M.S. THESIS

RBForest: Efficient In-Memory Format for Sparse Matrices on Dynamic Workloads

RBForest: 동적 워크로드 상의 희소행렬을 위한 효율적인 인-메모리 포맷

BY

Wonhyeon Kim

February 2023

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

RBForest: Efficient In-Memory Format for Sparse Matrices on Dynamic Workloads

RBForest: 동적 워크로드 상의 희소행렬을 위한 효율적인 인-메모리 포맷

지도 교수 문 봉 기

이 논문을 공학석사 학위논문으로 제출함
2022년 11월

서울대학교 대학원
컴퓨터공학부
김원현

김원현의 공학석사 학위논문을 인준함
2023년 1월

위 원 장 이 상 구 (인)

부위원장 문 봉 기 (인)

위 원 강 유 (인)

Abstract

Sparse matrices only store nonzero entries with their positional information. Thus, while dense matrices typically store all of their entries in a one-dimensional array, sparse matrices have various choices of internal data structures to use. So there can be many different in-memory formats where sparse matrices are managed.

Various sparse matrix formats have been proposed, and they have advantages in different ways. Formats that support fast linear algebra operations cannot handle matrix updates quickly and vice versa. However, several use cases of sparse matrices are highly dynamic. So the sparse matrices are expected to periodically process analytic queries, including linear algebra operations, while quickly handling a stream of matrix updates.

Thus, one of the mainstream strategies is to manage a sparse matrix in an update-friendly format during matrix updates and convert it to a format for fast linear algebra operations when we need to process analytic queries. To make the best use of this strategy, this paper proposes a new sparse matrix format called *RBForest*. The *RBForest* manages one red-black tree per row to reduce the tree re-balancing cost after the insertion or deletion of nonzero entries. It also manages a hash table to enable immediate access to its nonzero elements, reducing the initial search cost for all types of matrix updates. Our evaluations show that the *RBForest* performs more efficiently on real dynamic workloads than previously proposed formats that adhere to the same strategy.

Keywords: Sparse Matrix, Matrix Update, CSR Format

Student ID: 2021-20552

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Sparse matrices are widely used in many real-world applications, including social networks, financial analysis, and more [1, 2, 3, 4, 5]. Unlike their dense counterparts, sparse matrices only store nonzero entries with their positional information. Thus, while dense matrices typically store all of their entries in a one-dimensional array, sparse matrices have various choices of internal data structures to use. As a result, various in-memory sparse matrix formats exist where the nonzero entries of the matrix are stored in different data structures in different ways.

Entries of matrices can change over time, and we call this *Matrix Update*. Since a sparse matrix stores only nonzero values, changing its entry may cause an update on its internal data structures. For example, if a zero-valued entry changes its value to nonzero, it must be newly inserted into the matrix's data structure since it was not previously there. Similarly, an existing nonzero entry must be deleted if its value changes to zero because we don't need it anymore. Thus, matrix updates of a sparse matrix can be highly complex depending on its internal data structures.

However, many use cases of sparse matrices are highly dynamic. For example, when a sparse matrix is storing social graph data, the matrix can be used to mine periodic cliques [6] or answer temporal reachability and time-based path queries [7] while continuously accepting changes in relationships between the vertices. In these workloads, sparse matrices are expected to handle a stream of matrix updates while periodically answering analytic queries, including linear algebra (LA) operations. Thus, a sparse matrix should quickly process both LA operations and matrix updates to perform efficiently on dynamic workloads.

Many sparse matrix formats have been proposed, and depending on the characteristics of the internal data structures, formats for sparse matrices can be categorized into two big categories [8]. The first category includes the formats with strengths in fast LA operations. These formats typically store their nonzero entries in a continuous memory area, thus maximizing their spatial locality [9]. However, these formats suffer from high inefficiency in matrix updates because existing entries must be shifted along the continuous memory area, which takes, for a sparse matrix of N nonzero entries, $O(N)$ time in the worst case. One of the most popular formats in this category is the Compressed Sparse Row (CSR) [10, 11]. For the rest of this paper, we will use the CSR format as a representative of the other formats in the same category without loss of generality. It is legitimate because all the features and advantages of the CSR format covered by this paper can be similarly applied to other formats in the same category.

The second category includes the formats for fast matrix updates. These formats use update-friendly data structures such as a red-black tree, hash table, or linked list. However, these formats are not good at fast LA operations because nonzero entries are stored in a possibly non-continuous memory area,

increasing the possibility of cache miss during LA operations. Popular examples of this category are SciPy's [12] Dictionary of Keys (DOK), List of Lists (LIL), and Armadillo's [13] Red-Black Tree (RBT).

Thus, a fast performance on both LA operations and matrix updates is not achievable in a single format. Therefore, to efficiently manage sparse matrices in dynamic workloads, we can manage them in an update-friendly format while accepting changes to their entries. Then we can periodically convert them to CSR format when analytic queries, including LA operations, are requested. By doing this, we can exploit the advantages of both categories of sparse matrix formats, and this is one of the mainstream strategies for managing sparse matrices in dynamic workloads. Popular scientific libraries like SciPy and Armadillo adhere to this strategy.

To make good use of this strategy, we need an update-friendly format that can be converted to the CSR format quickly. If conversion to CSR format is not efficiently performed, latency can be introduced in periodic matrix analysis, including LA operations, no matter how fast the matrix updates are. Thus, matrix updates and CSR conversion should be processed quickly to analyze sparse matrices in dynamic use cases efficiently.

In this paper, we propose *RBForest* (RBF), which is the format for sparse matrices, performs better on dynamic workloads than the proposed formats. Instead of storing all nonzero entries of the matrix in one red-black tree, as in Armadillo's RBT, it manages one red-black tree for each row containing nonzero entries for the row. By doing that, we can reduce the average height of red-black trees, which noticeably reduces the tree re-balancing cost after the insertion or deletion of nonzero entries. And it also manages a hash table

whose key is the (*row*, *col*)-coordinate, and the value is the pointer to the red-black tree node that corresponds to it. It reduces the cost of searching nonzero entries to be inserted, modified, or deleted. Conversion to CSR format can be done by doing in-order traversal on each red-black trees. It takes $O(N)$ time which is the minimum time complexity required to perform CSR conversion. We showed that our proposed format performs better on dynamic workloads with real datasets than DOK, LIL, and RBT formats.

