A SCALABLE DIRECT MANIPULATION ENGINE FOR POSITION-AWARE PRESENTATIONAL DATA MANAGEMENT

BY

XINYAN ZHOU

THESIS

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

Urbana, Illinois

Adviser:

Professor Kevin Chen-Chuan Chang

ABSTRACT

With the explosion of data, large datasets become more common for data analysis. However, existing analytic tools are lack of scalability and large-scale data management tools are lack of interactivity. A lot of data analysis tasks are based on the order of data, we are proposing the very first positional storage engine supporting persistence and maintenance of orders for large datasets and allow direct manipulation on orders. We introduce a sparse monotonic order statistic structure for persisting and maintaining order. We also show how to support multiple orders and optimize the storage. After that, we demonstrate a buffered storage manager to ensure the direct manipulation interactivity. Last, we show our final system DataSpread which is interactive and scalable. In the end, we hope that our solution can point out a potential direction to support data analysis for large-scale data.

To my parents, for their love and support.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

Data analysis is playing a big role in this data explosion age. Data analytics usually want to visualize the data and directly manipulate the data for analysis. Spreadsheet software, from the pioneering VisiCalc [1] to Microsoft Excel [2] and Google Sheets [3], is one of the most popular data analysis tools which is considered as an interactive tool for organization, analysis and storage of data. However, these tools are not scalable to support the increasing sizes and complexities of data sets, as well as types of analyses. *e.g.*, Excel has lifted its size limits from 65k to 1 million rows, Google Sheets has expanded its size limit to 2 million cells. Database systems, on the other hand, are designed to handle high scalability but lack of interactivity support which requires users to be experts using batch command. Therefore, we want to holistically integrate spreadsheets and databases which is both interactive and scalable. The integration of these two different tools creates a lot of problems.

In this paper, we focus on the following fundamental problems - *How can we efficiently support persisting and maintaining orders for large datasets?* Our first challenge is how to support persisting and maintaining a single order in large scale. Notice that our data might not have a semantic order, which means the order cannot be determined by using some explicit record's value. Therefore, we need a positional mapping that can map the position of the data to the data value to preserve the order. Suppose we have 1000000 rows of records, inserting a single row in the front can lead to an expensive cascading update of the row numbers of all subsequent rows; thus, we must develop a positional mapping structure that allows 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 (e.g., a B+ tree) on the attribute corresponding to row number, the cascading update makes it hard to maintain such an index across edit operations.

Our second challenge is how do we persist and maintain multiple orders. Consider we have 3 different orders, we delete a single row from the first order. Now we need to synchronize this delete on the other two orders. Traditional indexing engine can maintain different orders by lookup their attribute value according to the deleted row. However, all our orders are all maintained based on its own order which is independent of each other. Therefore, we need to maintain a mapping between each order to allow us to trace the same record from all orders.

Last, our last challenge is to ensure interactivity for persisting and maintaining orders in the context of direct manipulation. The previous study from Lindgaard states that people can form basic visual impression very quickly. Although we support large-scale data here, we need a system that satisfies the 0.1-second response time limit which can make users feel like they are directly applying some actions.

Besides the positional awareness problem, there are other integration problems. One problem is *how can we flexibly represent the data within a database*. Unlike structured tables in database, spreadsheet has more freedom of its data layout. Therefore, we need to design a presentational storage engine which can preserve and maintain our data efficiently. As this storage engine is a published work, A more detailed technical support can be found here [4]. We show the related contributions in Chapter 6 which explains our data storage model and the architecture of the entire application.

In this paper, we address the aforementioned challenges in developing a scalable position storage manager for position-aware data. The thesis is organized in the following way. Chapter 2 formalizes our problem and defines the primitives that we aim to support and our strict direct manipulation concept. Chapter 3 introduces our positional mapping and indexing structure to persist and maintain a single order. Chapter 4 explains how we handle multiple order maintenance. Chapter 5 illustrates our auxiliary structure to ensuring the interactivity of our supported primitives. Chapter 6 shows some other related contributions including the overall system architecture. Chapter 7 shows the performance of our structure which is efficient. Chapter 8 presents the related works which inspire us the idea of designing this scalable position storage manager. In Chapter 9, we list down several interesting directions for future research. Last, we summarize our contribution in chapter 10.

CHAPTER 2: PRELIMINARY

To understand our problem, we conduct an empirical study to understand the real-world needs. According to the study, we summarize the primitives we are supporting in this paper and show the setting of our problem. We also introduce the definition of direct manipulation given our problem.

2.1 SCENARIOS

Example 2.1: Consider an admission history analysis task as follow:

1. First, we store the entire history data on the disk.
2. After that, we scroll to some arbitrary positions to see how the data is structured and check whether the imported data is correct.
3. We then copy and paste the data for this year in the end.
4. Once we have the entire data, we can select the whole data and create a summarization view based on attribute country.
5. Suppose we want to further investigate the applications whose applicants are from either China or India. We will select those records whose applicants are from China and move them next to the records whose applicants are from India.
6. Then we perform a sort on this partial records using GPA to get a deeper understanding. Finally, the administration header wants to get the top 100 applicants using the ranking score.

Example 2.2: Consider an analysis task for flight delay as follow: Suppose the flights originating from San Francisco, Seattle, Portland and Los Angeles have high delays because of a strike in an airline with heavy traffic from the west coast.

1. First, we load the entire flight data.
2. Then, we sort the data based on delay.
3. Now we want to look at the most delayed flights so we scroll to the bottom.
4. We repeatedly group flights from different cities together by reordering them together and perform some statistical formulae on them.

5. We finally find out that those highly delayed flights are mostly coming from the above 4 cities.

6. By investigating those 4 cities airport reports, we find out that the actual cause of this high delay is because of a strike.

Example 2.3:

```
SELECT Table1.AccountId, Table1.AccountData FROM Table1 INNER
   JOIN Table2 ON Table1.AccountId = Table2.AccountId WHERE
   Table2.AccountDate Is NULL
```

```
SELECT staff_id FROM comments WHERE dept_no = ‘G33’ or dept_no
   = ‘G44’ ORDER BY staff_id
```

2.2 PRIMITIVES

Based on the previous examples and our study of common operations on positions, we selected the following five primitives as our target direct manipulation operations.

- **Fetch:** This operation refers to looking up k records starting from any arbitrary position p. For example, when you load the data, you need to fetch the first k records for the representation. Scrolling to a position and page up/down will also require a fetch operation to fetch the current window.
- **Insert:** This operation refers to inserting k records at any arbitrary position p. For example, Insert some missing records in the middle of the data will require an insert operation.
- **Delete:** This operation refers to deleting k records at any arbitrary position p. A deletion of the redundant records is an example which will require a delete operation.
- **Move:** This operation refers to moving k records from one arbitrary position p to another arbitrary position p'. Cut and paste is one simple operation that is considered as a move operation.
- **Sort:** This operation refers to sorting k records between any two arbitrary position p and p' with respect to some semantics. From the above examples, we can see that reorder based on some criteria is really common. Reordering a subset of the data based on some attribute value is an example of this sort operation.

