T \cap U \rightarrow T$.*

We formalize the primitive of merging as function $\mathcal{M}$, which obtains a new schema S' from S by merging two of its tables $T, U \in S$, where $T \cap U \rightarrow U$ or $T \cap U \rightarrow T$, *i.e.*,

$$\mathcal{M}(S, T, U) = \{T \bowtie U\} \cup (S - \{T, U\}) . \tag{5.2}$$

We now define our search space using a recursive function $f_{\mathcal{X}}$ that takes a schema as an argument and recursively merges tables to obtain candidate schemas.

$$f_{\mathcal{X}}(S) = \{S\} \cup \{S' | \forall T, U \in S, T \neq U, T \cap U \rightarrow U \text{ or } T \cap U \rightarrow T, S' = \mathcal{M}(S, T, U)\} \tag{5.3}$$

Using Equation 5.3, we obtain our search space by passing minimal schema S^m as an argument, *i.e.*, $\mathcal{X} = f_{\mathcal{X}}(S^m)$. Generating the search space using the above equation ensures:

Scenario	*Schema*	*Execution Plan*
1a (S_{1a})	• StudioNm (<u>Sid</u>, Nm) • StudioAddr (<u>Sid</u>, Addr) • ProGenre (<u>Sid</u>, <u>MId</u>, Genre)	5. Nested Loop (cost=0.00..26823.06 rows=11 width=76) 4. Seq Scan on StudioAddr (cost=0.00..2.00 rows=100 width=37) 3. Nested Loop (cost=0.00..26804.53 rows=11 width=51) 2. Seq Scan on StudioNm (cost=0.00..2.00 rows=100 width=10) 1. Seq Scan on ProGenre (cost=0.00..26786.00 rows=11 width=41) Filter: (genre = 'SiFi'::text)
1b (S_{1b})	• StudioAddr (<u>Sid</u>, Addr) • ProducesNm (<u>Sid</u>, <u>MId</u>, Genre, Nm)	3. Nested Loop (cost=0.00..27903.53 rows=11 width=76) 2. Seq Scan on StudioAddr (cost=0.00..2.00 rows=100 width=37) 1. Seq Scan on ProducesNm (cost=0.00..27885.00 rows=11 width=47) Filter: (genre = 'SiFi'::text)
2a (S_{2a})	• StudioNm (<u>Sid</u>, Nm) • ProducesAddr (<u>Sid</u>, <u>MId</u>, Genre, Addr)	3. Nested Loop (cost=0.00..30700.53 rows=11 width=76) 2. Seq Scan on StudioNm (cost=0.00..2.00 rows=100 width=10) 1. Seq Scan on ProducesAddr (cost=0.00..30682.00 rows=11 width=74) Filter: (genre = 'SiFi'::text)
2b (S_{2b})	• ProducesAll (<u>Sid</u>, <u>MId</u>, Genre, Addr, Nm)	1. Seq Scan on ProducesAll (cost=0.00..31731.00 rows=11 width=76) Filter: (genre = 'SiFi'::text)

Figure 5.7: The impact of merging tables on the execution plan and cost of a query that accesses the merged tables.

(i) correctness, *i.e.*, all the schemas precisely preserve the input associations and *(ii)* completeness, *i.e.*, all the schemas that are correct are generated. We formalize this as the following two theorems.

Theorem 5.5 (Merge Correctness)**.** *All the schemas derived from minimal basis using the merge primitive are correct and withhold our assumptions (non-redundant and non-unintended associations).*

Theorem 5.6 (Merge Completeness)**.** *If a schema satisfies our assumptions (non-redundant and non-unintended associations), then it should be derivable from the minimal basis by merging else it is not correct.*

5.4 PRUNING DESIGN SPACE

In this section, we discuss the pruning of the schema design search space to obtain the optimal one. Due to the exponential nature of the search space, using traditional cost models [80], which are embedded in query optimizers, to evaluate every candidate from the search space is prohibitive—our insight in addressing this challenge is to understand a query optimizer's behavior when merging two tables, thereby pruning the search space by evaluating fewer candidates. To this end, we develop an anti-monotonic property, which makes a few basic assumptions for the query optimizers, to efficiently and correctly prune the search space.

5.4.1 Understanding Cost Behavior

To understand query optimizer's behavior when merging two tables, we discuss how the merge impacts a query's cost. Similar to earlier work [80], but with a purpose to understand the behavior of query optimizers, we classify an execution plan's operators as *table access* and *in-memory.* We term the cost incurred by a database to retrieve a table's tuples from storage as *table access cost*—this generally corresponds to the leaf operators on an execution plan. We term the cost incurred by non-leaf nodes on execution plans, which generally correspond to in-memory operations, *e.g.*, aggregation, as *in-memory cost.* We further break down the table access cost as "start-up" and "retrieval", where the start-up cost accounts for the effort spent by the database to obtain the first tuple, and the retrieval cost accounts for the effort spent by the database to obtain the remaining tuples.

As a query optimizer strives to select an optimal execution plan, we capture the behavior of the optimizer, and thus the cost, using the following two prepositions, for a query when two tables joined by the query are merged. We can show the correctness of the prepositions based on the Optimal Plan assumption. We develop our prepositions to specifically focus on the cost increase as a result of merging as this will enable us to develop pruning rules for the search space.

Assumption 5.2 (Optimal Plan). *A database's query optimizer chooses an optimal execution plan—thus, for accessing an individual table, it chooses the best table access method among the possible alternatives.*

Proposition 1 (Start-up Cost). *For a query Q that accesses a table T, adding an attribute to T does not increase the startup cost for T, but may increase or decrease the access cost for T.*

Proposition 2 (Access Cost). *For a query Q that accesses a table T, adding an attribute to T does not increase to the access cost of tables other than T.*

Based on Proposition 1 and 2 we conclude that after merging two tables the reason for increasing the cost of a query accessing the merged tables is one or more of the following: *(i)* increased row size, *(ii)* change in table access operator, and *(iii)* duplicated tuples. Not being aware of the actual reason for the cost increase, makes modeling the optimizer's behavior difficult.

However, if we merge tables in a *controlled* manner, we limit the impacted factors, thereby enabling us to predict the behavior of a query optimizer. Instead of arbitrarily choosing tables to merge, we choose one of the tables such that its attribute values are not duplicated. The FD between key attributes of the merged tables determine attribute value duplication—we capture this by defining a new notion of "table order", such that if T is ordered before U, denoted as $T \leq_S U$, then T's tuples are not duplicated when T and U are merged. We formalize this as:

Definition 5.5 (Table Order). *A schema S's table order, with respect to functional dependencies F, is a partial order among the tables within the schema such that for any two tables $T, U \in S$, with a common set of attributes, i.e., $V = T \cap U$, there exists an order (i) $T \leq_S U$ if $V \rightarrow U$, (ii) $U \leq_S T$ if $V \rightarrow T$, and (iii) $T =_S U$ if $V \rightarrow T, U$.*

5.4.2 Pruning using Anti-monotonicity

To demonstrate our intuition for anti-monotonicity, in Scenario 1, we first select a table, *i.e.*, StudioNm, which when merged with ProGenre results in increasing Q_2's cost as compared to its cost on S_{1a}. We then use Scenario 2 to show that the cost behavior, *i.e.*, increase in cost, persists when StudioNm is merged with a superset of ProGenre, *i.e.*, ProducesAddr, which we obtain by merging StudioAddr with ProGenre. To develop the anti-monotonic property, we reason out that the cost behavior that we observe in Scenario 2 based on Scenario 1's cost behavior.

For the two scenarios, we use our cost model to understand the cost behavior by finding out the responsible factors. We denote the three tables ProGenre, StudioNm, and StudioAddr by T, U, and V respectively. From the keys of the table we can observe that the tables are ordered as $T \leq_S U, V$.

Scenario 5.1. Here, we obtain S_{1b} from S_{1a} by merging StudioNm with ProGenre to obtain ProducesNm, we denote it as T_1. Referring to $E(S_{1b})$, we note that the query's cost increases despite eliminating a table access operator. Due to the table order of $T \leq_S U$, the tuples of ProGenre are not duplicated. The operator that accesses ProducesNm now bears an extra burden of accessing an additional attribute StudioName indicated by a higher width of 47 in Step 1 for $E(S_{1b})$ compared to that of 41 in Step 1 for $E(S_{1a})$. Since the overall cost increases, the cost increase due to the additional attribute is higher than the cost saving of eliminating the table access operator.

Scenario 5.2. Here, we consider the table ProducesNmAll, which we obtain by merging ProducesAddr and StudioNm. Due to the table order of $T_1 \leq_S V$, the tuples of ProducesAddr are not duplicated. By comparing the costs for S_{2a} and S_{2b}, we observe that the cost increase is similar to Scenario 5.1's cost increase. Contrasting the starting point of Scenario 5.2, *i.e.*, S_{2a}, with the starting point of Scenario 5.1, *i.e.*, S_{1a}, we note that the table T has an additional attribute, we denote it as T_1'. Since the additional attributes should not impact of merging StudioNm, similar to Scenario 5.1, we attribute the cost increase to the additional fields of ProducesAll, which we denote as T_2, as compared to that of ProducesAdd, which increase ProGenre's record size.