The rest of this paper is organized as follows. In Chapter 2, we illustrate the backgrounds of our work in detail. Chapter 3 introduces previously proposed formats (DOK, LIL, RBT) and their drawbacks. Chapter 4 presents our RBF format and analyzes its strength over proposed formats. Chapter 5 presents the experimental results on both synthetic and real datasets, and Chapter 6 concludes this paper and suggests future works.

Chapter 2

Backgrounds

In this chapter, we describe the matrix update of sparse matrices in detail and introduce two categories of sparse matrix formats based on the characteristics of their internal data structures.

2.1 Matrix Update

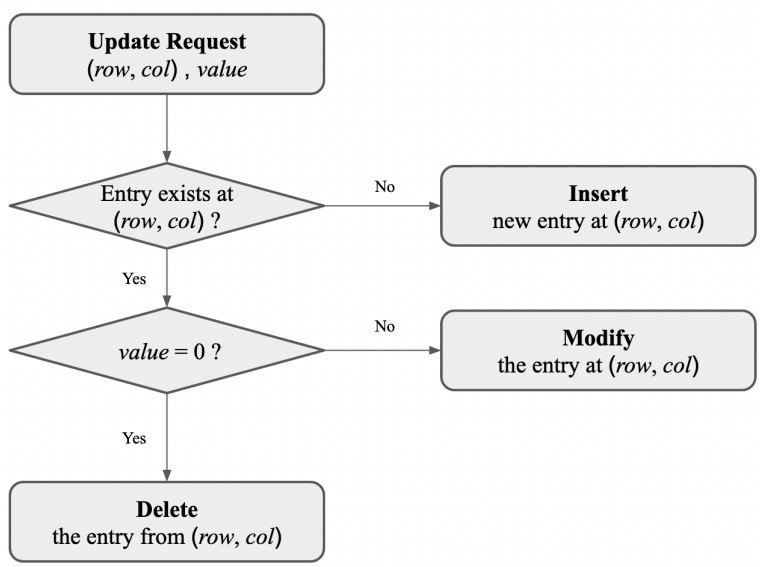


Figure 2.1 Matrix Update Procedures

Figure 2.1 shows how matrix update occurs in sparse matrices. Matrix receives an update request as (*coordinate*, *value*)-pair, on which the matrix set the value of its entry corresponds to the given *coordinate* with the given *value*. When an update request on a specific coordinate arrives, a sparse matrix figures out whether the corresponding nonzero entry exists in its data structures. If the entry does not exist, it is newly inserted into the matrix's data structures with the requested value. If the entry exists, the matrix determines whether to modify or delete it depending on the requested value. If the requested value is nonzero, modify the entry with the requested value, and if the requested value is zero, delete the entry from the matrix's data structures. Therefore, to support fast matrix updates, a sparse matrix should be able to search for the target entry quickly and handle the updates on its data structures quickly.

2.2 Categories of Sparse Matrix Formats

Depending on the characteristics of the internal data structures, sparse matrix formats can be categorized into two big categories: a format for fast LA operations and a format for fast updates [8].

2.2.1 Formats for Fast LA Operations

For a sparse matrix to show optimal performance in LA operations, it must provide spatial locality between its nonzero elements by storing them in a continuous memory area [9]. By doing this, the matrix can reduce cache miss on repeated sequential access through its nonzero elements. Examples in this category include the Compressed Sparse Row/Column format (CSR/CSC), Compressed Diagonal Storage (CDS), Jagged Diagonal Storage (JDS), Blocked Compressed Row format (BCSR), and much more [10, 11, 14, 15, 16, 17].

Among these examples, the CSR format is widely used due to its simplicity and efficiency [11], which consists of the following three arrays:

- *Data* for nonzero data values,
- *Indices* for corresponding column indices,
- *Row Offset* for starting index where the corresponing row starts.

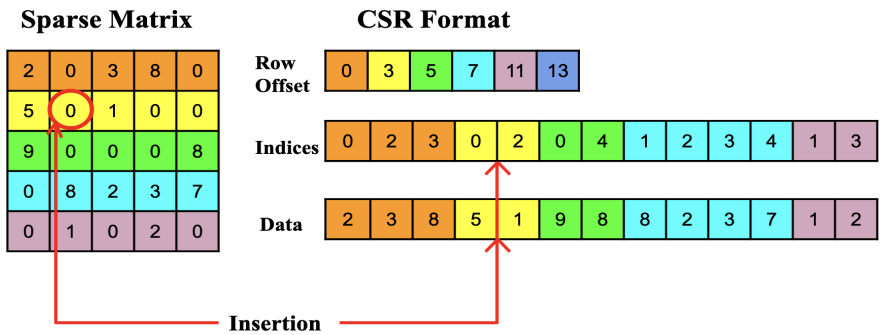


Figure 2.2 A sparse matrix in CSR format

Figure 2.2 shows an example of a sparse matrix in CSR format and its update scenario. In order to insert a new nonzero entry into the red circle, we need to place it at the red arrow-ed position of the *Indices* and *Data* array. It causes all the elements on the right to be shifted, which takes $O(N)$ time in the worst case. Like this example, formats for fast LA operations suffer from high inefficiency on the insertion or deletion of nonzero entries because existing nonzero entries must be shifted along a continuous memory area.

2.2.2 Formats for Fast Updates

Several formats have been proposed to avoid the expensive shifts of nonzero entries on matrix updates. These formats manage update-friendly data structures where each entry is stored in a possibly non-continuous memory area,

thus giving up spatial locality between them. By doing this, matrix updates can be done by just (de)allocating space for the nonzero entries and re-organizing the internal data structures when necessary. Popular examples include Scipy's Dictionary of Keys (DOK), which is a hash table consisting of nonzero entries; List of Lists (LIL), just like the Adjacency Lists for graph representation; and Armadillo's Red-Black Tree (RBT), which is a red-black tree storing nonzero entries.

Although these formats perform better on matrix updates than the CSR format, they cannot process LA operations quickly. Since these formats store nonzero entries in a possibly non-continuous memory area, pointer tracing becomes necessary to access adjacent nonzero data. Thus, we can't expect good cache behavior when we traverse through the nonzero entries of sparse matrices in this category. Since adjacent data are more likely to be referenced together during most of the LA operations, this lack of locality is a severe drawback.