2.3 SETTINGS

In this paper, we focus on solving the problem in the settings below:

- **Dataset:** There are different types of datasets in data analysis tasks, the tabular dataset is one of the most common datasets. When we considered large data set, usually they are structure into the tabular data set. Therefore, we take large tabular dataset as our primary target datasets.
- **Storage:** Many research work focus on theory assumes that everything can fit into memory. However, in practice, the computer always have limited memory. In this paper, we are presenting a practical solution to our problem. Therefore, our storage model is a multi-tiered storage model which takes the disk as the main storage and uses the limited memory storage as a temporary storage.

2.4 DIRECT MANIPULATION

The concept of *Direct Manipulation(DM)* was first introduced by Shneiderman in the early 1980s, at the time when the dominant interaction style was the command line. This concept has been widely applied to the modern interface design, which most of the analytic tools adapted. According to Shneiderman, there are four characteristics of the DM: (I)Continuous representation of the object of interest, (II) Physical actions instead of complex syntax, (III)Continuous feedback and reversible, incremental actions and (IV)Rapid learning. We take these characteristics to define our direct manipulation setting. Given users can only visualize a fixed size window of data, we define an operation to be a direct manipulation if and only if the operation is performed on some records which users manually selected during a short period and some results of this operation which can be used to validate the action should be immediately reflected to the current window. We formalize our definition of direct manipulation below:

Consider a direct manipulation operation is updating both a window state and storage state (contains both memory and disk storage). A window state can be either consistent state (showing the correct result) or inconsistent state (showing incorrect result or no result). We define three types of storage state: a consistent state (disk storage has the updated result), a partial-consistent state(memory storage has the updated result, disk storage not updated), and an inconsistent state (memory storage has some partial updated result or not updated, disk not updated). We define a direct manipulation operation if and only if it can reach a consistent window state and a consistent or partial-consistent storage state regardless of the entire data size.

CHAPTER 3: SINGLE ORDER

3.1 POSITIONAL MAPPING

We found that the cascading updates are caused by direct positional mapping which maps the tuples identifiers directly to its position. To avoid direct position mapping, we introduce the idea of *positional mapping*: we do not store positions but instead store what we call *positional mapping keys*. These positional mapping keys p are proxies that have a one-to-one mapping with the positions r, *i.e.*, $p \rightleftarrows r$. Formally, positional mapping $\mathbb{P}$ is a bijective function that maintains the relationship between the position and positional mapping keys, *i.e.*, $\mathbb{P}(r) \to p$. Here, even though the positional mapping keys do not correspond to the positions, we can fetch the n^{th} record by scanning the positional mapping keys in an increasing order while maintaining a running counter to skip $n-1$ records. Now, we need to assign each positional mapping key with a value.

3.1.1 Unique Positional Mapping Keys.

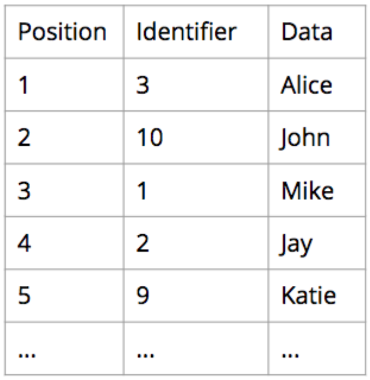
One naive approach is to assign each arbitrary positional mapping key with a unique aribitrary value, *i.e.*, for two arbitrary positions r_i and r_j, if $r_i \neq r_j$ then $\mathbb{P}(r_i) \neq \mathbb{P}(r_j)$. For example, consider the ordered list of items shown in Figure 3.1.

3.1.2 Monotonic Positional Mapping Keys.

Another approach towards positional mapping is to have positional mapping keys monotonically increase with position, *i.e.*, for two arbitrary positions r_i and r_j, if $r_i > r_j$ then $\mathbb{P}(r_j) > \mathbb{P}(r_i)$. For example, consider the ordered list of items shown in Figure 3.2. Here, the identifier is the positional mapping key. The monotonic positional mapping keys require that there is always a unique value between two consecutive positional mapping keys.

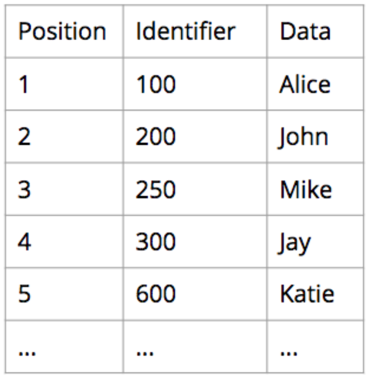
3.2 POSITIONAL INDEXING

As discussed in Section 4, positional mapping structure can solve the problem of cascading updates. To efficiently support operations on spreadsheets, we want to develop an indexing structure which works with the positional mapping structure. Besides, we want to maintain



Position	Identifier	Data
1	3	Alice
2	10	John
3	1	Mike
4	2	Jay
5	9	Katie
...	...	...

Figure 3.1: Unique Positional Mapping.



Position	Identifier	Data
1	100	Alice
2	200	John
3	250	Mike
4	300	Jay
5	600	Katie
...	...	...

Figure 3.2: Monotonic Positional Mapping.

the same efficiency as traditional index *e.g.*, a B-Tree index, which the complexity to access or update an arbitrary position is $\mathcal{O}(\log N)$.

3.2.1 Order Statistic Tree

As we known, a simple B-tree support lookup operation based on keys. Here, we introduce an order static tree as our index structure which is a variant of the B-tree that supports lookup based on positions. To turn a simple B-tree into an order statistic tree, the nodes of the tree need to store one additional value, which is the size of the subtree rooted at that node (i.e., the number of nodes below it). To lookup a position, we use the cumulative size of the leaf node as its position.

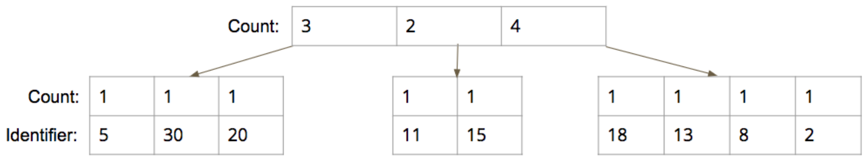


Figure 3.3: Order Statistic Tree.

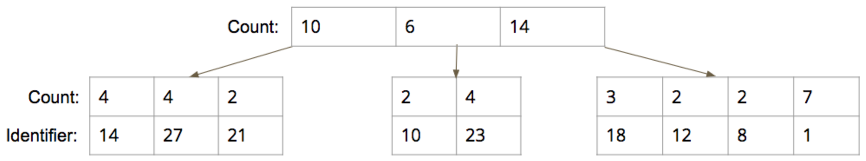


Figure 3.4: Sparse Monotonic Order Statistic Tree.