Motivated from the cost behavior, we develop an anti-monotonic property. Intuitively, if an attribute merged with a table *increases* a query's cost, then the attribute merged with a superset of the table with the same key will *increase* the query's cost. Consider query Q_2, along with three tables from S, *i.e.*, $T =$ ProGenre, $U =$ StudioNm, and $V =$ StudioAddr, such that $T \leq_S U, V$. We observe from Scenario 1 that by merging U with T increases Q_2's cost, *i.e.*, $S_1 = \{T_1\} \cup (S - \{T, U\})$, where $T_1 = T \bowtie U$ has a higher cost than S_{1a}. With an intuition that the cost increase observed in Scenario 2 is a consequence of the cost increase observed in Scenario 1, we introduce the anti-monotonic property, which we formalize as below.

Pruning Rule 5.1 (Anti-monotonicity)**.** *Suppose that we are given (i) a schema S, (ii) a query Q, (iii) tables $T, U, V \in S$, and (iv) $T \leq_S U, V$. Consider the schemas: (i) $S_1 = \{T_1\} \cup (S - \{T, U\})$, where $T_1 = T \bowtie U$. (ii) $S_1' = \{T_1'\} \cup (S - \{T, V\})$, where $T_1' = T \bowtie V$.*

Algorithm 5.1 Schema search space traversal.

Input: Schema S, workload W, and sub-optimal merges $\mathcal{M}^O$.
Output: List of candidate schemas.

```
1: function TRAVERSE($S, W, \mathcal{M}^O$)
2:    if $|S| = 1$ then                                          ▷ Terminating Condition.
3:       return $\{S\}$
4:    end if
5:    Initialize $R \leftarrow \emptyset$                                   ▷ Return Schema Set
6:    for $T, T' \in S, T \neq T'$ and ($T \leq_S T'$ or $T \leq_S T'$) do
7:       if $\nexists (U, U') \in \mathcal{M}^O \mid T =_S U, T' \subset U', T' =_S U'$ then
8:          $S' \leftarrow \mathcal{M}(S, T, T')$                                         ▷ (5.2)
9:          Prune ← true
10:         for each $Q \in W$ do
11:            if $T \cap Q \neq \emptyset$ and $T' \cap Q \neq \emptyset$ then
12:               if $\text{Cost}_{S'}(Q) < \text{Cost}_S(Q)$) then
13:                  Prune ← false
14:               end if
15:            end if
16:         end for
17:         if not Prune then
18:            $R \leftarrow R \cup$ TRAVERSE($S', W, \mathcal{M}^O$)
19:         else
20:            $\mathcal{M}^O \leftarrow \mathcal{M}^O \cup \{(T, T')\}$
21:         end if
22:      end if
23:   end for
24:   return $R$
25: end function
```

(iii) $S_2 = \{T_2\} \cup (S - \{T, U, V\})$, *where* $T_2 = T_1' \bowtie U$. *For query* Q, *if* $\text{Cost}(S_1) > \text{Cost}(S)$, *then* $\text{Cost}(S_2) > \text{Cost}(S_1')$.

We now develop an algorithm to use the anti-monotonic property to prune the schema search space efficiently. Although our goal is to optimize a schema for a workload, to simplify our discussion without losing generality, we consider a single query. While optimizing for a workload, we consider the cost of each query from the workload and prune a schema only if it is sub-optimal for *all* queries. Although this reduces the pruned schemas, for realistic workloads due to the similarity between the queries in terms of accessed attributes the impact on the design time is not adverse, which is also evident from our experiments.

The idea for cost-based pruning is to keep track of merges that increase cost. Recall that according to the anti-monotonic property, if an attribute merged with a table **increases** the cost of a query then the attribute merged with a superset of the table, which has the same primary key, will also **increase** the query's cost. We term a merge that increases the cost of a schema as *sub-optimal*, as the schema obtained by such a merge has a higher cost than its parent, and can be pruned. We capture all such suboptimal merges as a set $\mathcal{M}^O$ containing pairs of tables that result in a sub-optimal merge. Consider a schema S, containing tables T and U, which when merged results in increasing a query Q's cost. Then for a schema S', which contains table T', a superset of T, *i.e.*, $T' \supset T$, merging T' with U will result in increasing the cost of the query, and hence we skip such a schema. That is, before merging two tables, we ensure that a subset of the tables has not resulted in a sub-optimal merge, *i.e.*, $\nexists (U, U') \in \mathcal{M}^O \mid T =_S U, T' \subset U', T' =_S U'$. If "no" then we do not merge the tables thus eliminating their children; if "yes" then we merge the tables to obtain the child schema.

As the process described above navigates the space by incrementally merging tables, we naturally represent it as a recursive algorithm, which we summarize as Algorithm 5.1. The algorithm starts from minimal schema, traverses the space pruning the sub-optimal candidates, and returns a substantially small set of candidates, out of which we merely pick up the minimum cost schema as optimal.

5.5 EXPERIMENTAL EVALUATION

In this section, we experimentally validate our framework. We first describe our setting and then demonstrate the effectiveness of our solution and validate core components.

5.5.1 Experimental Setting

For evaluation we choose PostgreSQL 9.2 [96] for our database along with three datasets. To use PostgreSQL as a black box for schema evaluation, we tweak its source to return a workload's cost with respect to a schema solely based on table statistics. We choose the datasets to represent the space of data-driven applications faithfully and to stress our framework. Considering schema design as a combination of two orthogonal schema optimization methods, *i.e.*, partitioning and denormalization, we select two datasets such that they only

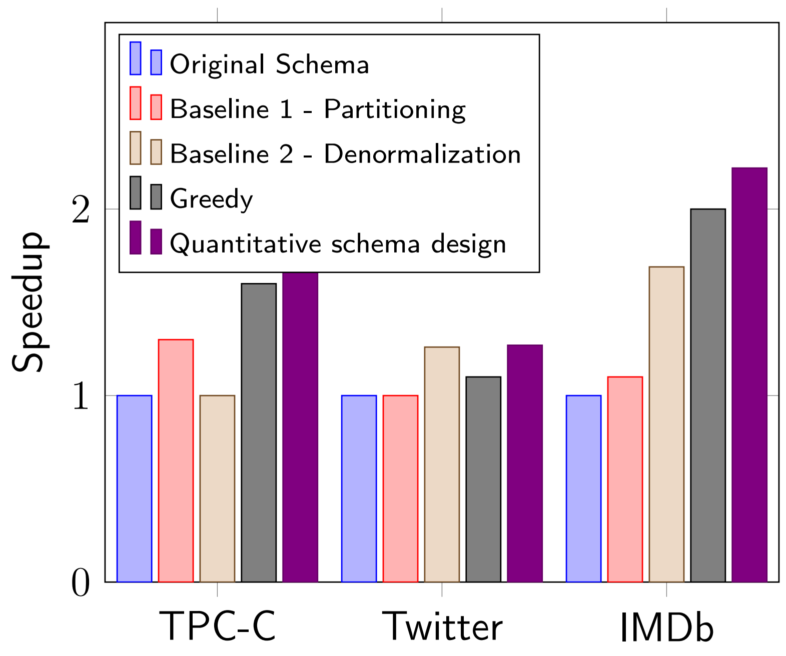


Figure 5.8: Speedup of quantitative schema design.

benefit from one of the two methods while the third is in the middle of the two extremities. We list the datasets in a descending order of their complexity in terms of attribute counts and joins performed by their corresponding queries.

TPC-C. TPC-C [97] benchmark is a popular standard for comparing performances of OLTP databases. It has a mixture of user and system generated content, which is similar to product websites such as Amazon [98]. The system generated content, *e.g.*, product information, is read-only and has the corresponding user-generated content, *e.g.*, reviews, which is frequently updated along with reads. TPC-C's well-defined workload resembles the activities of an ordering application. The workload consists of select queries for reporting, which perform complex joins across multiple tables, which have high multiplicity, accessing a few attributes from each table, along with simple transaction queries. For our purpose, we capture 2.5k queries from a benchmark run with a default setup.

Twitter. This dataset represents applications that primarily store and serve user-generated content. A majority of the tables are transactional, which are frequently inserted into and queried based on user actions. We have handwritten the schema based on traditional design techniques, *i.e.*, normalized to BCNF, to simulate the working of Twitter [99] and populated

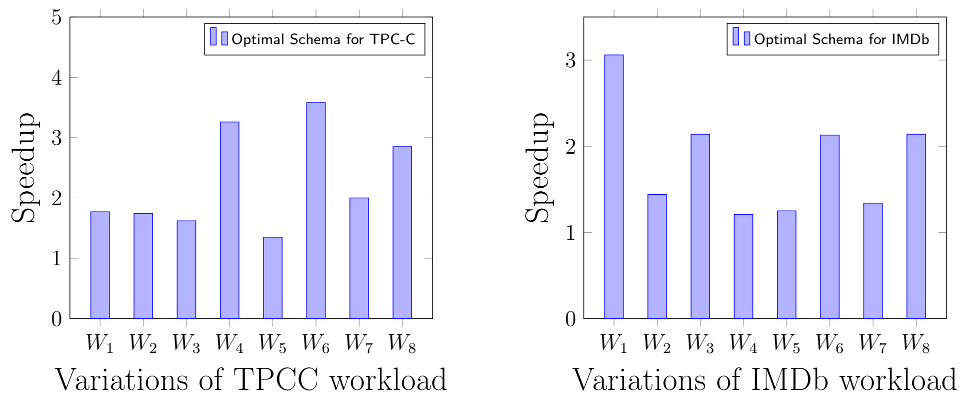


Figure 5.9: Performance gains variations across workloads.

with 1M randomly generated tweets from 10k users. For the workload we have 2k queries, which have simple joins and access almost all attributes from the tables, to simulate Twitter activities, *e.g.*, reading tweets.