So when we need to perform LA operations on this type of matrices, it is highly inefficient to do it on these formats directly. Instead, we need to convert them into CSR format and apply the operations on these CSR matrices. Though a certain amount of conversion overhead exists, it is generally negligible considering the high inefficiency of update-friendly formats on LA operations. Several open-source array processing libraries, such as Scipy and Armadillo, adopt this strategy. In Scipy [12], matrix updates can be done efficiently using DOK or LIL formats. However, when LA operations are required, Scipy internally converts them into CSR format and performs the operations. Armadillo [13] provides a sparse matrix in a hybrid format. Depending on workload, it can

automatically switch between CSC and RBT formats, where CSC specializes in fast LA operations, and RBT is good at matrix updates.

Chapter 3

Proposed Sparse Matrix Formats

This paper aims to suggest a new update-friendly sparse matrix format (Chapter 2.2.2) that can efficiently handle matrix updates and CSR conversion. In this chapter, we introduce some previously proposed formats of this category and analyze their pros and cons in detail. Table 3.1 below summarizes symbols frequently used for the rest of this paper. And Table 3.2 on the next page summarizes the performance comparison between the update-friendly sparse matrix formats addressed in this paper.

Table 3.1 Notation

Symbol	Description
m	Number of rows of the matrix
n	Number of columns of the matrix
N	Number of nonzero entries of the matrix
R	Number of nonzero entries in the row where the target entry exists
U	Number of all matrix updates
C	Number of CSR conversions
T	Period for CSR conversions

Table 3.2 Performance comparison between update-friendly formats

Format	Search	Insertion	Modification	Deletion	CSR Conversion	Space
DOK	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(NlogN)$	$O(N)$
LIL	$O(R)$	$O(R)$	$O(R)$	$O(R)$	$O(N)$	$O(m+N)$
RBT	$O(logN)$	$O(logN)$	$O(logN)$	$O(logN)$	$O(N)$	$O(N)$
RBF	$O(1)$	$O(logR)$	$O(1)$	$O(logR)$	$O(N)$	$O(m+N)$

3.1 Dictionary of Keys (DOK)

Scipy's Dictionary of Keys format (DOK), as its name suggests, manages a hash table whose key is a (*row*, *col*)-coordinate of the nonzero element and value is its nonzero value. It takes $O(N)$ space consisting of N hash table entries along with some extra unfilled buckets. Since the search for an element in a hash table takes $O(1)$ time on average, insertion, modification, and deletion of a nonzero entry in the DOK sparse matrix also take $O(1)$ time on average. Thus, DOK is one of the fastest formats for matrix update.

However, elements in a hash table are generally not ordered. So DOK ill-performs when we need to do ordered iteration on its elements, as in CSR conversion. CSR manages nonzero data ordered by its column indices. Thus, in order to convert DOK into CSR, we need to sort its elements by their keys which takes $O(NlogN)$ time on average. As Table 3.2 shows, this is the worst CSR conversion performance among the update-friendly formats. Chapter 5 shows that this sorting overhead can become a massive bottleneck in dynamic workloads, making it quite inefficient compared to other formats despite its optimal performance on matrix updates.

3.2 List of Lists (LIL)

Scipy's List of Lists format (LIL) resembles Adjacency List (AL) for graph representation. It manages an array whose index represents a row of the matrix, and its element is a linked list of column indices and a data value of the nonzero entry in the corresponding row. For m-by-n sparse matrix of N number of nonzero elements, LIL takes $O(m + N)$ space where m is generally negligible compared to N.

Let R denote the number of nonzero elements of the row where the target entry exists. Then, searching for the entry takes $O(R)$ time in the worst case because we need to do a linear search on the corresponding linked list. Thus, insertion, modification, and deletion of a nonzero entry in the LIL sparse matrix also take $O(R)$ time in the worst case. Though elements in each linked list are ordered by its column indices, it does not help to reduce the time because we cannot perform binary searches on a linked list. Chapter 5 shows that LIL seriously ill-performs on matrix update compared to other formats because of this linear time complexity.

CSR conversion can be done by just iterating over each linked list which takes $O(N)$ time. Since elements in each linked list are ordered already, we do not need to sort them during the conversion. However, in spite of this optimal performance on CSR conversion, LIL was found to be the most inefficient format among those introduced in this chapter because of its outstanding effect of linear time complexity on matrix update.

3.3 Red-Black Tree (RBT)

Armadillo's Red-Black Tree format (RBT) manages a red-black tree whose key is a (*row*, *col*) coordinate of the nonzero element, and the value is its nonzero value. Just like DOK, it also takes $O(N)$ space consisting of N red-black tree nodes. Since it is height-balanced, searching for the entry takes $O(logN)$ time in the worst case. Moreover, tree re-balancing cost after the insertion or deletion of its node also takes $O(logN)$ time in the worst case. Thus, all kinds of matrix update in RBT sparse matrix take $O(logN)$ time in the worst case. As Table 3.2 shows, this is slower than DOK but generally faster than LIL.

RBT compares its keys in lexicographical order. Thus, CSR conversion can be done by performing in-order traversal on the red-black tree, which takes $O(N)$ time. Due to its optimal CSR conversion performance and its moderate speed on matrix updates, RBT was found to be more efficient for dynamic workloads than the others introduced in this chapter.

Chapter 4

RBForest

In this chapter, we propose the RBForest format (RBF), which performs efficiently on dynamic workloads better than the previously proposed formats in Chapter 3. We introduce our core design choices that make the RBF a better choice on dynamic workloads than the others. After that, detailed explanations of the RBF's matrix update and CSR conversion procedures will follow. From these, we emphasize the RBF's advantages over the other formats in Chapter 3.

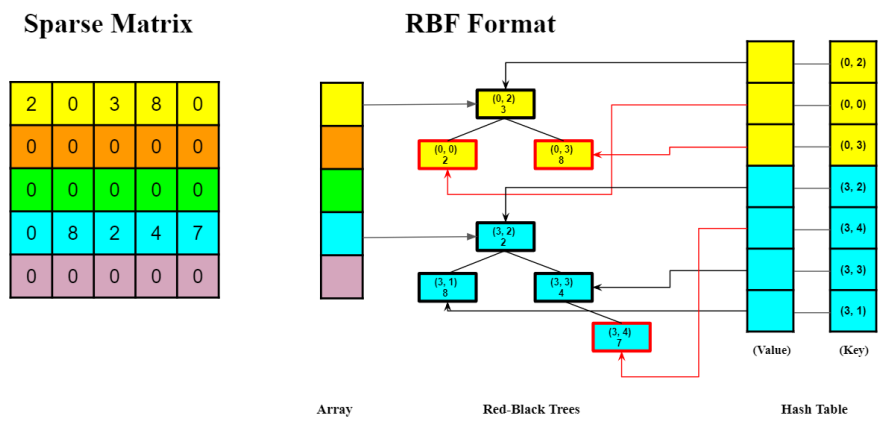


Figure 4.1 A sparse matrix in RBF format

4.1 Design Choices

RBF consists of the following data structures:

- *Array* of pointers to each red-black tree.
- *Red-Black Tree* of nonzero entries for the corresponding row.
- *Hash Table* whose key is the (*row*, *col*)-coordinate, and value is the pointer to the red-black tree node that corresponds.