3.2.2 Sparse Monotonic Order Statistic Tree

Our order statistic tree is a dense structure which captures each position as an element. Observed that people usually insert a brunch of records at a time, we might not need to store the positions one by one. Notice that our order static tree supports both unique positional mapping keys and monotonic positional mapping keys. One benefit of the monotonic sequence is that itself captures the position of the sequence. Since monotonoic sequence mapping always requires there exists a unique value in between two continuous key, a simple solution is using float numbers as our keys' value since integer numbers can not have infinit gaps in between. However, using the float number as the identifer of the tuple is not as effecient as using interger number as the identifier of the tuple and. Therefore, we developed sparse monotonic order statistic tree which is an optimized structure that compresses the node size by grouping those continuous monotonic positional mapping keys. Now, each node can represent more than one positional mapping keys. A leaf node that contains a count ct greater than 1 and mapping key value v means that this node contains ct positional mapping keys starting from the v. Take Figure 4.4 as an example, if we are looking for the second position, we will look at the first leaf node which has count 4 and positional key value starting at 14. From that, we know that the actual positional mapping key value is 14 + 1 = 15 since the mapping key value is maintained in a continuous monotonic sequence.

Lookup operation.

Suppose we are performing a lookup operation at position n.

(i) We start from the root node and lookup for position n.

(ii) For a node, we identify the child to follow, by subtracting the children count cumulatively from n till the remainder is negative. We update our lookup position n' by substracting all the previous children counts and move to the child node.

(iii) Looking at the new node, we lookup for the new position n'.

(iv) We repeat the previous steps (ii) and (iii) till we reach the leaf node and find the element contains the position we are looking for.

Once we reach the leaf element we use the following steps to lookup the actual element.

(i) If the element is a single-key element, we directly use the key value as our lookup result.

(ii) Otherwise, we are having a multi-keys element. We calculate the corresponding position n' of this element, by substracting all the previous elements' count. Now, knowing the new position n', we can get the actual key value which should be key value + n'.

Insert operation.

Suppose we are performing an insert operation inserting m records at position n.

(i) we first perform the same lookup protocal till we get the leaf element v containing our insert position.

(ii) If the leaf element v is a single-key element, we create an element on the left of this element.

(iii) If the leaf element v is a multi-keys element, we split this element v into two elements v and w which v captures the count before position n and w captures the count after position n. If position n is the starting position or the end position, we can avoid creating v or w accordingly. Now, we can create a new element in between element v and element w.

(iv) For the created element, if $m = 1$, we create a single-key element. If $m > 1$, we create a multi-keys element with the key value using the current maximum key value + 1.

(v) Last, we update all the correponding nodes for the new children count involving split and rebalance.

Delete operation.

Suppose we are performing a delete operation deleting m records at position n.

(i) we first perform the same lookup protocal till we get the leaf element v containing our insert position. We perform a linear scan using count m to get all the elements that sits between position n and $n + m$.

(ii) For the first and last element, if it is a multi-keys element, we split it into two elements v and w which v captures the count before the position and w captures the count at and after the position. If the position is the starting position or the end position, we can avoid creating v or w accordingly.

(iii) Now we can delete all the elements within the range.

(iv) Last, we update all the correponding nodes for the new children count involving merge and rebalance.

CHAPTER 4: MULTI ORDER

From the previous positional mapping and indexing, we are able to update or look-up a value based on a position efficiently. Now consider we have two users working on the same data with different orders O_1 and O_2. Let's just consider the basic operations for update data which are lookup, insert, delete and update. Suppose one user perform these operations using O_1, we need to find the corresponding position of the data and change it accordingly. Now look-up and update are already handled and insert can be done with a design of appending the record at the end of O_2. However, the delete operation can not be performed since, given the position of O_1, we don't know which position in O_2 should be deleted. Therefore, we need to maintain a table to help us find the node that contains the same positional mapping key across all orders.

4.1 KEY TABLE

Key Table maps a data set to a set of pointers that point to the same data in different orders of the same table. Since every positional mapping key is a unique value, we maintained a hash table for the dataset that takes the positional mapping key as the hash key and an array of leaf node pointers of each order. Therefore, given a position of one order, we can retrieve all the different positions of the other orders.

4.2 BACKWARD POINTERS

Now we have a Key Table that can help us retrieve the leaf node that contains the corresponding position. However, our structure keeps a cumulative statistic which an update on the leaf node needs to be reflected all the way up to the root node. Therefore, we need to allow a backward traversal from leaf to root. To achieve this, we add backward pointers to our proposed structure. For each node, we maintain a backward pointer that points to its parent node. The complexity of maintaining these backward pointers is shown in APPENDIX I.

4.3 OPTIMIZED MULTI-ORDER STRUCTURE

By having the Key Table and Backward Pointers, now we can persist and maintain multiple orders. Say we delete a data set from one order; the result should be reflected on all orders of the given table. First, we delete and retrieve the data set. Next, using Key Table, we locate the deleted value in the leaf nodes of other orders. And finally, we delete the newly found data and update the root nodes by traversing from leaf to root node with the backward pointers.

However, a single update using one order would require a change of all the other order's structure. From our observations, the reordering would only occur for a small chunk of data of the whole table. Our goal is to minimize the storage of different orders and allow updates efficiently.

We introduce a dual structure which includes a base order and the changes which are stored in Delta. The base order is the proposed indexing structure we mentioned above. To store the difference, We use an enhanced sparse monotonic order statistic indexing structure which is similar to the proposed one. Here, instead of storing the count of elements, we store a pair of (c,s) for each node which c represents the count of elements and s represents the shift of position to look-up back in the base order. In addition, we also store a pointer which either points to the base order or a newly inserted set of data.

For an original semantic order, Delta will only have one node with c as the total number of data and s as zero, since we didn't change. And the pointer will point to the original semantic order, which is itself. Say we want to move n rows from the middle of the table to the end. This is the equivalent of deleting the rows in the middle and then inserting them in the end. When we delete the rows in the middle, we split the single node of Delta in two at the position of the deletion, and insert a new node in between them. The two nodes resulting from the splitting will have c of row count in their portion, and s of zero. While in the newly inserted node will have c of zero and s of n. Since we deleted n rows, all the rows that come after them are shifted up for n positions. All three nodes' pointers will point to the original semantic order. Next, we insert a new node in the back with c of n, s of -n, and the pointer will point to the newly inserted array of data, completing the operation. Thus, our simple update of moving a block of rows from the middle to the end is stored in the Delta structure with only four nodes, which is considerably more efficient than storing a new order.