IMDb. This dataset represents applications that primarily serve read-only content. The dataset is a dump from IMDb [73] for around 2M movies. The workload has handwritten queries to depict different activities on IMDb, *e.g.*, looking up a movie's cast. These queries are further duplicated, with different values, to obtain a total of 1k queries to simulate the effect of various activities by multiple users.

Evaluation Metric. As the query optimizer's cost estimate is a good indication of a query's response time, we use it for evaluation. We use speedup, which we calculate as a ratio of the old cost to the new one, as our evaluation metric.

Platform. We run our experiments on a desktop computer equipped with an AMD Phenom II 1045T processor and 16GB of RAM. The operating system is Ubuntu Linux 14, and the schema design framework is a single threaded application developed in Java.

Baselines. We use the most relevant work, *i.e.*, partitioning and merging, as our two baselines. For partitioning, we choose Autopart [100], which is the current state-of-the-art. Since the impact of merging tables has not been quantified previously in the literature, we create a parallel work for it based on the greedy approach proposed by Autopart. The idea is to greedily pick up two tables to merge such that they benefit the workload.

5.5.2 Schema Design Performance

We demonstrate the significance and practicality of quantitative schema design by evaluating it on aspects such as effectiveness, running time and workload.

Effectiveness. We demonstrate the effectiveness of our proposed method by contrasting its gain with the two baselines as shown in Figure 5.8. We observe a speedup of 1.72, 1.27, and 2.22 for TPC-C, Twitter, and IMDb respectively, which is an improvement over the baselines. For TPC-C, due to the complex joins and the high multiplicity, merging by itself is unable to benefit. Since the queries from the workload access relatively few attributes from a table, a partitioning technique helps the workload by reducing the overhead incurred from reading the unnecessary attributes. We perform better than partitioning by leveraging the benefit of grouping attribute together. For Twitter, we observe that partitioning by itself is unable to benefit as queries generally access all the attributes from the tables. The benefit achieved by merging is leveraged by our method. Finally, for IMDb, our method reaps the benefits of unifying partitioning and merging as compared to just one of them.

Design time. We ensure practicality of our method by validating that it completes in a reasonable time. We observe a design time of 250, 20, and 10 seconds for TPC-C, Twitter, and IMDb respectively. The design time is a vast improvement over the naïve method, which we were unable to run to completion in a reasonable time of an hour due to its exponential nature. The design time for the two baselines is considerably less due to a smaller search space. The variation in the design time aptly corresponds to dataset complexity in terms of attribute count. The main offender here is schema creation and the population of table statistics, which is required by the query optimizer for schema evaluation. As schema design is an off-line process the design time in the order of minutes is acceptable.

Workload Generality. We ensure that the gains shown in Figure 5.8 not incidental by running our method against a few variations of the workload for two datasets. Here, we randomly sample 20% of queries from each dataset's workload to obtain eight distinct variations of the workload. From the varied performance gains shown in Figure 5.9, we infer *(i)* quantitative schema design achieves significant gains across different variations and *(ii)* the gain obtained, and hence the optimum schema is dependent on the workload.

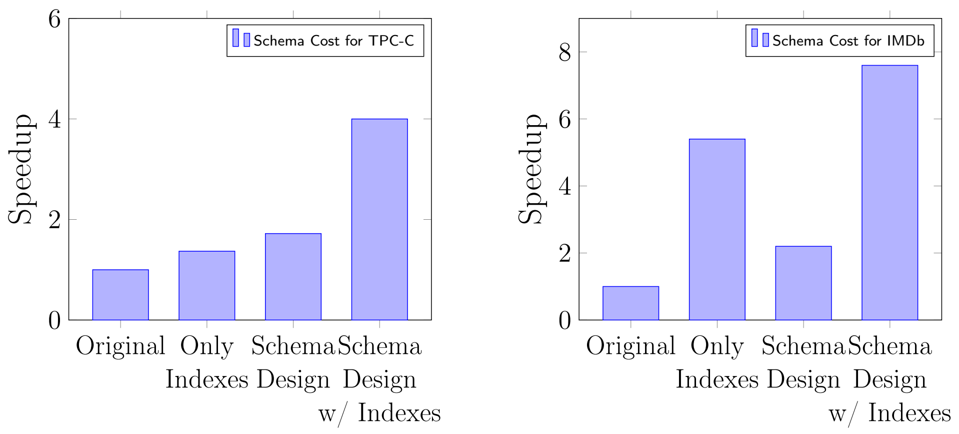


Figure 5.10: Speedup of Indexes vs Schema Design—(a) TPC-C. (b)IMDb.

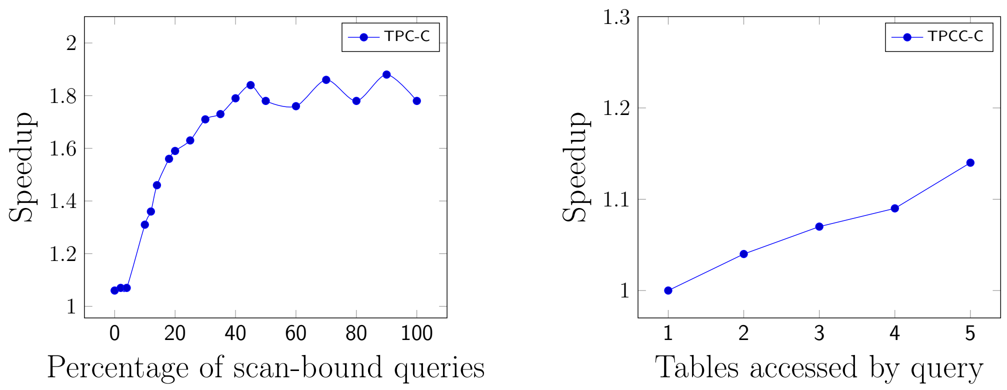


Figure 5.11: Workload Characteristics—(a) Scan-bound queries. (b) Join-bound queries.

Schema Design with Indexes. We now show how our method can work in conjunction with indexes. As the benefit of indexes is orthogonal, when we combine them we observe from Figure 5.10 a higher speedup than either of them. For TPC-C, we utilized the indexes that were present in the original schema, whereas for IMDb we used the Index Tuning Wizard in Microsoft SQL Server to recommend indexes. As IMDb's workload is dominated by selection queries, we observe that the indexes alone provide better speedups. We observe a contrary behavior for TPC-C where the workload has a mix of selection and update queries.

Workload Characteristics. To obtain insight on the gains observed in Figure 5.9, we study a workload's influence on the gains. Based on our cost model, we identify the table access operator and the accessed tables count as the factors that affect a query's cost.

We study the effect of a table's access method on performance gains by segregating the TPC-C workload in two categories, to represent the extremities in terms of table accesses, namely *(i) scan-bound*: access fewer tables and the cost of table access operator is dominating, and *(ii) join-bound*: accesses relatively more tables and join cost is the dominating component of the total cost.

Scan-bound queries generally access a large number of tuples due to which the table access cost is the dominating factor. On the other hand, join-bound queries access relatively more tables, but fewer tuples, and hence the join cost dominates. We obtain the workload by selecting all the join-bound queries and varying the percentage of scan bound ones by randomly sampling them from the original workload. We observe, as shown in Figure 5.11(a), that the gain increases with scan-bound queries. A scan-bound query benefits more compared to a join-bound one due to the reduction in table scan cost due to the decrease in the table's width.

We next study the benefit in terms of speedup with respect to the number of tables accessed by the query as shown in Figure 5.11(b). Since the original workloads do not have queries with all possible table counts, we synthetically generate the workload. To obtain a query that accesses n tables, we first randomly select n naturally join-able tables and then generate a query that performs joins among them. Using a collection of such queries, we obtain the workload. We observe that the gain increases with the table count. Here, since relatively few tuples accessed from each table, the join overhead is significant and the gain is due to its reduction.

5.5.3 Qualitative Evaluation

We now present a qualitative evaluation of the ideas discussed in this paper with a goal to demonstrate the following. *(i)* How we improve the usability of databases by automatically designing a relational schema. *(ii)* How the notion of associations can be used to capture the requirements of a schema and identify redundancy between associations. For this evaluation, we used data organized on a spreadsheet in the form of tables as our starting point. From there we use the tabular structure and the data to identify the associations and constraints, which capture the requirements of the schema. These requirements along with the workload, represented using spreadsheet formulae, are used to quantitatively design a schema.

We consider a small business as a scenario for our evaluation. Here, the owner of a small retail startup currently uses a spreadsheet to manage customers and sales. She would like to routinely perform, such as insert (customers), modify (due dates of invoices), filter (overdue invoices), join (invoices and payments), and aggregate (the total amounts). The updates are done directly to the table in the spreadsheet and reporting, *i.e.*, join and aggregate, is done using spreadsheet formula on the data. As the scale of her operations increase, spreadsheets are unable to handle the scale of the data, and she would like to migrate the data to a relational database, but has trouble designing a database schema that not only serves her requirement, *i.e.*, captures the necessary information, but also is efficient for her reporting, *i.e.*, workload in form of queries.