Figure 4.1 shows the architecture of the RBF format in detail. It has an array whose index represents a row of the matrix, and its element is a pointer to the root node of the red-black tree where nonzero entries of the corresponding row are stored. It is a variation of the LIL format, where linked lists are replaced with red-black trees. It also has a hash table storing nonzero entries of the matrix. Its key is a coordinate of each nonzero entry, and the value points to the corresponding red-black tree node. It enables a faster search for the sparse matrix's nonzero entries.

So it is a combination of three primitive data structures used for the other formats in Chapter 3. It takes $O(m + N)$ space in total, where m is generally negligible compared to N. The main design choices of the RBF are as follows.

4.1.1 Individual Red-Black Trees for each row

In RBF, each row has its own red-black tree containing nonzero entries that correspond. The name RBForest stems from this idea. We chose this design to reduce the height of each red-black tree lower than that of Armadillo's RBT format, where all nonzero entries are stored in a single red-black tree. The height of a red-black tree is logarithmically proportional to the number of nodes it contains. So storing each row's nonzero entries in separate red-black

trees helps to reduce each tree's height from $O(logN)$ to $O(logR)$. Reduced tree height then reduces the time to re-balance each red-black tree after its insertion or deletion. Thus, by adopting this design choice, we were able to improve the matrix update performances while maintaining space occupied by red-black trees same as that of Armadillo's RBT.

4.1.2 Hash Table Connected to Red-Black Trees

RBF manages a hash table to enable fast search on its nonzero entries. Each hash table entry itself is a pointer to the red-black tree node corresponding to its (*row*, *col*)-coordinate. So retrieving a nonzero value on a specific position of the sparse matrix can be done by just a single lookup on the hash table, which takes $O(1)$ time. Since any matrix update must be done after the initial search for the target nonzero entry (Figure 2.1), our design choice of using a hash table significantly improves their performances compared to the other formats in Chapter 3. Furthermore, though hash table entries are generally unordered, CSR conversion in RBF can be done by in-order traversing through each red-black tree which takes $O(N)$ time in total. So RBF can exploit the advantages of the hash table while still maintaining its ordered iteration cost as $O(N)$.

4.2 Operations

Here, we illustrate RBF's matrix update and CSR conversion procedures in detail. We also analyze their time complexities and compare them with the others in Table 3.2, thus emphasizing the strength of RBF over the others.

4.2.1 Search

As we mentioned in Figure 2.1, all types of matrix updates are done after the search for the target entry, and in RBF sparse matrix, this is done by its hash table (Chapter 4.1.2). If a nonzero entry exists at the target coordinate, the hash table entry is returned, which is a pointer to the corresponding red-black tree node. If it does not exist, $NULL$ is returned. Thus, for all types of matrix updates in RBF, the initial search for the entry takes $O(1)$ time on average.

4.2.2 Insertion

If there was previously no nonzero entry on the target coordinate, it must be newly inserted. For the RBF sparse matrix, insertions are done in the following steps.

Step 1 (Insertion at the Red-Black Tree) First, we allocate a new red-black tree node and insert it at the corresponding row's red-black tree. Since the height of the target red-black tree is $O(logR)$, the insertion and re-balancing all take $O(logR)$ time in the worst case.

Step 2 (Insertion at the Hash Table) Then, we insert the (row, col)-coordinate as a key into the hash table and set its value to the pointer to the red-black tree node just inserted. Obviously, this takes $O(1)$ time on average.

Thus, the overall performance of the RBF insertion is $O(logR)$, which is faster than that of LIL and RBT formats. And it is obviously slower than the DOK format, which takes $O(1)$ time on the same operation. However, DOK takes $O(NlogN)$ time in CSR conversion, which is significantly worse than the RBF. And, since R is generally much smaller than N, the performance loss the RBF suffers on the insertion and deletion ($O(1) \rightarrow O(logR)$) is negligible when

the performance gain it achieves on the CSR conversion ($O(NlogN) \rightarrow O(N)$) is considered.

4.2.3 Modification

If a nonzero entry already exists at the target coordinate and the requested value is nonzero, we modify the old value to the new one. So the modification can be done by modifying the value of the entry we found at the *Search* stage. Thus, its time complexity is equal to that of the *Search* operation. Since RBF search takes $O(1)$ time, RBF modification also takes $O(1)$ time on average, which is faster than LIL and RBT and equal to that of DOK.

4.2.4 Deletion

If a nonzero entry exists at the target coordinate but the requested value is zero, it must be deleted from the matrix. For the RBF sparse matrix, deletions are done in the following steps.

Step 1 (Deletion from the Red-Black Tree) First, we need to delete the red-black tree node that corresponds to the target entry and re-balance the tree if required. However, unlike the *Insertion*, we do not need to move downward through the tree to reach the target node because we can immediately access to it at the *Search* stage using its hash table. So we can simply deallocate the target node and re-balance the tree, which takes $O(logR)$ time in the worst case.

Step 2 (Deletion from the Hash Table) Then, we delete the hash table entry we found at the *Search* stage. It takes $O(1)$ time on average

Thus, the overall performance of the RBF deletion is $O(logR)$, which is faster than that of LIL and RBT formats. Again, it is slower than the DOK,

but it is trivial considering the RBF's CSR conversion performance, which is much faster than the DOK.

4.2.5 CSR Conversion

Conversion to the CSR format can be done in the following steps:

Step 1 (Allocating Memory for the CSR Matrix) First, we need to allocate memory where the converted CSR matrix will be written. Since we know the number of nonzero entries of the matrix a priori, we can immediately allocate enough memory for the CSR matrix. Thus, this step takes $O(1)$ time.