4.3.1 Generic B+Tree

The generic B+Tree structure incorporates different types of B+Tree implementations, such as key-based B+Tree and order statistic B+Tree. The most direct benefit of this tree structure is the simplification of the B+Tree code; instead of programming a new B+Tree class for every type, we simply add a new type of statistic, an indexing method for the tree. As mentioned above, key-based and count-based are examples of such statistics. Furthermore, the generic B+Tree also allow us to combine several statistics. One such example is the Delta structure mentioned earlier, which combines two statistics count and shift as it's indexing method. This provides the flexibility needed for further implementations similar to Delta.

4.3.2 Structure

We develop an enhanced statistic hierarchical indexing structure for positional mapping. Here, we have a similar structure to the proposed hierarchical indexing structure. Instead of storing the count of elements, we store a pair of (c,s) for each node which c represents the count of elements and s represents the shift of position to lookup back in the base order. Similar to a hierarchical indexing structure of order m, our structure satisfies the following. (a) Every node has at most m children. (b) Every non-leaf node (except root) as at-least $\lceil \frac{m}{2} \rceil$ children. (c) All leaf nodes appear in the same level. (d) All leaf nodes should contain a value or the value can be lookup from the base structure. Again similar to B+tree, we ensure the invariant by either splitting a node into two when the number of children overflow or merging two nodes into one when the number of children underflow. This ensures that the height of the tree is at most $\log_{\lceil m/2 \rceil} N$.

By enhancing a B+tree structure to store with each link to a sub-tree the count of elements and the total shift value stored in that whole sub-tree (instead of just the count), we can support efficient position based operations. The enhanced statistic hierarchical structure of the B+tree enables us to gain efficiency, while dealing with insert, delete, and fetch operations.

Lookup operation.

To access the n^{th} value, we will first lookup the position n from th Delta using the following steps.

(i) We start from the root node of Delta.

(ii) For a node, we identify the child to follow, by subtracting the children count cumulatively from n till the remainder is negative. This steps adjusts the value of n and moves a level down the tree.

(iii) We repeat the previous step till we reach the leaf node, from where we obtain n^{th} element from the leaf node.

Once we reach the leaf node we use the following steps to lookup the actual element.

(i) If there is a element corresponding to position n which contains a key value in Delta, we use this key value as our lookup result.

(ii) Otherwise, we calculate the shift total in front of n and lookup the (n-shift)$^{\mathrm{th}}$ position from the base order.

Since we traverse the tree vertically, the complexity of lookup is $\mathcal{O}(\log N)$.

Insert operation.

For an insert operation of position n, we first obtain the leaf node in Delta using the count and keep track of all the nodes visited in the process. After which, we use the following steps:

(i) If there exists a single-key element which is corresponding to the n^{th} position, we create a new element on the left.

(ii) If the n^{th} position sits in between the range of a node v, we split the node v so that we can create a new node w for the nth position.

(iii) For the created node w, we store the new value there. We set the count of the created node to 1 and the shift to 1 and increment the children count and shift value for the visited nodes.

Since this required only at most two new nodes creation in addition, the complexity of the add operation is $\mathcal{O}(\log N)$.

Update operation.

For an update operation of position n, we first obtain the leaf node in Delta using the count and keep a track of all the nodes visited in the process. After which, we use the following steps:

(i) If there exists a single node which is corresponding to the n^{th} position, we update the node with the new value.

(ii) Otherwise, if the n^{th} position sits in between of the range of a node v, we split the node v so that we can create a new node w for the nth position.

(iii) For the created node w, we store the new value in there. We set the count of the created node to 1 and the shift to 0.

(iv) We update the children count and shift value accordingly for the visited nodes.

Since this required only at most one new node creation in addition, the complexity of the update operation is $\mathcal{O}(\log N)$.

Delete operation.

For a delete operation of position n, we first obtain the leaf node in Delta using the count and keep a track of all the nodes visited in the process. After which, we use the following steps:

(i) If there exists a single node which is corresponding to the n^{th} position, we directly delete the old item from the leaf node and update the children count and shift value for the visited nodes.

(ii) Otherwise, if the nth position sits in between of the range of a node, we first split the node so that we can create a single node for the n^{th} position. We set the count of that single node to 0 and the shift of position to 1.

(iii) We update the children count and shift value accordingly for the visited nodes.

Since this only required at most two new nodes creation in addition, the complexity of the delete operations is still $\mathcal{O}(\log N)$.

CHAPTER 5: INTERACTIVITY FOR DIRECT MANIPULATION

As the theoretical time complexity shown in the previous two chapters, we support all operations in $\mathcal{O}(\log N)$. However, this complexity time is still relevant to the size of the data which is not identical if the data is super large. As we observed some characteristics of direct manipulation, we want to develop a tool that can support those direct manipulation operations regardless of the size of the data. In other words, we want the theoretical time complexity of those direct manipulation operations to be the constant time or close to constant time in practice. We define the theoretical setting for time complexity here:

Given a dataset of N records, an operation performed on a selection of at most M records in total, the machine has a limited constant memory size that can hold at most K records and users can only see a fixed number of W records.

In this chapter, we first introduce a cursor pointer which help us to utilize the specific setting of direct manipulation and improve the performance of our fetching operation. After that, we developed a buffer manager and a clipboard storage which utilize the memory storage to achieve the goal of support direct manipulation operations in near constant time.

5.1 CURSOR POINTER

In the real-world scenarios, people like to group the data they are interested in together so that they can compare those data. However, we found that our proposed solution always fetch data, no matter what data we have previously fetched. Here, we introduce a current window cursor pointer which allows us to fetch data more efficiently if the new fetching position is close to the current window. In APPENDIX II, we show the proof that in the worst case, when the distance between the current window pointer and the new fetching position is N, we still have our theoretical fetching time complexity as $\mathcal{O}(\log N)$. The current window cursor pointer always points to the leaf node that contains the current position mapping key. Now suppose we change our window to a new position, we first using the previous window cursor pointer to access the leaf node. Given the new fetching position, we can calculate the distance between two pointers which tells us how many records we should look back or skip. The two cases of look back and skip are handled slightly different and we will show how to handle them in details.

5.1.1 Skip

In the case of skip, we first scan the current leaf node starting from the previous position and return the new position mapping key if contained. If the distance is further than the distance between the previous position and the last position of the current leaf node, given that we have maintained backward pointers for each node, we can access its parent node. Now we scan this parent node again and if the new position sits in between, we go to the corresponding leaf node to retrieve the positional mapping key. Repeatedly, in the worst case, we will go all the way up to the root and perform the same lookup as a normal lookup to find the new position.

5.1.2 Forward

In the case of forward, we first scan forward the leaf node starting from the previous position and return the new position mapping key if contained. If the distance is further than the distance between the previous position and the first position of the current leaf node, we access its parent node. Now we can scan forward this parent node again and if the new position sits in the node, we go to its corresponding leaf node to retrieve the positional mapping key. Repeatedly, similar to the case of skip, in the worst case, we will go up to the root and perform the same lookup to find the new position.