Our overall work flow can be described as the following steps. *(i) Explicit Association Inference.* Here, we detect the tables within the spreadsheet by identifying tabular regions that are well structured. That is, we looks for areas that have a header corresponding to attribute names and data corresponding to tuples. For example, we detect the Customer table with attributes CustID and CustomerName. We consider the attributes in a table as associated with each other. *(ii) Key Inference.* To detect keys, we check which attribute values are distinct. For example, the CustID has distinct values for the Customer table. Note, that there can be false positives, *i.e.*, detecting an attribute as key when it is actually not—we use user's feedback to eliminate them. *(iii) Implicit Association Inference.* To understand the implicit associations, we consider the similarity of the names and the value of attributes across tables. For example, CustID in Customer and Invoice refer to the same attribute, and and thus indicates a referral constraint. *(iv) Association Redundancy.* Once we have a list of associations, we identify the redundant ones using the inference rules. For example, the Invoice table has attributes CustID and CustomerName—this association is already captured in the Customer table. We eliminate such redundant associations after confirming from the user. *(v) Design schema quantitatively.* Finally, after all the requirements of the schema are captured using the associations, we quantitatively design the optimal schema. Here, we use the spreadsheet formula translated in SQL queries as our workload. For example, join between invoices and payments, which is originally performed using a VLOOKUP on the spreadsheet is translated to a join query.

5.6 RELATED WORK

We review the literature in several aspects. *(i)* With the objective of a "good" schema, most efforts study schema normalization. *(ii)* For the target objective of query performance, we are related to such work in *physical design*: First, and the closest to ours, *schema optimization* attempts to transform a given schema to improve performance. Second, the work on *indexes and views* focuses on building auxiliary data structures, which is orthogonal and complementary to ours. Third, the efforts on rethinking *storage models* beyond the traditional row stores target at improving performance.

1. Schema Normalization The study of schemas mostly focused on *normalization*: Given a schema (or simply a table) R and set of dependencies $\mathcal{F}$, transform R into another schema R' that satisfies certain normal-form criteria with respect to $\mathcal{F}$ (*e.g.*, [77, 88, 21]). The normalization problem is thus fundamentally different from our quantitative schema design: First, we perform "design" starting from only a conceptual model instead of a concrete schema. Second, our metric of a target schema is quantitatively on query performance, and not qualitatively on normal forms. We note that there is a line of efforts on *schema synthesis* [21, 22, 23, 24, 25], which attempts to "synthesize" a normalized schema *w.r.t.* a set of dependencies as the input, without an initial schema. We share the same synthesis approach—but differ in the conceptual requirements—which we start with "essential associations" (see Section 5.2) instead of dependencies—and the objective of quantitative performance.

2.1 Schema Optimization. For query performance, several efforts focus on *schema optimization*: Given a schema R, transform R to another schema R', by partitioning and denormalizing the tables in R, to improve query speed *w.r.t.* a workload. Partitioning [101, 100, 102] splits up a table into multiple fragments to optimize I/O performance. Denormalization [103, 104, 105], a special case of "merging" (our primitive for generating new schemas; Section 5.3), groups attributes or adds redundant data to improve read performance. While we also target query performance, we fundamentally differ in that, *first*, we perform "design" starting from only a conceptual model instead of a concrete schema and, *second*, we consider a richer search space that subsumes partitioning and denormalization. Our problem formulation and approach ensures finding quantitively optimal schemas not limited by an initial schema and the smaller search space.

2.2 Indexes and Views. Indexes [80] and materialized views are auxiliary structures that improve query performance. Chaudhuri et al. [106] presents index selection based on query optimizer's cost estimates for a workload. Agrawal et al. [107] additionally integrates materialized view selection. Indexes and materialized views provide limited benefits in interactive application scenarios, where response time is critical and the original schema might be sub-optimal, due to the overheads for update-heavy workloads [100]. While sharing similar objectives, these efforts are *complementary* to ours—As our approach does not depend on a specific query optimizer, it can accommodate auxiliary structures as long as the optimizer supports them. Our work can thus work in conjunction with theirs.

2.3 Storage Models. Many efforts have proposed to rethink storage structures to tailor to special workloads such as OLAP. For example, C-Store [108, 109] proposes a column-oriented storage. Our schema design addresses the problem of "what attributes" should be logically grouped into tables, rather than their physical storage, which can be accounted for as we simply rely on a system's native query optimizer. Like indexing and views, the work on storage models thus addresses a distinct aspect of physical design and is orthogonal and complementary to our focus.

5.7 CONCLUSION

In this chapter, we establish and solve the problem of schema design by involving a workload in the process. We have developed the notion of attribute association to precisely capture the requirements of a relational schema. This enables us to establish the search space to encompass partitioning and merging. We develop a schema-centric cost model, which enables us to develop an anti-monotonic property to efficiently and correctly prune the search space. Finally, we experimentally demonstrate up to 2x speedups, which is a significant improvement over the current state-of-the-art.

In conclusion, we have developed the notion of attribute association and a versatile quantitative schema design method that significantly improves database performance for a well-defined workload and can be used across different relational database systems. The quantitative schema design techniques discussed in this chapter improve interactivity for DataSpread by optimize the schema of the relational tables stored in the underlying database.

CHAPTER 6: FUTURE WORK: DIRECTED DATA MANAGEMENT

While direct manipulation greatly increases the usability of databases, we discovered that it is not as effective at scale. When dealing with data that is extremely large, direct manipulation is only adequate for certain simple actions, like editing a cell or deleting a row that is already visible, or adding a new row or column. On the other hand, it is hard to envision scrolling through a billion row spreadsheet manually, without some ability to navigate to the rows of interest. For example, our biology collaborators at Mayo Clinic, while thrilled at the ability to examine their large genomics dataset in a spreadsheet, acknowledged that they wouldn't be able to scroll through all of it, limiting its usefulness. Cockburn and others argue that mentally assimilating and manipulating a large information space can lead to cognitive and mechanical burdens on users [110, 111]. It is also hard to expect users to express computation on a billion row spreadsheet, such as dragging a formula through the entirety of a column, which may span all billion rows. In fact, a single relational operator may, in this case, be able to replace a number of formulae proportional to the size of the data. Finally, some formulae can take a long time to execute with no feedback to the user about the status of the computation.

To address these limitations, we identified three extensions to direct manipulation that help us make a substantial leap towards our original research goal. We call this new paradigm *directed data management.*[1] Directed data management extends the direct manipulation principles in various ways (in *italics*)—see Figure 6.1, allowing users *(i)* to interact with a *multi-perspective* continuous representation of the object of interest—that is, allowing operations at various organizations and granularities of the data *(ii)* via *accelerated* actions—allowing users to skip fine-grained steps if necessary, using coarse-grained operations (such as relational algebra and SQL), and *(iii)* by performing operations whose impact is *progressively visible*—that is, the system constantly provides partial results to the user, even for expensive computations.

[1] Directed data management is not just reliant on direct manipulation—computation can be "directed" by the user and the system: users "direct" computation at a higher level if needed, aided by automation, so the system "directs" computation as well.

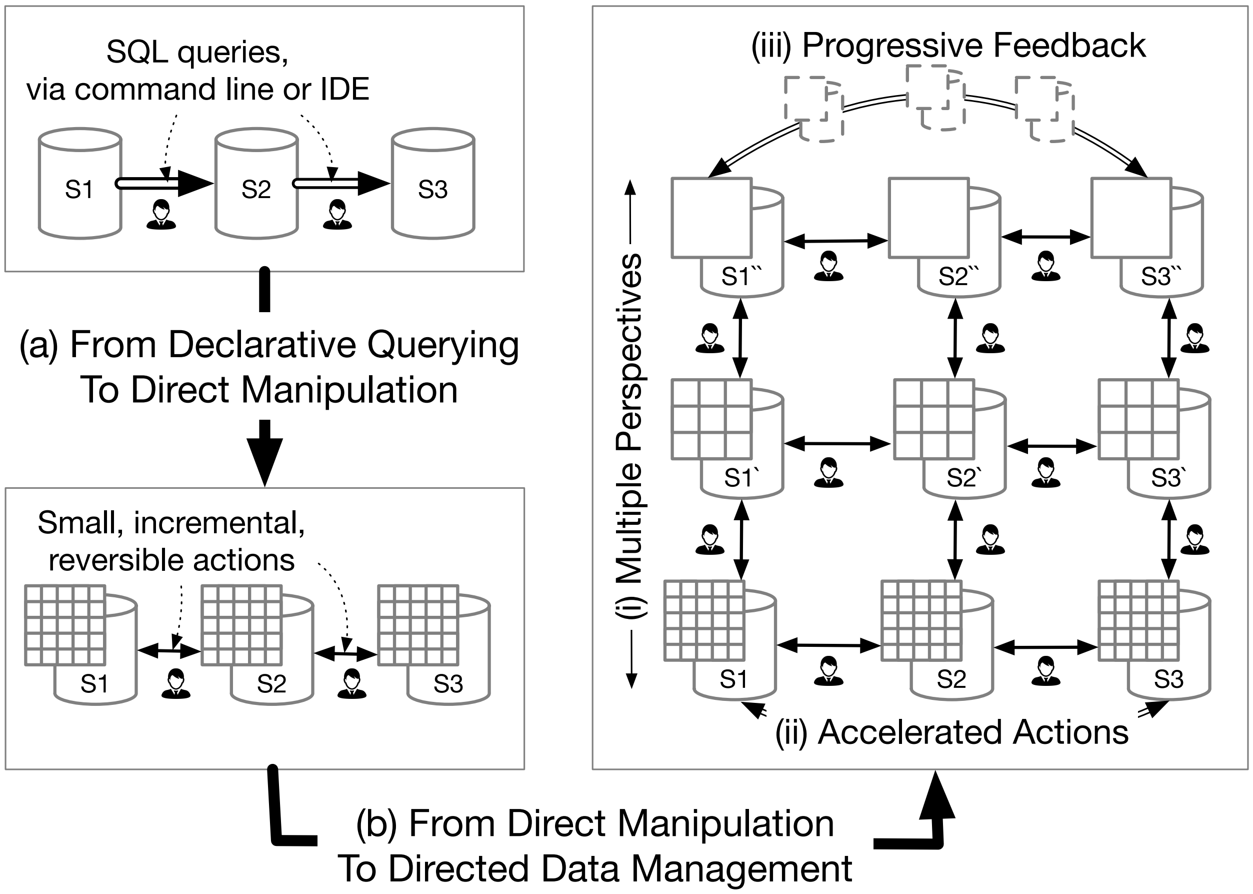

SQL queries,
via command line or IDE
S1
S2
S3
(a) From Declarative Querying To Direct Manipulation
Small, incremental, reversible actions
S1
S2
S3
(iii) Progressive Feedback
(i) Multiple Perspectives
S1``
S2``
S3``
S1`
S2`
S3`
S1
S2
S3
(ii) Accelerated Actions
(b) From Direct Manipulation To Directed Data Management

Figure 6.1: Steps towards Directed Data Management.