Step 2 (Ordered Iteration over each of the Red-Black Trees) Nonzero data in the CSR sparse matrix is sorted by its column indices. So we need to follow this ordering when we write nonzero entries of the matrix into the pre-allocated memory for the CSR. Thus, the ordered iteration is the main operation we need to perform to do the CSR conversion. In RBF, there is an array containing pointers to red-black trees ordered by their row indices, and each red-black tree compares its keys by column indices. So the ordered iteration can be done by in-order traversing each red-black tree which takes $O(N)$ time in total.

If we convert the RBF matrix in Figure 4.1 into the CSR, for example, we first need to allocate enough memory for the CSR representation of it. Then we write nonzero entries in the yellow-colored red-black tree, representing the first row, into this pre-allocated memory while in-order traversing over it. Do the same for the blue-colored red-black tree also, which represents the fourth row. Since we iterate each node only once for all red-black trees, the CSR conversion can be done in $O(N)$ time.

So the overall performance of the CSR conversion is $O(N)$, which is the same as the LIL and RBT. And it is much faster than the DOK, which takes $O(NlogN)$ time because of the sorting cost. Chapter 5 shows that this sorting overhead is so significant that it over-compensates the performance loss the RBF suffers at the *Insertion* and *Deletion* operations.

Chapter 5

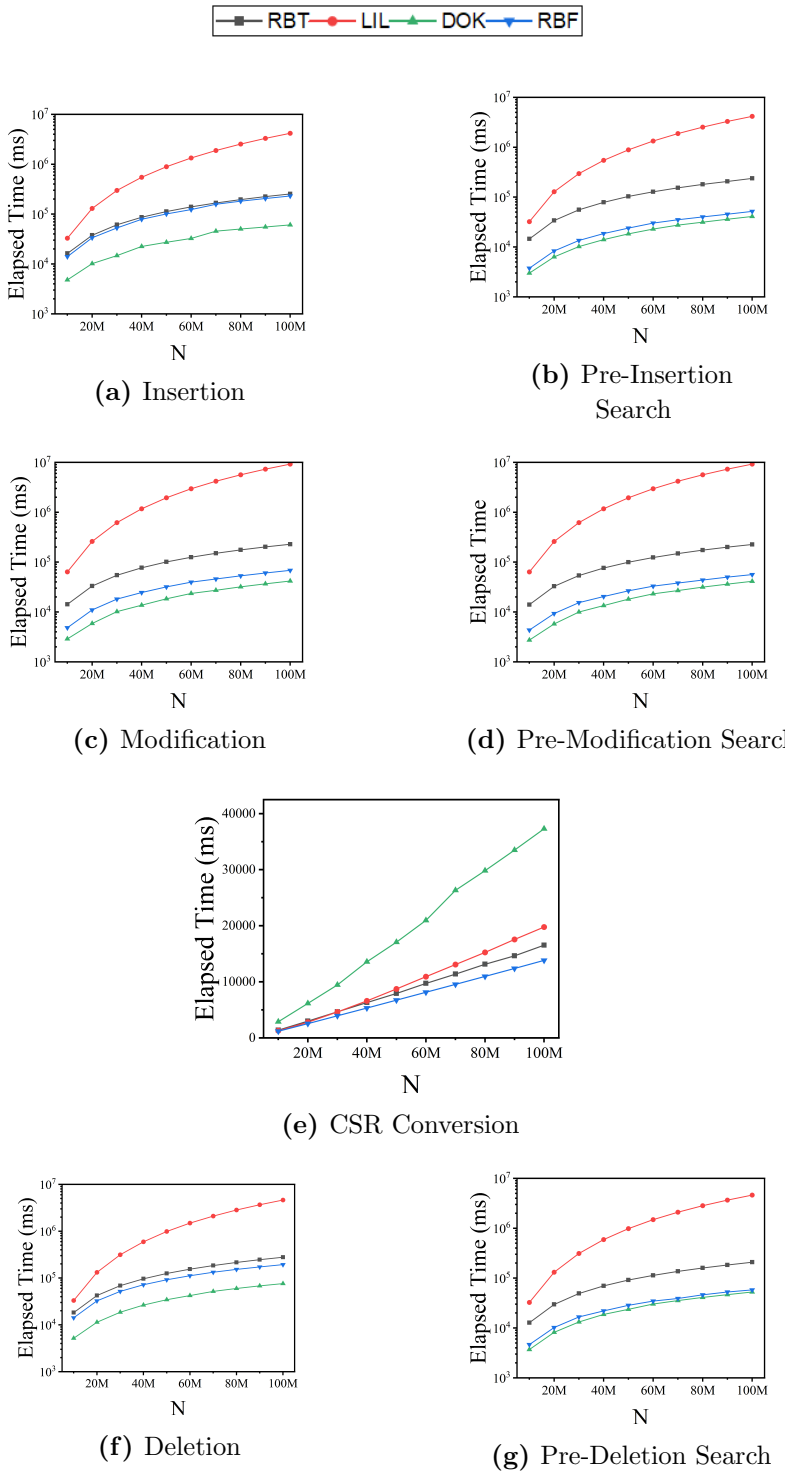
Evaluation

In this chapter, we analyze the performance of the RBF format by comparing it with the other update-friendly formats introduced in Chapter 3. All the experiments were conducted on a single machine with a Ryzen 7 1700 8-Core Processor, 64 GB RAM, and a 256 GB SSD, running Ubuntu 18.04 LTS. We implemented all formats for comparison in C++17 with g++ 7.5.0. All experimental results except for Chapter 5.3 are the average of 5 intermediate values of 7 executions.

5.1 Performance on Individual Operations

Here, we used synthetically generated $100K * 100K$ sparse matrices, varying the number of nonzero entries from $10M$ to $100M$. All nonzero entries were generated at a random position in the matrix. Using these synthetic sparse matrices, we conducted the experiments as follows:

- **Insert** all the nonzero entries into the target format.
- **Modify** the value of all nonzero entries just inserted.
- **Convert** the target format into the CSR.
- **Delete** all the nonzero entries from the target format.



(a) Insertion

(b) Pre-Insertion Search

(c) Modification

(d) Pre-Modification Search

(e) CSR Conversion

(f) Deletion

(g) Pre-Deletion Search

Figure 5.1 Performance comparisons on individual operations

Then, we plot the performances for each matrix update operation and the CSR conversion individually. For the matrix updates, we also plot the initial search performance separately to visualize the impact of search cost on them. Based on these plots, we analytically compare the performances of the sparse matrix formats for individual operations.