5.2 BUFFER MANAGER

To access the data on the disk storage in $\mathcal{O}(\log N)$, we are actually performing *log* N random I/O read/writes. Since random access is way too expensive as compared to in-memory access, we want to utilize the memory storage. Given that the size of the memory storage is limited, we need to design an optimized buffer manager to decide which data to be preserved in memory. Notice that analysts usually perform tasks on the data they are currently viewing is selected. Therefore, the data that people might want to work with should be the priority data that we keep in the memory which memory access is more efficient than disk access. Besides, analysts might temporally change the dataset order based on different needs of the task. Therefore, keeping these changes in the buffer can benefit from saving them to the disk. Our in-memory buffer data structure is similar to the Delta structure in the previous chapter. In addition to the structure, our buffer manager will always preserve the current window data with the buffer. If the size of the in-memory data structure reaches the buffer storage limit, the buffer manager will take out the most untouched data keep the in-memory structure.

Besides fetching the data, we also need to preserve the changes in the buffer first and later flush to the disk. As we mentioned previously, random I/O writes is expensive. Therefore, by grouping the changes together in the buffer and flush them together to the disk can give us a performance boost. As we discussed in the previous section, we can use the logical representation of the operation in our buffer structure to optimize our preserved data. The work of the flushing mechanism design is not finished and the insights are discussed in the future work section.

5.3 CLIPBOARD

As we explained in the preliminary chapter, direct manipulation operations are usually performed on selected data. In other words, when people select some data, we can expect that people will perform some operations on them. Notice our buffer manager has already fetched the data when people select, we don't need to fetch from the disk again when performing operations on them. However, people may not immediately perform the task on their selected data which our buffer manager may have decided to clean up the selected data. Thus, we introduce a clipboard in-memory storage to preserve the selected data. When people select the data, the clipboard will preserve a copy of data from the buffer storage. If the user selects another range of data, the clipboard will be replaced with the new data. If the user wants to do multi-selections, the user can use the shift key to specify the continuous of selection. This clipboard storage guarantees that those direct manipulation operations can be done in subliminal time which can be considered interactive.

CHAPTER 6: RELATED CONTRIBUTION

Although the solution we proposed is a general solution that is independent any specific interface and back-end storage, we have applied our solution to DataSpread, which holistically integrate spreadsheets as a front-end interface with databases as a back-end data-store, with dual objectives (i) allowing users to manipulate data from 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) operating on datasets not limited by main memory—thereby addressing the limitation of spreadsheets. While developing DataSpread is a multi-year vision, we have already made significant headway, with a functional prototype (see http://dataspread.github.io). Below is my main contribution of this project containing both research and engineering works.

6.1 HYBRID MODEL

In addition to the positional awareness problem, another problem is how do we store the presentational information within a database. A user may manage several table-like regions within a spreadsheet, interspersed with empty rows or columns, along with formulae. Here, we first describe our primitive data models represent trivial solutions for spreadsheet representation with a single table. Then, we introduce a hybrid model which is a combination of the primitive data models. Before we describe these data models, we discuss a small wrinkle that affects all of these models. Our positional accessing solution is applied to each models to capture the cell's position. 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:

6.1.1 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: $\mathsf{ROM}(\underline{RowID}$, $Col1$, ..., $Colc_{max})$—we illustrate the ROM representation of Figure 6.5 in Figure 6.3(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

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

Figure 6.1: Row-Oriented Model for Figure 6.5.

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

Figure 6.2: Column-Oriented Model for Figure 6.5.

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

Figure 6.3: Row-Column-Value Model for Figure 6.5.

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.

6.1.2 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. The schema for COM is: COM($\underline{ColID}$, $Row1$, ..., $Rowr_{max}$). Figure 6.3(b) illustrates the COM representation of Figure 6.5. 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.

6.1.3 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 6.5 is provided in Figure 6.3(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

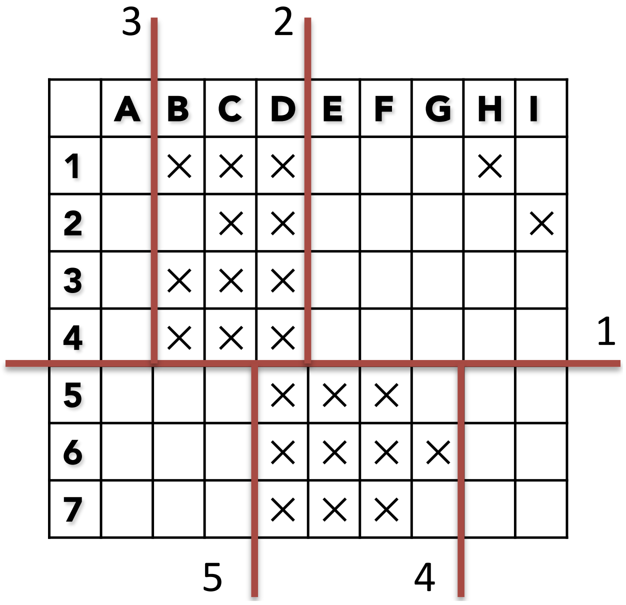


Figure 6.4: Hybrid Data Model: Recursive Decomposition.

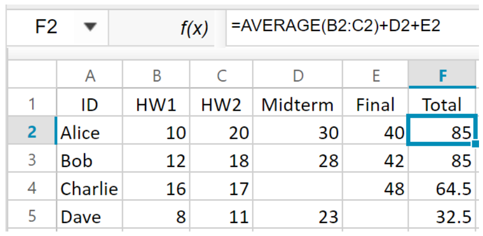
cost of recording and retrieving the row and column numbers for each cell as compared to ROM and COM, and has a much larger number of tuples. RCV is also efficient when it comes to retrieving specific cells at a time.

6.1.4 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.

6.1.5 Hybrid Model

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*. 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 6.4, 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.



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 6.5: DataSpread Screen Shot

6.2 DATASPREAD ARCHITECTURE

To realize interactive, scalable data access by integrating relational databases and spreadsheets, we have implemented a fully functional prototype as a web-based tool using the open-source ZK Spreadsheet frontend [5] on top of a PostgreSQL database. The prototype along with documentation and user guide can be found at http://dataspread.github.io. Along with standard spreadsheet features, the prototype supports all the spreadsheet-like and database-like operations listed below. Screenshots of DataSpread in action can be found in Figure 6.5.

Figure 6.6 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. We handle TOM as a special case of ROM. The hybrid translator is responsible for mapping the different regions on a spreadsheet to corresponding data models. ROM/TOM, COM, and RCV translators service getCells by using the tuple pointers, obtained from the *positional mapper*, to fetch required tuples. For a hybrid model, the mapping from a range to model is stored as *metadata*. The hybrid translator services getCells and other operations by identifying the responsible data models and delegating the call to them. 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. Other operations such as cell updates are performed by the hybrid model in a similar fashion. 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* along with *positional mapping indexes*, which map row/column numbers to tuple pointers, and *metadata*, which records information about the hybrid data model.