6.1 TOWARDS DIRECTED DATA MANAGEMENT

Since a direct manipulation interface is unable to support certain types of operations at scale, we propose a new paradigm called *directed data management* that extends direct manipulation by including support for both user-directed and system-directed analysis of very large datasets. We now discuss the motivation and the underlying challenges.

6.1.1 Enabling Direct Manipulation at Scale

Even though spreadsheets provide direct manipulation capabilities, given the scale of data, such an interface introduces a discontinuity between the information displayed in its limited window, creating a cognitive burden for the users in understanding the overall structure of the data, and navigating through it [111]. We aim to instead allow users to make sense of data using different *perspectives*, reducing the discontinuity, and allowing users to rapidly jump between perspectives; for example, between a bird's eye view of the entire interface, and a close-up, as in online maps. Our insight at addressing this is to allow users to manipulate data using different *perspectives* or via different granularities—this reduces the discontinuity, allowing users to rapidly jump between perspectives; for example, between a bird's-eye view of the entire interface and a close-up, much like the idea adopted by online maps.

Then, to manipulate data at scale, small, incremental actions to accomplish a given task (*e.g.*, manually selecting a range of a million cells, or filling out a column of a million formulae) can be tedious and error-prone. Simplifying or *accelerating such actions* can greatly improve the usability of the system.

Finally, even with actions that are easier to express, sometimes the computation time of certain operations (*e.g.*, a formula that aggregates a billion cells, or a sort operation) makes it impossible to receive feedback immediately. Studies in HCI have shown that delays of even half a second can severely affect the user experience, and may lead to users abandoning the intended task [112]. Thus, there is a need to provide an immediate response to the user through *progressively visible* feedback.

6.2 MULTI-PERSPECTIVE REPRESENTATION

To reduce the impact of information overload with data at scale, we propose to enable data to be presented using multiple perspectives—where each perspective refers to both a way of organizing the data, and a specific granularity (or "zoom" level) for that organization. For example, a realtor, examining a list of Airbnb listings, may choose to view the listings by price or reviews, or both, at various granularities, and may choose to view an aggregate corresponding to the total number of rentals per month. Each perspective offers a unique vantage point. Supporting multiple perspectives can be challenging since it impacts the interface design, as well as the entire database stack.

From the interface standpoint, the primary challenge is to create an interpretable representation that can be used in conjunction with the traditional spreadsheet-like interface, allowing users to interact with both interfaces, enabling rapid exploration and drill-down. Our *navigation panel* (see Figure 2.5) is a step towards this idea by enabling users to work with data at different granularities on a hierarchical interface. Users can view data at different levels by "zooming in and out". *How can we automatically construct this hierarchical interface that adapts to various data types and user needs*? Presently, we use a simple histogram-based equi-depth binning strategy. Automated techniques can be augmented by learning user preferences through examples, as in Excel's flash-fill [113].

From the backend standpoint, *when and how do we construct the hierarchical interface*? One option is to do so *lazily*, when the user explores the data organized in a certain way and granularity. However, this may take a long time—in which case we may need new storage models and indexing schemes for data at various granularities. For this, pre-materialized data cubes, as well as incremental approaches such as database cracking seem promising.

6.3 ACCELERATED ACTIONS

The goal of accelerated actions is to allow users to move to the desired state by skipping a cumbersome sequence of small actions. Such accelerated actions can be either user or system-directed, and can be issued within a given perspective, as well as across perspectives. As a first step, we support SQL queries within a given perspective to allow users to go beyond basic spreadsheet-like direct manipulation to operate on collections of data at a time (as in Figure 2.7). For example, to filter out records that occur between two dates, our realtor can simply use a relational operation, instead of either manually selecting the rows of interest, or using the advanced spreadsheets filtering capabilities that requires multiple interactions. However, to display the result of the relational operation, users still have to refer to the results via an INDEX function that needs to be dragged across a rectangular region (as in Figure 2.7) which can be tedious.

The challenge, therefore, is: *how do we design intuitive and interpretable interactions within a given perspective, complementary to declarative querying*? Our navigation panel introduces new actions to aggregate, format, and organize data within a given perspective. For example, in Figure 2.5, computation of the charts for each price bucket is an accelerated action that is directed by the user that would otherwise require manual selection of the region in each price bucket, followed by formulae that operate on those regions. We can further enrich the interactions by allowing the charts to be used as visual query interfaces in order to support ad-hoc formula specification. For example, the realtor examining Airbnb listings can use a slider to move along the histogram in Figure 2.5 to select different prices within a price bucket, without typing out the corresponding formula.

The next challenge is to support accelerated actions across perspectives. For example, if the user zooms out, DataSpread recomputes the charts from the new perspective. Multi-perspective navigation can be overwhelming and users can lose context due to abrupt changes in view. *How do we design interface cues to provide users context for these transitions*? For example, by displaying a navigation history we can provide users a context of where they were and where they are now. Then, to enable efficient query execution for actions across perspectives, we can adopt view materialization schemes, so that repeated actions are not recomputed, while staying within a storage budget.

There are other preliminary avenues to support accelerated actions. For example, having queries or formulae embedded along with the data can be used by the system to identify errors or recommend reorganization/cleaning actions. Example-driven interactions are another rich avenue whose potential has only been explored within spreadsheets for data extraction [113, 114]. Our investigation of pain points in spreadsheets [12] reveals a lack of knowledge of typical workflows as a primary contributor to errors in formulae: auto-suggestions of next steps, or providing explanations for formulae can greatly increase understanding and efficiency.

6.4 PROGRESSIVELY VISIBLE FEEDBACK

In addition to the interactive computation ideas discussed in Section 2.1.2, our goal is to quickly provide users with approximate results that improve over time [115, 116]. For example, our realtor may want to sort millions of listings by price, or compute an average price of the listings. As the computation happens, the system can display the progress to the user incrementally. However, it is not clear *how to display this information in a way* *that maximizes user understanding and allows users to make decisions early.* Progressively updating the interface while presenting a stable and consistent view can be challenging. A natural approach would be to draw samples on the fly and update the interface periodically, with probabilistic error guarantees. Conveying uncertainty for direct manipulation operations in an interpretable manner is an open problem. One approach for formula computation is to display a progress bar as it is being computed (as the cell itself), with the current best estimate being overlaid on top.

From a query optimization standpoint, *how do we identify optimal sampling strategies for progressive feedback, optimizing for the many computations currently underway*? We need to trade-off the benefit of a sample across these computations. Moreover, traditional progressive approximation approaches [115, 116] only provide error guarantees for summary statistics. However, for operations that impact the position of data, such as *sort*, *how do we capture and display the positional uncertainty of the data*? For example, for a sort operation, we can leverage the cardinality statistics of the sort column to model the positional uncertainty. To be able to access data by position as it is sorted, we need to construct the positional index progressively—one strategy can be to bulk load data in increments, and reconstruct the indexes at each increment. However, such an approach may lead to a high degree of positional error. We need schemes that can incrementally index the data while maintaining the diversity of samples to be displayed to the user.

6.5 CONCLUSION

As this chapter indicates, there are many interesting challenges that remain in making ad-hoc interactions with very large datasets feasible.

CHAPTER 7: RELATED WORK

Our work on DataSpread draws on related work from multiple areas: *(i)* that introduce direct manipulation principles, *(ii)* that enhance database usability, and *(iii)* those that attempt to merge spreadsheet and database functionalities, but without a holistic integration.

1. Direct Manipulation Principles. To support *interactivity*, we adopt *direct manipulation* guidelines that emphasize user control proposed in the field of human-computer interaction, first coined by Shneiderman [5], and further characterized by Hutchins et al., [6]. The principles mandate continuous *representation* of the object of interest, *physical actions* instead of complex syntax, and *small, incremental* units of operations. This thesis aims to realize these principles to achieve the "feeling of directedness" of manipulation for data. Direct manipulation interfaces are not without their caveats as Hutchins [6] explains. They do not easily support repetitive operations and they often expect clear metaphors that map to familiar practices. A creative designer can exploit semantic and articulatory directness to handle newer, more abstract contexts. A relevant example is "data wrangling"—preparing data for analysis. Kandel et al. created an interactive visual specification of data transformation [117], combining elements of direct manipulation and exploration to handle a task traditionally difficult for direct manipulation interfaces—repetition.

2. Making Databases More Usable. There has been a lot of recent work on making database interfaces more user friendly [118, 14]. This includes recent work on gestural query and scrolling interfaces [26, 119, 16, 120, 18], visual query builders [121, 19], query sharing and recommendation tools [122, 123, 27], schema-free databases [124], schema summarization [125], and visual analytics tools [126, 127, 128, 129]. However, none of these tools can replace spreadsheet software which has the ability to analyze, view, and modify data via a direct manipulation interface [5] and has a large user base [4].