5.1.1 Insertion

For the insertion performance, DOK is the fastest, followed by the RBF, RBT, and LIL in this order (Figure 5.1a). This is consistent with the theoretical analysis in Table 3.2. However, despite the significant gap in time complexity between the RBT and RBF, their actual performance does not vary much. This is because the RBF inserts a nonzero entry into the two data structures, the hash table, and the red-black tree, whereas the RBT inserts it only at the red-black tree. Thus the RBF has to do much more work after the initial search for the entry than the RBT.

Deletion also suffers from this issue, but as shown in Figure 5.1f, RBF performs better on deletion than insertion. This is because, for the insertion, RBF must reach the leaf node of the corresponding red-black tree starting from its root to insert the entry. On the other hand, this does not happen at deletion because RBF can access the target red-black tree node immediately using the hash table. Thus, the RBF's insertion involves more pointer-tracing than deletion, which leads to its worse performance.

However, the RBF still performs better than the RBT because of the reduced tree height we introduced in Chapter 4.1.1. Thus, the overheads we mentioned are tolerable considering the overall performance on insertion and

the potential benefit it can bring on the other operations, such as modification and deletion.

5.1.2 Modification

The modification performance shows the same trend as in the insertion. Since the modification procedure is not that complicated, its performance (Figure 5.1c) is similar to the performance of the initial search conducted right before it (Figure 5.1d). Since RBF's search operation is done using its hash table, its update performance is similar to that of the DOK and much faster than the others. The effect of our design choice of using the hash table (Chapter 4.1.2) becomes maximized in the modification operation.

5.1.3 CSR Conversion

For the CSR conversion, DOK is the worst, while the others show little difference (Figure 5.1e). This is because the DOK must sort its unordered entries for the conversion while the others do not have to. The $O(NlogN)$ time complexity for sorting makes the DOK the least suitable format for the fast CSR conversion. Though the DOK is the winner on all matrix update operations, it is not the best for the dynamic workload (Chapter 5.2) because of this huge drawback on CSR conversion. All other formats do CSR conversion in $O(N)$ time, so there is no significant performance gap between them.

5.1.4 Deletion

The deletion performance also shows the same trend as in the insertion and modification. As mentioned earlier, the performance gap between the RBT and RBF becomes larger than for the insertion. Thus the RBF's deletion also

benefits from our design choice of using the hash table (Chapter 4.1.2). The RBF's performance gap between the deletion (Figure 5.1f) and the initial search right before it (Figure 5.1g) is still large because, just like the insertion, it must delete the entry from the two data structures. Despite this overhead of managing multiple data structures, RBF performs better than RBT because of its reduced tree height (Chapter 4.1.1).

5.2 Performance on Dynamic Workload

In this chapter, we conducted experiments on the dynamic workload, which is the main concern of our research. While the sparse matrix in an update-friendly format accepts any updates at random, CSR conversion is conducted periodically. We conducted these experiments using both synthetic and real-world sparse matrices.

Again, we use a $100K * 100K$ sparse matrix, fixing the total number of matrix updates with $100M$ at random positions. Among these updates, the number of insertions, modifications, and deletions is $50M$, $25M$, and $25M$, respectively. Then, we vary the number of periodic CSR conversions from 10 to 100 and plot the performance. The CSR conversion was conducted after the fixed amount of matrix updates. For example, if the number of conversions is 10, then the conversion happens at every $10M$ update.

5.2.1 Test with Synthetic Matrix

Figure 5.2 shows the result of this experiment. The RBF beats the LIL and RBT regardless of the number of conversions. It is because the RBF's matrix update is more efficient than the LIL and RBT. Thus, the RBF is more suitable for the dynamic workload than the LIL and RBT. Especially, the LIL is found

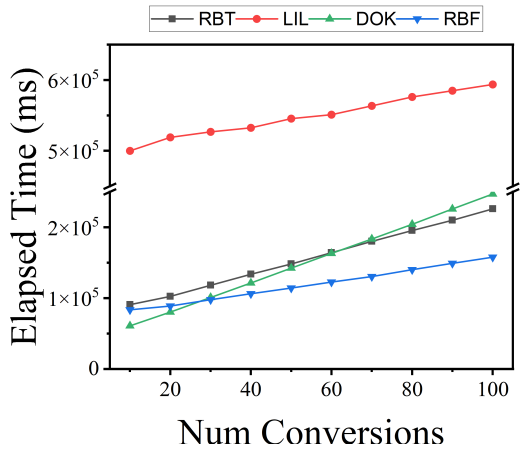


Figure 5.2 Performance comparison on dynamic workload with a synthetic matrix

to be very inefficient for the dynamic workload due to its extremely poor matrix update performance compared to the others.

Compared with the DOK, however, the RBF is slower than the DOK, where the number of conversions is smaller than 30. It is because the DOK's matrix update is faster than the RBF. The RBF's faster CSR conversion performance does not stand out when the number of conversions is insufficient. If the number of conversions becomes larger than 30, the RBF starts to beat the DOK, and the performance gap between them becomes larger as the number of conversions increases. It shows that the DOK's CSR conversion is quite inefficient, which further shows that the DOK is not the best choice for dynamic workloads. Though there exists a point where the performances of the DOK and RBF are reversed, Chapter 5.2.2 shows that, in reality, the CSR conversion is more likely to happen frequently, making the RBF the best choice for the dynamic workloads.

5.2.2 Test with Real-World Matrices

We also conducted experiments using real-world sparse matrices. They were selected from various contexts where sparse matrices are widely used. Each matrix contains a certain amount of matrix updates with a timestamp indicating the time when the update happened. For each of the matrices, we performed matrix updates in increasing order of timestamps. And based on the timestamps, we selected the appropriate period in which CSR conversions are conducted. The periods were selected considering the time span of the given matrix. For example, if the matrix spans more than a year, we set the period to a month. And if the matrix spans within a week, we set the period to an hour. So we reproduced the real-time analysis scenario on sparse matrices with legitimate CSR conversion frequencies considering the time spans of each matrix

We used 7 real-world sparse matrices for the experiment. Following is a brief description of the matrices that we used for the experiment.

- **StackOverflow** [18, 19] is an interaction network from the online site "*Stack Overflow*". It spans from August 2008 to March 2016.
- **ESWIKI** [18, 20] is a bipartite edit network from the Spanish Wikipedia. It spans from May 2001 to July 2017.
- **HS** [21] is a contact and friendship network between students in a high school in Marseilles, France in December 2013.
- **PS** [22, 23] is a contact network between the children and teachers in a primary school in January 2014.
- **LastFm** [18, 24] is a bipartite network that represents the user-song listening relationship of the online music service site "*Last.fm*". It spans from February 2005 to September 2013.