Spreadsheet-like Operations

We now describe the spreadsheet-like operations from our user survey.

(i) 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 6.5. Similarly, *formula evaluation* also accesses one or more ranges of cells.

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

(iii) 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: (a) insertRowAfter(row), (b) insertColumnAfter(column), (c) deleteRow(row) and (d) deleteColumn(column).

Database-like Operations

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

(i) 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.

(ii) 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.

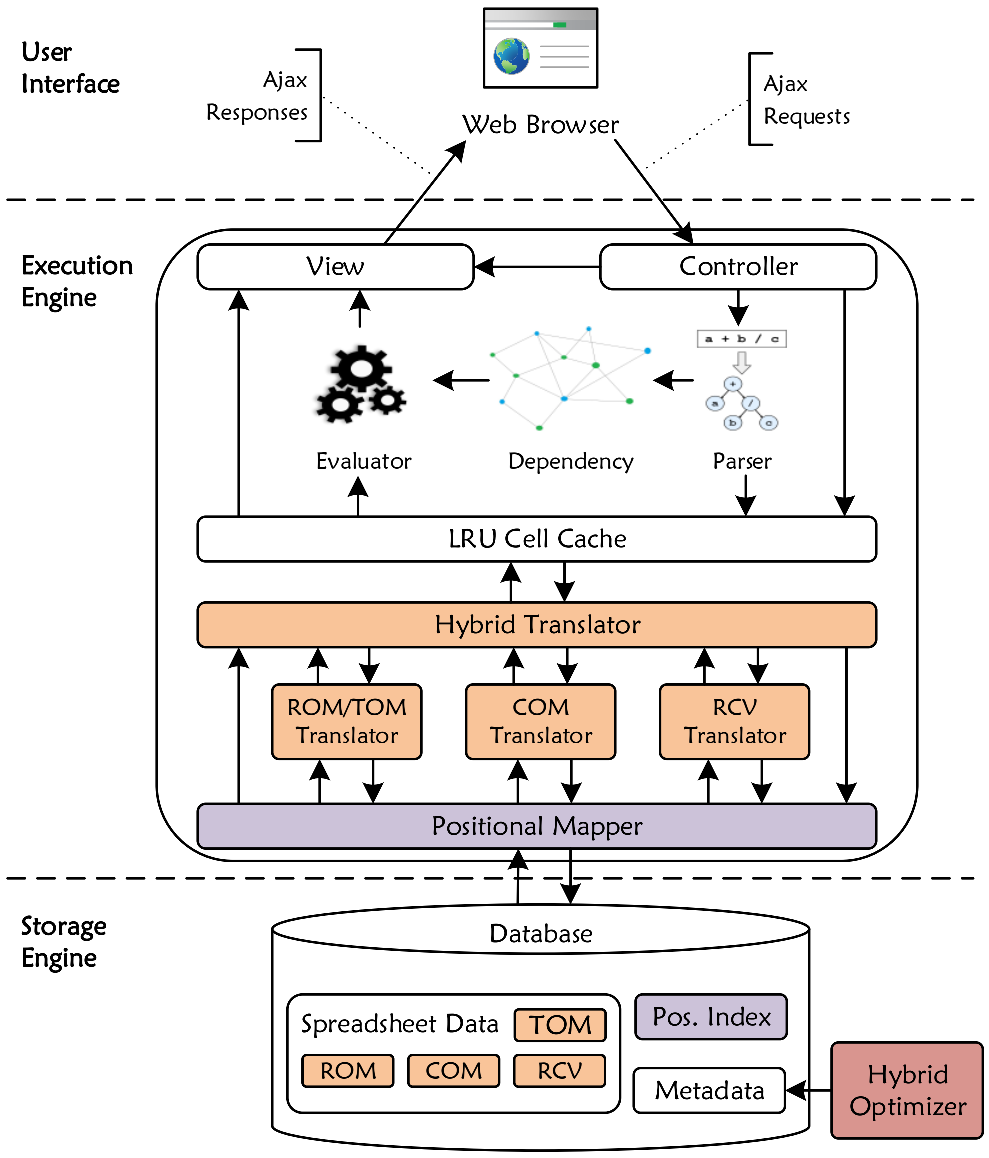


Figure 6.6: DataSpread Architecture.

6.2.1 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.

6.2.2 Relational Operations.

Since DataSpread is built on top of a traditional relational database, it can seamlessly support SQL queries, via the sql function. In addition, we support relational operators via the following spreadsheet functions: union, difference, intersection, crossproduct, join, filter, project, and rename. 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. Since the input and output of all these functions is a table, they can be arbitrarily nested to obtain complex expressions.

CHAPTER 7: EXPERIMENTAL EVALUATION

We perform an extensive experimental study to compare the efficiency performance of our positional addressing methods, under different settings. In this section, we describe these experimental settings and explain the results of our experimental evaluation.

7.1 EXPERIMENTAL SETUP

We implement our data models and the associated positional addressing methods on top of a PostgreSQL (version: 9.5) database.The database has been configured with default parameters. We run all experiments on a workstation with the following settings; Processor: `AMD A10-5700 APU 3.40 GHz`, RAM: `16 GB`, Operating System: `Ubuntu 14.04`. Our test scripts are single-threaded applications developed in `Java`.

7.2 EVALUATION OF POSITIONAL ADDRESSING

In this set of experiments, we compare the efficiency performance of different positional addressing methods. Specifically, we contrast between (i) direct positional mapping with b-tree index (row number stored as-is), (ii) relative positional mapping with counted b-tree index, (iii) relative positional mapping with sparse monotonic counted b-tree index. In this set of experiments, we test the following three operations varying the number of tuples: (a) Select: looking the values of ten tuples based on position, (b) Insert: inserting ten values at arbitrary positions, and (c) Delete: deleting ten values at arbitrary positions.

7.2.1 Results

Figure 7.1 summarizes the results corresponding to the performance of the three positional addressing methods. We see that the storing the row number as-is performs well for select, but time increases linearly with data size for insert and delete operations. This is expected, as this method requires updating row number for subsequent rows after the inserted/deleted rows. We see that both relative positional mapping structure performs very well *for all operations* and performance does not get degrade even with data sizes of 10^9 tuples. Contrasting with other data structures, relative positional mapping with sparse monotonic counted b-tree index performs all the three operations in few milliseconds, which makes it practical.