3a. One-way Import of Data from Databases to Spreadsheets. There are various mechanisms for importing data from databases to spreadsheets, and then analyzing this data within the spreadsheet. This approach is followed by Excel's Power BI tools, including Power Pivot [10], with Power Query [9] for exporting data from databases and the web or deriving additional columns and Power View [9] to create presentations; and Zoho [130] and ExcelDB [131] (on Excel), and Blockspring [132] enabling the import from a variety of sources including databases and the web. Unlike us, their import is one shot, with the data residing in the spreadsheet from that point on, negating the scalability benefits from the database. Indeed, Excel 2016 specifies a limit of 1M records that can be imported, illustrating that the scalability benefits are lost. Zoho specifies a limit of 0.5M records. Furthermore, the connection to the base data is lost: modifications made at either end are not propagated.

3b. One-way Export of Operations from Spreadsheets to Databases. There has been some work on exporting spreadsheet operations into database systems, such as Oracle [133, 134], 1010Data [135] and AirTable [136], to improve the performance of spreadsheets. However, the database itself has no awareness of the existence of the spreadsheet, making the integration superficial. These techniques do not consider the skew in structures, positional/ordering aspects, as well as the cascading problems caused due to row/column inserts. In particular, positional and ordering aspects are not captured, and user operations on the front-end, *e.g.*, inserts, deletes, and adding formulae, are not supported. Indeed, the lack of awareness makes the integration one-shot, with the current spreadsheet being exported to the database, with no future interactions supported at either end: thus, in a sense, the *interactivity* is lost. Other efforts in this space include that by Cunha et al. [137] to recognize functional dependencies in spreadsheets. Other work has examined the extraction of structured relational data from spreadsheets [138, 139].

3c. Using a Spreadsheet to Mimic a Database. There has been some work on using a spreadsheet to pose as traditional database. For example, Tyszkiewicz [140] describes how to simulate database operations in a spreadsheet. However, this approach loses the scalability benefits of databases. Bakke et al. [141, 142, 143] support joins by depicting relations using a nested relational model. Liu et al. [144] use spreadsheet operations to specify single-block SQL queries; this effort is essentially a replacement for visual query builders. Recently, Google Sheets [3] has provided the ability to use single-table SQL on its frontend, without availing of the scalability benefits of database integration. Excel, with its Power Pivot and Power Query [9] functionality has made moves towards supporting SQL in the front-end, with the same limitations. Like this line of work, we support SQL queries on the spreadsheet frontend, DataSpread's functionality goes beyond this, in representing and manipulating large datasets all on a spreadsheet-like interface.

CHAPTER 8: CONCLUSION

This thesis should convince readers of the importance of the advocated research direction of *unifying spreadsheets with databases for enabling ad-hoc interactive data management at scale*. Our system, DataSpread, a concrete realization of the research, offers a valuable hybrid between spreadsheets and databases, retaining the ease-of-use of spreadsheets, and the power of databases. The research challenges and their solutions discussed in this thesis played a significant role in the realization of DataSpread. Chapter 2 discussed the overall architecture and system development. The remaining three chapters focused on three major research challenges for DataSpread. In Chapter 3, we discussed the first problem of designing a storage engine for persisting spreadsheet data within a relational database. In Chapter 4, we discussed the problem of ensuing interactivity of computationally heavy spreadsheets. In Chapter 5, we have generalized the storage problem by quantitatively designing a relational schema based on a workload. In Chapter 6, we presented our vision that goes beyond direct manipulation, thereby providing a roadmap for future work.

Although DataSpread uses a REACT based interface (originally ZK spreadsheet and Microsoft Excel) as the front-end and a relational database, specifically PostgreSQL, as the back-end database, the challenges and insights discussed in this thesis are not limited to this setting. In fact, any user interface that supports management of big data via an interactive interface will have to deal with similar challenges.

While we have made significant headway towards supporting direct manipulation on large datasets, there are a number of challenges that need additional research and engineering efforts as we discussed in Chapter 6.

REFERENCES

[1] B. Grad, "The creation and the demise of VisiCalc," *IEEE Annals of the History of Computing*, vol. 29, no. 3, pp. 20–31, 2007.

[2] `https://products.office.com/en-us/excel`, "Microsoft Excel, Spreadsheet Software,"

[3] `http:/google.com/sheets`, "Google Sheets (retrieved March 10, 2015),"

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

[5] B. Shneiderman, "Direct Manipulation: A Step Beyond Programming Languages.," *IEEE Computer*, vol. 16, no. 8, pp. 57–69, 1983.

[6] E. L. Hutchins, J. D. Hollan, and D. A. Norman, "Direct Manipulation Interfaces," *Hum.-Comput. Interact.*, vol. 1, pp. 311–338, Dec. 1985.

[7] S. G. Powell, K. R. Baker, and B. Lawson, "A critical review of the literature on spreadsheet errors," *Decision Support Systems*, vol. 46, pp. 128–138, Dec. 2008.

[8] R. R. Panko, "What We Know About Spreadsheet Errors," *J. End User Comput.*, vol. 10, pp. 15–21, May 1998.

[9] C. Webb, *Power Query for Power BI and Excel.* Apress, 2014.

[10] `http://www.microsoft.com/en-us/download/details.aspx?id=43348`, "Microsoft sql server power pivot (retrieved march 10, 2015),"

[11] "Microsoft Excel spreadsheet subreddit." `https://www.reddit.com/r/excel/`.

[12] K. Mack, J. Lee, K. Chang, K. Karahalios, and A. Parameswaran, "Characterizing scalability issues in spreadsheet software using online forums," in *SIGCHI*, 2018.

[13] E. F. Codd, "A Relational Model of Data for Large Shared Data Banks," *Commun. ACM*, vol. 13, pp. 377–387, June 1970.

[14] H. Jagadish, A. Chapman, A. Elkiss, M. Jayapandian, Y. Li, A. Nandi, and C. Yu, "Making database systems usable," in *SIGMOD*, pp. 13–24, ACM, 2007.

[15] B. Shneiderman, "Improving the human factors aspect of database interactions," *ACM Transactions on Database Systems (TODS)*, vol. 3, no. 4, pp. 417–439, 1978.

[16] A. Nandi, "Querying Without Keyboards.," in *CIDR*, 2013.

[17] F. Li and H. Jagadish, "Constructing an interactive natural language interface for relational databases," *VLDB Endowment*, vol. 8, no. 1, 2014.

[18] S. Idreos and E. Liarou, "dbTouch: Analytics at your Fingertips.," in *CIDR*, 2013.

[19] A. Abouzied, J. Hellerstein, and A. Silberschatz, "DataPlay: interactive tweaking and example-driven correction of graphical database queries," in *UIST*, 2012.

[20] "ZK Spreadsheet." `https://www.zkoss.org/product/zkspreadsheet`.

[21] P. A. Bernstein, "Synthesizing third normal form relations from functional dependencies," *ACM Trans. Database Syst.*, vol. 1, no. 4, pp. 277–298, 1976.

[22] R. Fagin, "The decomposition versus synthetic approach to relational database design," in *Proceedings of the Third International Conference on Very Large Data Bases - Volume 3*, VLDB '77, pp. 441–446, VLDB Endowment, 1977.

[23] E. F. Codd, "Further normalization of the data base relational model," *IBM Research Report, San Jose, California*, vol. RJ909, 1971.

[24] E. C. Foster and S. V. Godbole, "Integrity rules and normalization," in *Database Systems*, pp. 57–81, Springer, 2014.

[25] P. A. Bernstein, J. R. Swenson, and D. C. Tsichritzis, "A unified approach to functional dependencies and relations," *Proceedings of the 1975 ACM SIGMOD international conference on Management of data - SIGMOD '75*, p. 237, 1975.

[26] A. Nandi, L. Jiang, and M. Mandel, "Gestural Query Specification," *VLDB Endowment*, vol. 7, no. 4, 2013.

[27] N. Khoussainova, Y. Kwon, M. Balazinska, and D. Suciu, "SnipSuggest: Context-aware autocompletion for SQL," *VLDB Endowment*, vol. 4, no. 1, pp. 22–33, 2010.

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

[29] B. A. Nardi and J. R. Miller, "An ethnographic study of distributed problem solving in spreadsheet development," in *Proceedings of the 1990 ACM conference on Computer-supported cooperative work*, pp. 197–208, ACM, 1990.

[30] J. Blazewicz, K. H. Ecker, E. Pesch, G. Schmidt, and J. Weglarz, *Scheduling computer and manufacturing processes.* springer science & Business media, 2013.

[31] M. Cox, "Inside Airbnb." `http://insideairbnb.com/get-the-data.html`.

[32] J. Callan, M. Hoy, C. Yoo, and L. Zhao, "Clueweb09 data set," 2009.

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

[34] A. Lingas, R. Pinter, R. Rivest, and A. Shamir, "Minimum edge length partitioning of rectilinear polygons.," in *Annual Allerton Conference on Communication, Control, and Computing*, pp. 53–63, 1982.

[35] C. E. L. Cormen, Thomas H. and R. L. Rivest, *Introduction to Algorithms. Cambridge.* MA: MIT, 1990.

[36] V. Raman, B. Raman, and J. M. Hellerstein, "Online dynamic reordering for interactive data processing," in *VLDB*, vol. 99, pp. 709–720, 1999.