- **LKML** [18, 25] is a communication network of the Linux kernel mailing list. It spans from January 2006 to January 2014.
- **Delicious** [18, 26] is a network that represents the user-tag relationship of the social bookmarking web service "*Delicious*". It spans from September 2003 to December 2007.

Table 5.1 Statistics of the real-world sparse matrices

Matrix	m	n	N	U	C	T
Stack Overflow	$2.6M$	$2.6M$	$36.2M$	$63.5M$	92	month
ESWIKI	$935.5K$	$5.9M$	$30.6M$	$67.4M$	194	month
HS	$1.9K$	$1.9K$	$5.8K$	$188.5K$	41	hour
PS	$0.5K$	$0.5K$	$8.3K$	$125.8K$	20	hour
LastFm	$0.9K$	$1.1M$	$4.4M$	$19.2M$	55	month
LKML	$63.4K$	$63.4K$	$243.0K$	$1.1M$	97	month
Delicious	$833.1K$	$4.5M$	$82.0M$	$301.2M$	52	month

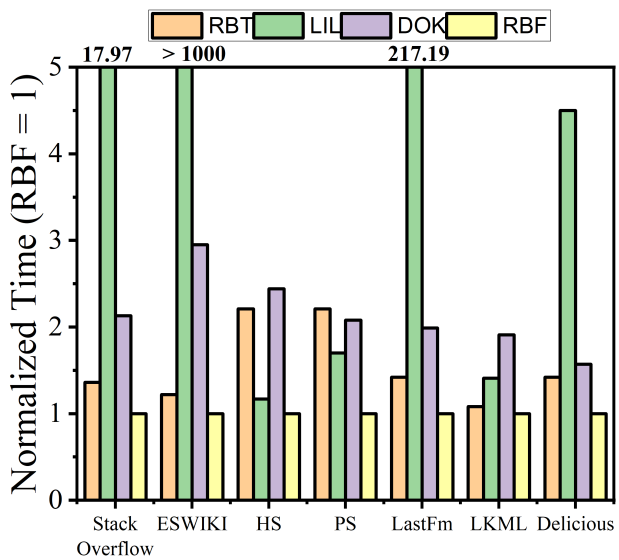


Figure 5.3 Performance comparison on dynamic workload with real-world matrices

Figure 5.3 shows the normalized running time of each format with respect to the actual running time of the RBF on each sparse matrix. The RBF shows the best performance for all real-world sparse matrices compared to the other formats. The RBF's superiority over the LIL and RBT is obvious because the RBF performs matrix updates much more efficiently than the LIL and RBT. The RBF also performs better than the DOK for all matrices, although the RBF's matrix update performance is poor compared to the DOK. It is because the RBF's efficiency on the CSR conversion over-compensates its drawbacks on matrix updates over the DOK. Thus the RBF is proven to be the best choice for real-world dynamic workloads.

5.3 Memory Consumption

We also measured the memory consumption of each sparse matrix format. We used synthetically generated $100K * 100K$ sparse matrices, varying the number of nonzero entries from $10M$ to $100M$. Then, we measured the amount of memory occupied by each format.

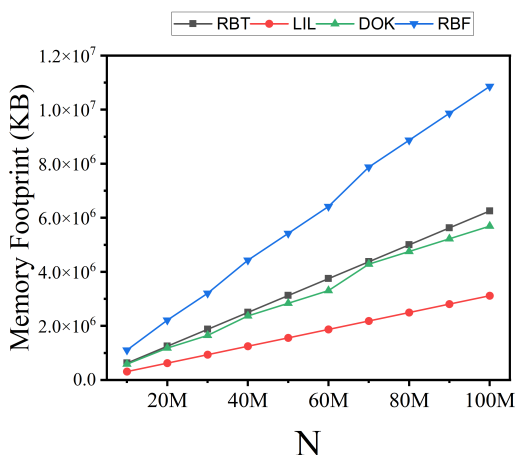


Figure 5.4 Memory Consumption of each sparse matrix format

Figure 5.4 shows the results. The LIL was found to be the most memory-efficient format. It is because the LIL consists of singly-linked lists which do not need much memory per element. The RBT and DOK occupy similar amounts of memory. The RBT and DOK consume more memory than the LIL because they need more memory per element.

The RBF was found to be the least memory-efficient format. It is because the RBF manages both the red-black tree and hash table, resulting in its total memory occupation becoming approximately the sum of the RBT and DOK. Thus, a tradeoff exists between dynamic workload performance and memory consumption.

5.4 Additional Remarks on Experimental Results

Except for the memory test, the performance of LIL was found to be extremely poor compared to the others. It is because the LIL's matrix update is more inefficient than the others. As described in Figure 2.1, all types of matrix updates are preceded by an initial search for the target nonzero entry. Thus, if the initial search itself is slow, the whole matrix update procedure also becomes slow. And Figure 5.1 shows that the LIL performs poorly on this initial search for the target entry. Therefore, the LIL's poor performance on matrix updates is due to its poor performance on the search operation.

LIL's search operation is slow because we have to perform a linear search on the list corresponding to the row where the target entry exists. It takes $O(R)$ time in the worst case, and as Table 3.2 shows, this is the worst search performance among the four formats. Performing linear search on every matrix update is a serious bottleneck, making the LIL the least desirable format for dynamic workloads.

However, LIL is still popular and widely used to represent graphs which can also be represented as sparse matrices. It is because the LIL format is good at retrieving all neighbors of a certain vertex. Since the LIL associates each vertex in the graph with the collection of its neighboring vertices [27], retrieving all neighbors of the certain vertex can be efficiently done by just iterating over the single list that corresponds. Neighbor retrieval is widely used in several graph operations such as PageRank [28], Single-Source Shortest Path [29], and many others.

So, if neighbor retrievals are the major workload, LIL's performance can be much better than that of the DOK and RBT and similar to that of the RBF. In LIL and RBF, the elements are aggregated in terms of their first coordinate, making neighbor retrieval much easier and more efficient. However, in DOK and RBT, neighbor retrieval may require the search on entire data structures, which is much more inefficient compared to the LIL and RBF.