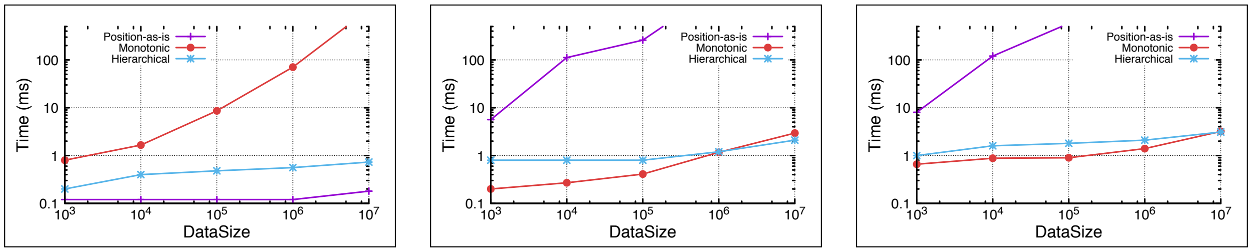


Figure 7.1: Positional Mapping - (a) Select. (b) Insert. (c) Delete.

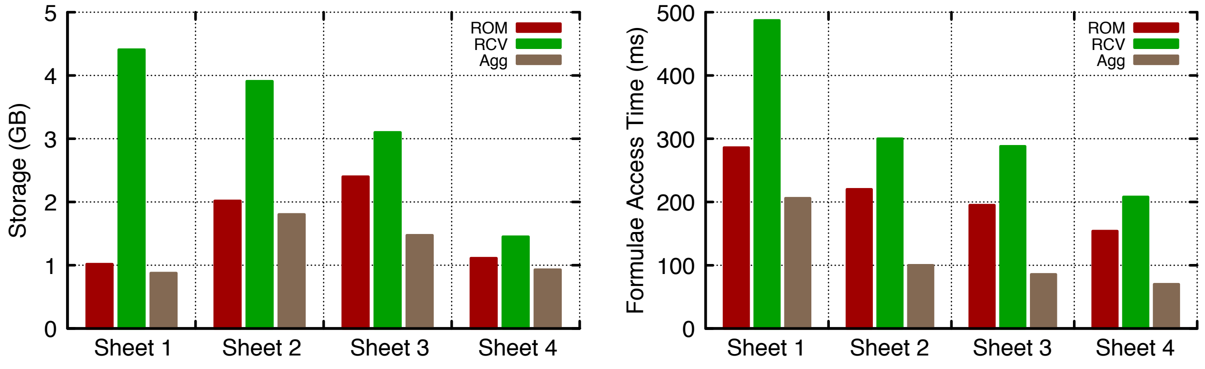


Figure 7.2: Synthetic sheets (a) Storage. (b) Access time. Agg is a hybrid model using some aggressive greedy algorithm.

7.3 STORAGE EVALUATION ON SYNTHETIC DATASET

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 7.2(a) and 7.2(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 Hybrid is better than ROM, which is better than RCV; as density is decreased, RCV's performance becomes closer to ROM. Hybrid performs the best, providing substantial reductions of up to 50-75% of the time taken for access with ROM or RCV.

7.3.1 Results

We observe that ROM significantly optimizes the storage as compared to to the RCV by reducing the number of tuples. Hybrid takes a step further as compared to ROM by splitting the spreadsheet into smaller tables to cover the dense regions thereby reducing the overhead due to storage of nulls columns. The Agg data model also provides significant performance benefits for formulae accesses by reducing the size (number of attributes) of the accessed table.

CHAPTER 8: RELATED WORK

We described our vision for DataSpread in an earlier demo paper [6]. Our work draws on related work from multiple areas; we review papers in each of the areas, and describe how they relate to DataSpread. We discuss 1) efforts that enhance the usability of databases, and 2) those that attempt to merge the functionality of the two paradigms, but without a holistic integration.

8.1 MAKING DATABASES MORE USABLE

The database community has repeatedly lamented the lack of interface research [7, 8]. Much research has emerged in recent years, however, most of this research does not support direct data manipulation in the manner spreadsheets do, rendering them unsuitable for interactive data management (as opposed to browsing or querying). This includes recent work on gestural query and scrolling interfaces [9, 10, 11, 12], visual query builders [13, 14], query sharing and recommendation tools [15, 16, 17], schema-free databases [18], schema summarization [19], and visual analytics tools [20, 21, 22, 23].

8.2 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, inluding Power Pivot [24], with Power Query [25] for exporting data from databases and the web or deriving additional columns and Power View [25] to create presentations; and Zoho [26] and ExcelDB [27] (on Excel), and Blockspring [28] (on Google Sheets [29]) enabling the import from a variety of sources including the databases and the web. Typically, the import is one-shot, with the data residing in the spreadsheet from that point on, negating the scalability benefits derived from the database. Indeed, Excel 2013 specifies a limit of 1M records that can be analyzed once 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: any modifications made at either end are not propagated.

8.3 ONE WAY EXPORT OF OPERATIONS FROM SPREADSHEETS TO DATABASES

There has been some work on exporting spreadsheet operations into database systems, such as the work from Oracle [30, 31] as well as startups 1010Data [32] and AirTable [33], to improve the performance of spreadsheets. However, the database itself has no awareness of the existence of the spreadsheet, making the integration superficial – in particular, without the database being aware of (a) the user window, it is cannot prioritize for its execution over other computation (b) positional and ordering aspects, it cannot support queries on that efficiently (c) user operations on the front-end, e.g., inserts, deletes, moving formulae around, it cannot react to user events. 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.

8.4 USING A SPREADSHEET TO MIMIC A DATABASE

There has been efforts to use a spreadsheet as an interface for posing traditional database queries. There is nothing preventing us from applying traditional database queries in our front-end as well. One effort in this space is by Tyszkiewicz [34], who describes how to simulate database operations in a spreadsheet. However, this approach loses the scalability benefits of relational databases. There has also been some isolated work augmenting spreadsheets by modifying the spreadsheet interface to support joins *e.g.*, Bakke et al. [35, 36], by depicting relations using a nested relational model. Another effort is that by Liu et al. [37], where the goal is to use spreadsheet operations to specify single-block SQL queries; this effort is essentially a replacement for visual query builders. The authors compare against visual query builders in their evaluation, as opposed to a holistic integration. Recently, Google Sheets [29] 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 [25] functionality has made moves towards supporting SQL in the front-end, with the same limitations.

CHAPTER 9: FUTURE WORK

This chapter lists down several directions for further study.

1. In chapter 5, we have introduced a simple buffer manager which take the benefit of the memory to support direct manipulation operations. A more complex manager can be designed to solve the following questions: (a) What's the optimal partition of the entire memory size for both the buffer storage and the clipboard storage? (b) What is the optimal distance which triggers the flush of the changes to the disk? Should we flush all the changes to the disk? (c) Can we have an optimized flushing structure for the changes? Do we have to merge all the changes to the existing structure? Can we have a multi-layer merging mechanism?

2. In the context of direct manipulation, we are just focusing on the basic primitives. Based on the current system we design, sorting is still an expensive operation. Therefore, supporting sorting with direct manipulation will be an interesting topic for research. Moreover, there exists other operations that we didn't cover in this paper. For example, such common operations like filter or vlookup are not considered in our study. An extensive study of these operations can also make our project more useful.