[37] V. Raman, B. Raman, and J. M. Hellerstein, "Online dynamic reordering," *The VLDB Journal*, vol. 9, pp. 247–260, Dec. 2000.

[38] P. G. Brown, "Overview of SciDB: Large Scale Array Storage, Processing and Analysis," in *SIGMOD*, pp. 963–968, ACM, 2010.

[39] S. Papadopoulos, K. Datta, S. Madden, and T. Mattson, "The TileDB array data storage manager," *VLDB*, vol. 10, no. 4, pp. 349–360, 2016.

[40] N. Bruno and S. Chaudhuri, "An online approach to physical design tuning," in *ICDE*, pp. 826–835, IEEE, 2007.

[41] N. Bruno and S. Chaudhuri, "To tune or not to tune?: a lightweight physical design alerter," in *VLDB*, pp. 499–510, VLDB, 2006.

[42] S. Idreos, M. L. Kersten, S. Manegold, *et al.*, "Database cracking.," in *CIDR*, vol. 7, pp. 68–78, 2007.

[43] A. Gupta, I. S. Mumick, *et al.*, "Maintenance of materialized views: Problems, techniques, and applications," *IEEE Data Eng. Bull.*, vol. 18, no. 2, pp. 3–18, 1995.

[44] M. Grund and Kothers, "Hyrise: a main memory hybrid storage engine," *VLDB*, vol. 4, no. 2, pp. 105–116, 2010.

[45] J. Schaffner, A. Bog, J. Krüger, and A. Zeier, "A hybrid row-column oltp database architecture for operational reporting," in *BIRTE Workshop*, pp. 61–74, Springer, 2008.

[46] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, *et al.*, "C-store: a column-oriented DBMS," in *VLDB*, pp. 553–564, VLDB Endowment, 2005.

[47] S. Papadomanolakis and A. Ailamaki, "Autopart: Automating schema design for large scientific databases using data partitioning," in *SSDBM*, pp. 383–392, IEEE, 2004.

[48] J. Rao, C. Zhang, N. Megiddo, and G. Lohman, "Automating physical database design in a parallel database," in *SIGMOD*, pp. 558–569, ACM, 2002.

[49] S. Agrawal, V. Narasayya, and B. Yang, "Integrating vertical and horizontal partitioning into automated physical database design," in *SIGMOD*, ACM, 2004.

[50] L. Sun, M. J. Franklin, J. Wang, and E. Wu, "Skipping-oriented partitioning for columnar layouts," *VLDB*, vol. 10, no. 4, pp. 421–432, 2016.

[51] Z. Liu and J. Heer, "The effects of interactive latency on exploratory visual analysis," *IEEE Trans. Visualization & Comp. Graphics (Proc. InfoVis)*, 2014.

[52] C. Williams and L. Caputo, "Excel performance: Improving calculation performance." https://docs.microsoft.com/en-us/office/vba/excel/concepts/excel-performance/excel-improving-calcuation-performance, 2017.

[53] A. Guttman, "R-trees: A dynamic index structure for spatial searching," in *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, SIGMOD '84, (New York, NY, USA), pp. 47–57, ACM, 1984.

[54] J. Culberson and R. Reckhow, "Covering polygons is hard," *Journal of Algorithms*, vol. 17, no. 1, pp. 2 – 44, 1994.

[55] E. L. Lawler, "Sequencing jobs to minimize total weighted completion time subject to precedence constraints," in *Annals of Discrete Mathematics*, vol. 2, pp. 75–90, Elsevier, 1978.

[56] R. W. Conway, W. L. Maxwell, and L. W. Miller, *Theory of scheduling*. Courier Corporation, 2003.

[57] "Spring Framework." https://spring.io/.

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

[59] A. V. Kononov, B. M. Lin, and K.-T. Fang, "Single-machine scheduling with supporting tasks," *Discrete Optimization*, vol. 17, pp. 69 – 79, 2015.

[60] A. Marcus, E. Wu, D. R. Karger, S. Madden, and R. C. Miller, "Crowdsourced databases: Query processing with people," Cidr, 2011.

[61] A. G. Parameswaran, H. Park, H. Garcia-Molina, N. Polyzotis, and J. Widom, "Deco: declarative crowdsourcing," in *Proceedings of the 21st ACM international conference on Information and knowledge management*, pp. 1203–1212, ACM, 2012.

[62] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," *Computer networks and ISDN systems*, vol. 30, no. 1-7, pp. 107–117, 1998.

[63] D. Models, "Excel's smart recalculation engine," 2014.

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

[65] C. Williams and L. Caputo, "Speeding up calculations and reducing obstructions." https://docs.microsoft.com/en-us/office/vba/excel/concepts/excel-performance/excel-improving-calcuation-performance\#speeding-up-calculations-and-reducing-obstructions, 2017.

[66] R. Cilibrasi and P. Vitanyi, "Clustering by compression," *IEEE Transactions on Information Theory*, vol. 51, no. 4, pp. 1523–1545, 2005.

[67] Y. Liu, T. Safavi, A. Dighe, and D. Koutra, "Graph summarization methods and applications: A survey," *ACM Computing Surveys (CSUR)*, vol. 51, no. 3, p. 62, 2018.

[68] S. Navlakha, R. Rastogi, and N. Shrivastava, "Graph summarization with bounded error," in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pp. 419–432, ACM, 2008.

[69] A. Maccioni and D. J. Abadi, "Scalable pattern matching over compressed graphs via dedensification," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 1755–1764, ACM, 2016.

[70] N. Tang, Q. Chen, and P. Mitra, "Graph stream summarization: From big bang to big crunch," in *Proceedings of the 2016 International Conference on Management of Data*, pp. 1481–1496, ACM, 2016.

[71] "Overview of excel tables." `https://support.office.com/en-us/article/overview-of-excel-tables-7ab0bb7d-3a9e-4b56-a3c9-6c94334e492c`.

[72] "Excel databases: Creating relational tables." `https://www.pcworld.com/article/3234335/software/excel-databases-creating-relational-tables.html`.

[73] "Internet movie database.." `http://www.imdb.com`.

[74] J. W. Palmer, "Web site usability, design, and performance metrics," *Information Systems Research*, vol. 13, no. 2, pp. 151–167, 2002.

[75] S. Clemens, "5 Ways To Tell You Have Outgrown Excel." `http://www.insightsquared.com/2011/06/5-ways-to-tell-you-have-outgrown-excel/`.

[76] T. J. Teorey, D. Yang, and J. P. Fry, "A logical design methodology for relational databases using the extended entity-relationship model," *ACM Comput. Surv.*, vol. 18, pp. 197–222, June 1986.

[77] E. F. Codd, "A relational model of data for large shared data banks," *Communications of the ACM*, vol. 13, no. 6, pp. 377–387, 1970.

[78] S. McDonald, Y. Waern, and G. Cockton, eds., *People and Computers XIV - Usability or Else*. London: Springer London, 2000.

[79] S. Madden, R. Morris, M. Stonebraker, and C. Curino, "6.830 Database Systems. Fall 2010. Massachusetts Institute of Technology: MIT OpenCourseWare." https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-830-database-systems-fall-2010/lecture-notes/MIT6_830F10_lec03.pdf.

[80] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price, "Access path selection in a relational database management system," in *Proceedings of the 1979 ACM SIGMOD international conference on Management of data*, SIGMOD '79, (New York, NY, USA), pp. 23–34, ACM, 1979.

[81] H. Garcia-Molina, *Database systems: the complete book.* Pearson Prentice Hall, 2009.

[82] R. Ramakrishnan and J. Gehrke, *Database Management Systems.* Berkeley, CA, USA: Osborne/McGraw-Hill, 2nd ed., 2000.

[83] C. Beeri, A. O. Mendelzon, Y. Sagiv, and J. D. Ullman, "Equivalence of relational database schemes," in *Proceedings of the Eleventh Annual ACM Symposium on Theory of Computing*, STOC '79, (New York, NY, USA), pp. 319–329, ACM, 1979.

[84] R. Hull, "Relative information capacity of simple relational database schemata," in *Proceedings of the 3rd ACM SIGACT-SIGMOD symposium on Principles of database systems*, pp. 97–109, 1984.

[85] a. V. Aho, a. Beeri, and J. D. Ullman, "The theory of joins in relational data bases," *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, vol. 4, no. 3, pp. 297–314, 1977.

[86] C. P. Wang and H. H. Wedekind, "Segment synthesis in logical data base design," *IBM J. Res. Dev.*, vol. 19, pp. 71–77, Jan. 1975.

[87] W. W. Armstrong, "Dependency structures of data base relationships.," in *IFIP Congress*, pp. 580–583, 1974.

[88] E. F. Codd, "Recent investigations in relational data base systems," in *IFIP Congress*, pp. 1017–1021, 1974.

[89] R. Fagin, "Multivalued dependencies and a new normal form for relational databases," *ACM Transactions on Database Systems (TODS)*, vol. 2, no. 3, pp. 262–278, 1977.

[90] R. Fagin, "Normal forms and relational database operators," in *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*, SIGMOD '79, (New York, NY, USA), pp. 153–160, ACM, 1979.

[91] J. Biskup, U. Dayal, and P. A. Bernstein, "Synthesizing independent database schemas," in *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*, SIGMOD '79, (New York, NY, USA), pp. 143–151, ACM, 1979.

[92] E. F. Codd, "Further normalization of the data base relational model," *IBM Research Report, San Jose, California*, vol. RJ909, 1971.

[93] C. Beeri, P. A. Bernstein, and N. Goodman, "A sophisticate's introduction to database normalization theory," in *Readings in Artificial Intelligence and Databases* (J. Mylopolous and M. Brodie, eds.), pp. 468 – 479, San Francisco (CA): Morgan Kaufmann, 1989.

[94] E. F. Codd, "Further normalization of the data base relational model," *IBM Research Report, San Jose, California*, vol. RJ909, 1971.

[95] P. Atzeni, G. Ausiello, C. Batini, and M. Moscarini, "Inclusion and equivalence between relational database schemata," *Theoretical Computer Science*, vol. 19, no. 3, pp. 267 – 285, 1982.

[96] "Postgresql.." `http://www.postgresql.org`.

[97] "Tpc-c benchmark.." `http://www.tpc.org/tpcc/`.

[98] "Amazon.." `http://www.amazon.com/`.

[99] "Twitter.." `http://twitter.com/`.

[100] S. Papadomanolakis and A. Ailamaki, "Autopart: Automating schema design for large scientific databases using data partitioning," in *Scientific and Statistical Database Management, 2004. Proceedings. 16th International Conference on*, pp. 383–392, IEEE, 2004.

[101] S. B. Navathe and M. Ra, "Vertical partitioning for database design: a graphical algorithm," *SIGMOD Rec.*, vol. 18, pp. 440–450, June 1989.

[102] S. Agrawal, V. Narasayya, and B. Yang, "Integrating vertical and horizontal partitioning into automated physical database design," in *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, SIGMOD '04, (New York, NY, USA), pp. 359–370, ACM, 2004.

[103] G. Sanders and S. Shin, "Denormalization effects on performance of RDBMS," in *Proceedings of the 34th Annual Hawaii International Conference on System Sciences, 2001*, p. 9 pp., Jan. 2001.

[104] Z. Wei, J. Dejun, G. Pierre, C.-H. Chi, and M. van Steen, "Service-oriented data denormalization for scalable web applications," in *Proceedings of the 17th international conference on World Wide Web*, pp. 267–276, ACM, 2008.

[105] W. H. Inmon, "Denormalize for efficiency," *Computerworld*, vol. 21, p. 19, Mar. 1987.

[106] S. Chaudhuri and V. Narasayya, "An efficient, cost-driven index selection tool for microsoft SQL server," in *Proceedings of the International Conference on Very Large Data Bases*, pp. 146–155, 1997.

[107] S. Agrawal, S. Chaudhuri, and V. R. Narasayya, "Automated selection of materialized views and indexes in SQL databases," in *Proceedings of the 26th International Conference on Very Large Data Bases*, pp. 496–505, 2000.

[108] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik, "C-store: A column-oriented dbms," in *Proceedings of the 31st International Conference on Very Large Data Bases*, VLDB '05, pp. 553–564, VLDB Endowment, 2005.

[109] A. Lamb, M. Fuller, R. Varadarajan, N. Tran, B. Vandiver, L. Doshi, and C. Bear, "The vertica analytic database: C-store 7 years later," *Proc. VLDB Endow.*, vol. 5, pp. 1790–1801, Aug. 2012.

[110] A. Cockburn, A. Karlson, and B. B. Bederson, "A review of overview+ detail, zooming, and focus+ context interfaces," *ACM Computing Surveys (CSUR)*, vol. 41, no. 1, p. 2, 2009.

[111] S. Kruck, J. J. Maher, and R. Barkhi, "Framework for cognitive skill acquisition and spreadsheet training," *Journal of Organizational and End User Computing (JOEUC)*, vol. 15, no. 1, pp. 20–37, 2003.

[112] Z. Liu and J. Heer, "The effects of interactive latency on exploratory visual analysis," *IEEE TVCG*, no. 1, pp. 1–1.

[113] S. Gulwani, "Automating string processing in spreadsheets using input-output examples," in *ACM SIGPLAN Notices*, vol. 46, pp. 317–330, ACM, 2011.

[114] V. Le and S. Gulwani, "Flashextract: a framework for data extraction by examples," in *ACM SIGPLAN Notices*, vol. 49, 2014.

[115] J. M. Hellerstein, P. J. Haas, and H. J. Wang, "Online aggregation," in *Acm Sigmod Record*, vol. 26, pp. 171–182, ACM, 1997.

[116] S. Rahman *et al.*, "I've seen enough: incrementally improving visualizations to support rapid decision making," *PVLDB*, 2017.

[117] S. Kandel, A. Paepcke, J. Hellerstein, and J. Heer, "Wrangler: Interactive visual specification of data transformation scripts," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pp. 3363–3372, ACM, 2011.

[118] S. Abiteboul, R. Agrawal, P. Bernstein, M. Carey, S. Ceri, B. Croft, D. DeWitt, M. Franklin, H. G. Molina, D. Gawlick, *et al.*, "The lowell database research self-assessment," *Commun. ACM*, vol. 48, pp. 111–118, May 2005.

[119] A. Nandi and H. V. Jagadish, "Guided interaction: Rethinking the query-result paradigm," *VLDB Endowment*, vol. 4, no. 12, pp. 1466–1469, 2011.

[120] M. Singh, A. Nandi, and H. V. Jagadish, "Skimmer: rapid scrolling of relational query results," in *SIGMOD*, pp. 181–192, ACM, 2012.

[121] T. Catarci, M. F. Costabile, S. Levialdi, and C. Batini, "Visual query systems for databases: A survey," *Journal of Visual Languages & Computing*, vol. 8, no. 2, pp. 215–260, 1997.

[122] N. Khoussainova, M. Balazinska, W. Gatterbauer, Y. Kwon, and D. Suciu, "A Case for A Collaborative Query Management System.," in *CIDR*, www.cidrdb.org, 2009.

[123] U. Cetintemel, M. Cherniack, J. DeBrabant, Y. Diao, K. Dimitriadou, A. Kalinin, O. Papaemmanouil, and S. B. Zdonik, "Query Steering for Interactive Data Exploration.," in *CIDR*, 2013.

[124] L. Qian, K. LeFevre, and H. V. Jagadish, "CRIUS: user-friendly database design," *VLDB Endowment*, vol. 4, no. 2, pp. 81–92, 2010.

[125] C. Yu and H. V. Jagadish, "Schema summarization," in *VLDB Endowment*, pp. 319–330, 2006.

[126] S. P. Callahan, J. Freire, E. Santos, C. E. Scheidegger, C. T. Silva, and H. T. Vo, "VisTrails: visualization meets data management," in *SIGMOD*, pp. 745–747, ACM, 2006.

[127] J. Mackinlay, P. Hanrahan, and C. Stolte, "Show me: Automatic presentation for visual analysis," *TVCG*, vol. 13, no. 6, pp. 1137–1144, 2007.

[128] C. Stolte, D. Tang, and P. Hanrahan, "Polaris: A system for query, analysis, and visualization of multidimensional relational databases," *TVCG*, vol. 8, no. 1, pp. 52–65, 2002.

[129] H. Gonzalez, A. Y. Halevy, C. S. Jensen, A. Langen, J. Madhavan, R. Shapley, W. Shen, and J. Goldberg-Kidon, "Google fusion tables: web-centered data management and collaboration," in *SIGMOD*, pp. 1061–1066, ACM, 2010.

[130] `https://www.zoho.com/`, "Zoho Reports (retrieved March 10, 2015),"

[131] `http://www.excel-db.net/`, "Excel-DB (retrieved March 10, 2015),"

[132] `http://www.blockspring.com/`, "Blockspring (retrieved March 10, 2015),"

[133] A. Witkowski, S. Bellamkonda, T. Bozkaya, N. Folkert, A. Gupta, J. Haydu, L. Sheng, and S. Subramanian, "Advanced SQL modeling in RDBMS," *ACM Transactions on Database Systems (TODS)*, vol. 30, no. 1, pp. 83–121, 2005.

[134] A. Witkowski, S. Bellamkonda, T. Bozkaya, A. Naimat, L. Sheng, S. Subramanian, and A. Waingold, "Query by excel," in *VLDB*, pp. 1204–1215, 2005.

[135] `https://www.1010data.com/`, "1010 Data (retrieved March 10, 2015),"

[136] `https://www.airtable.com/`, "Airtable (retrieved March 10, 2015),"

[137] J. Cunha, J. Saraiva, and J. Visser, "From Spreadsheets to Relational Databases and Back," in *SIGPLAN*, PEPM '09, (New York, NY, USA), pp. 179–188, ACM, 2009.

[138] Z. Chen and M. Cafarella, "Automatic web spreadsheet data extraction," in *Proceedings of the 3rd International Workshop on Semantic Search Over the Web*, pp. 1:1–1:8, ACM, 2013.

[139] Z. Chen, M. Cafarella, J. Chen, D. Prevo, and J. Zhuang, "Senbazuru: A prototype spreadsheet database management system," *VLDB*, vol. 6, no. 12, pp. 1202–1205, 2013.

[140] J. Tyszkiewicz, "Spreadsheet as a relational database engine," in *SIGMOD*, pp. 195–206, ACM, 2010.

[141] E. Bakke and D. R. Karger, "Expressive query construction through direct manipulation of nested relational results," in *SIGMOD*, ACM, 2016.

[142] E. Bakke *et al.*, "A spreadsheet-based user interface for managing plural relationships in structured data," in *CHI*, pp. 2541–2550, ACM, 2011.

[143] E. Bakke and E. Benson, "The Schema-Independent Database UI: A Proposed Holy Grail and Some Suggestions.," in *CIDR*, 2011.

[144] B. Liu and H. V. Jagadish, "A spreadsheet algebra for a direct data manipulation query Interface," pp. 417–428, IEEE, Mar. 2009.