CHAPTER 10: CONCLUSION

In this paper, we present a positional-awareness storage engine to efficiently support different orders on large data sets with direct manipulation. Our solution provides the scalability and collaboration capabilities. We proposed a positional mapping structure for avoiding cascading updates, and augmented our mechanisms with auxiliary indexing structures to maintain efficient lookup and update. These techniques help us reduce latency by an order of magnitude. Furthermore, we propose a mechanism to efficiently support maintaining different orders across same data set. We also design a buffer manager which utilize the limited memory storage and allows us to interactively support direct manipulation. Finally, we apply these techniques in DataSpread with the goal to support interactivity and scalable at the same time. Our experiment study that our optimizations help users operate on large data. In conclusion, our solution offers a new path to work with massive data.

REFERENCES

[1] D. Bricklin and B. Frankston, “Visicalc 1979,” *Creative Computing*, vol. 10, no. 11, p. 122, 1984.

[2] “Microsoft excel,” http://products.office.com/en-us/excel.

[3] “Google sheets,” https://www.google.com/sheets/about/.

[4] M. Bendre et al., “Towards a holistic integration of spreadsheets with databases: A scalable storage engine for presentational data management,” in *ICDE*. IEEE, 2018.

[5] “ZK Spreadsheet,” https://www.zkoss.org/product/zkspreadsheet.

[6] M. Bendre et al., “Dataspread: Unifying databases and spreadsheets,” *VLDB Endowment*, vol. 8, no. 12, pp. 2000–2003, Aug. 2015. [Online]. Available: http://dx.doi.org/10.14778/2824032.2824121

[7] S. Abiteboul et al., “The lowell database research self-assessment,” *Commun. ACM*, vol. 48, no. 5, pp. 111–118, May 2005. [Online]. Available: http://doi.acm.org/10.1145/1060710.1060718

[8] H. V. Jagadish et al., “Making database systems usable,” in *SIGMOD*. ACM, 2007. [Online]. Available: http://dl.acm.org/citation.cfm?id=1247483 pp. 13–24.

[9] A. Nandi, L. Jiang, and M. Mandel, “Gestural Query Specification,” *VLDB Endowment*, vol. 7, no. 4, 2013. [Online]. Available: http://web.cse.ohio-state.edu/~jianglil/files/gestureQuerySpecification.pdf

[10] A. Nandi and H. V. Jagadish, “Guided interaction: Rethinking the query-result paradigm,” *VLDB Endowment*, vol. 4, no. 12, pp. 1466–1469, 2011. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.227.8243&rep=rep1&type=pdf

[11] A. Nandi, “Querying Without Keyboards.” in *CIDR*, 2013. [Online]. Available: http://www.cidrdb.org/cidr2013/Papers/CIDR13_Paper37.pdf

[12] M. Singh, A. Nandi, and H. V. Jagadish, “Skimmer: rapid scrolling of relational query results,” in *SIGMOD*. ACM, 2012. [Online]. Available: http://dl.acm.org/citation.cfm?id=2213858 pp. 181–192.

[13] A. Abouzied, J. Hellerstein, and A. Silberschatz, “DataPlay: interactive tweaking and example-driven correction of graphical database queries,” in *UIST*. ACM, 2012. [Online]. Available: http://dl.acm.org/citation.cfm?id=2380144 pp. 207–218.

[14] 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. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1045926X97900379

[15] N. Khoussainova et al., "A Case for A Collaborative Query Management System." in *CIDR*. www.cidrdb.org, 2009. [Online]. Available: http://dblp.uni-trier.de/db/conf/cidr/cidr2009.html#KhoussainovaBGKS09

[16] U. Cetintemel et al., "Query Steering for Interactive Data Exploration." in *CIDR*, 2013.

[17] N. Khoussainova et al., "SnipSuggest: Context-aware autocompletion for SQL," *VLDB Endowment*, vol. 4, no. 1, pp. 22–33, 2010. [Online]. Available: http://dl.acm.org/citation.cfm?id=1880175

[18] L. Qian, K. LeFevre, and H. V. Jagadish, "CRIUS: user-friendly database design," *VLDB Endowment*, vol. 4, no. 2, pp. 81–92, 2010. [Online]. Available: http://dl.acm.org/citation.cfm?id=1921075

[19] C. Yu and H. V. Jagadish, "Schema summarization," in *VLDB Endowment*, 2006. [Online]. Available: http://dl.acm.org/citation.cfm?id=1164156 pp. 319–330.

[20] S. P. Callahan et al., "VisTrails: visualization meets data management," in *SIGMOD*. ACM, 2006. [Online]. Available: http://dl.acm.org/citation.cfm?id=1142574 pp. 745–747.

[21] J. Mackinlay, P. Hanrahan, and C. Stolte, "Show me: Automatic presentation for visual analysis," *TVCG*, vol. 13, no. 6, pp. 1137–1144, 2007.

[22] C. Stolte et al., "Polaris: A system for query, analysis, and visualization of multidimensional relational databases," *TVCG*, vol. 8, no. 1, pp. 52–65, 2002.

[23] H. Gonzalez et al., "Google fusion tables: web-centered data management and collaboration," in *SIGMOD*. ACM, 2010. [Online]. Available: http://dl.acm.org/citation.cfm?id=1807286 pp. 1061–1066.

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

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

[26] https://www.zoho.com/, "Zoho Reports (retrieved March 10, 2015)."

[27] http://www.excel-db.net/, "Excel-DB (retrieved March 10, 2015)."

[28] http://www.blockspring.com/, "Blockspring (retrieved March 10, 2015)."

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

[30] A. Witkowski et al., "Advanced SQL modeling in RDBMS," *ACM Transactions on Database Systems (TODS)*, vol. 30, no. 1, pp. 83–121, 2005. [Online]. Available: http://dl.acm.org/citation.cfm?id=1061321

[31] A. Witkowski et al., "Query by excel," in *VLDB*, 2005. [Online]. Available: http://dl.acm.org/citation.cfm?id=1083733 pp. 1204–1215.

[32] https://www.1010data.com/, “1010 Data (retrieved March 10, 2015).”

[33] https://www.airtable.com/, “Airtable (retrieved March 10, 2015).”

[34] J. Tyszkiewicz, “Spreadsheet as a relational database engine,” in *SIGMOD*. ACM, 2010. [Online]. Available: http://dl.acm.org/citation.cfm?id=1807191 pp. 195–206.

[35] E. Bakke et al., “A spreadsheet-based user interface for managing plural relationships in structured data,” in *CHI*. ACM, 2011. [Online]. Available: http://dl.acm.org/citation.cfm?id=1979313 pp. 2541–2550.

[36] E. Bakke and E. Benson, “The Schema-Independent Database UI: A Proposed Holy Grail and Some Suggestions.” in *CIDR*, 2011.

[37] B. Liu and H. V. Jagadish, “A spreadsheet algebra for a direct data manipulation query Interface.” IEEE, Mar. 2009. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4812422 pp. 417–428.