Thus, it may seem that the LIL's advantages are underrated in this paper. However, this paper supposes that all analytic operations are done in a format for fast LA operations such as CSR. The update-friendly formats are supposed to handle only matrix updates. Thus, LIL's efficiency on some operations other than the matrix updates is out of the interest of this paper. If neighbor retrievals are requested, the matrix is first converted to the CSR and then the retrievals are conducted on the CSR matrix.

Furthermore, this paper proposes a new format for general sparse matrices, not for particular types of matrices such as graphs. Thus, we must consider more general workloads, including LA operations on matrices, and cannot focus on specific types of operations, such as neighbor retrievals. Considering its usage as a general-purpose sparse matrix, LIL is the worst-performing format for

sparse matrices, and there was no underestimation regarding the advantages of the LIL format in this paper. However, since the LIL was found to be much worse than the others in our experimental results, we leave the LIL's possible advantages and example workloads where it can show optimal performances as an additional remark here.

Chapter 6

Conclusion and Future Works

This paper proposes a new sparse matrix format called *RBForest* (RBF), that efficiently performs on the dynamic workload than the previously proposed formats. RBF manages one red-black tree per row to reduce the average height of the trees. It reduces the tree re-balancing cost after inserting or deleting the nonzero entries. Also, the RBF uses a hash table to enable immediate access to the target nonzero entry. It reduces the cost of searching for the target nonzero entry to be inserted, modified, or deleted. Though the RBF is not the first-prize winner for all individual operations, it was proven to be the best choice for practical dynamic workloads where the matrix updates and the CSR conversions occur altogether.

One of the possible areas of future work can be trying to reduce the memory consumption of the RBF format. Since the RBF consists of many primitive data structures, inefficient memory usage is inevitable. Therefore, one can try to reduce this high memory overhead while maintaining or even improving its performance on dynamic workloads.

Bibliography

[1] H. Kwak, C. Lee, H. Park, and S. Moon, "What is twitter, a social network or a news media?," in *Proceedings of the 19th international conference on World wide web*, pp. 591–600, 2010.

[2] Y. Zhou, L. Liu, S. Seshadri, and L. Chiu, "Analyzing enterprise storage workloads with graph modeling and clustering," *IEEE Journal on Selected Areas in Communications*, vol. 34, no. 3, pp. 551–574, 2016.

[3] R. Albert, H. Jeong, and A.-L. Barabási, "Diameter of the world-wide web," *nature*, vol. 401, no. 6749, pp. 130–131, 1999.

[4] H. Jeong, B. Tombor, R. Albert, Z. N. Oltvai, and A.-L. Barabási, "The large-scale organization of metabolic networks," *Nature*, vol. 407, no. 6804, pp. 651–654, 2000.

[5] D. Dai, R. B. Ross, P. Carns, D. Kimpe, and Y. Chen, "Using property graphs for rich metadata management in hpc systems," in *2014 9th parallel data storage workshop*, pp. 7–12, IEEE, 2014.

[6] H. Qin, R.-H. Li, G. Wang, L. Qin, Y. Cheng, and Y. Yuan, "Mining periodic cliques in temporal networks," in *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pp. 1130–1141, IEEE, 2019.

[7] H. Wu, Y. Huang, J. Cheng, J. Li, and Y. Ke, "Reachability and time-based path queries in temporal graphs," in *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*, pp. 145–156, IEEE, 2016.

[8] Wikipedia contributors, "Sparse matrix — Wikipedia, the free encyclopedia." `https://en.wikipedia.org/w/index.php?title=Sparse_matrix&oldid=1116925321`, 2022.

[9] M. Silva, "Sparse matrix storage revisited," in *Proceedings of the 2nd conference on Computing frontiers*, pp. 230–235, 2005.

[10] Z. Bai, J. Demmel, J. Dongarra, A. Ruhe, and H. van der Vorst, *Templates for the solution of algebraic eigenvalue problems: a practical guide*. SIAM, 2000.

[11] F. Smailbegovic, G. N. Gaydadjiev, and S. Vassiliadis, "Sparse matrix storage format," in *Proceedings of the 16th Annual Workshop on Circuits, Systems and Signal Processing*, pp. 445–448, 2005.

[12] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, *et al.*, "Scipy 1.0: fundamental algorithms for scientific computing in python," *Nature methods*, vol. 17, no. 3, pp. 261–272, 2020.

[13] C. Sanderson and R. Curtin, "A user-friendly hybrid sparse matrix class in c++," in *International Congress on Mathematical Software*, pp. 422–430, Springer, 2018.

[14] S. Vassiliadis, S. Cotofana, and P. Stathis, "Block based compression storage expected performance," in *High Performance Computing Systems and Applications*, pp. 389–406, Springer, 2002.

[15] Y. Saad, "Krylov subspace methods on supercomputers," *SIAM Journal on Scientific and Statistical Computing*, vol. 10, no. 6, pp. 1200–1232, 1989.

[16] A. Ekambaram and E. Montagne, "An alternative compressed storage format for sparse matrices," in *International Symposium on Computer and Information Sciences*, pp. 196–203, Springer, 2003.

[17] R. Shahnaz, A. Usman, and I. R. Chughtai, "Review of storage techniques for sparse matrices," in *2005 Pakistan Section Multitopic Conference*, pp. 1–7, IEEE, 2005.

[18] J. Kunegis, "Konect: the koblenz network collection," in *Proceedings of the 22nd international conference on world wide web*, pp. 1343–1350, 2013.

[19] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection." `http://snap.stanford.edu/data`, June 2014.

[20] "Wikimedia downloads," 2010.

[21] R. Mastrandrea, J. Fournet, and A. Barrat, "Contact patterns in a high school: a comparison between data collected using wearable sensors, contact diaries and friendship surveys," *PloS one*, vol. 10, no. 9, p. e0136497, 2015.

[22] V. Gemmetto, A. Barrat, and C. Cattuto, "Mitigation of infectious disease at school: targeted class closure vs school closure," *BMC infectious diseases*, vol. 14, no. 1, pp. 1–10, 2014.

[23] J. Stehlé, N. Voirin, A. Barrat, C. Cattuto, L. Isella, J.-F. Pinton, M. Quaggiotto, W. Van den Broeck, C. Régis, B. Lina, *et al.*, "High-resolution measurements of face-to-face contact patterns in a primary school," *PloS one*, vol. 6, no. 8, p. e23176, 2011.

[24] Òscar Celma, *Music Recommendation and Discovery.* Springer Berlin, Heidelberg, 2010.

[25] D. Homscheid, J. Kunegis, and M. Schaarschmidt, "Private-collective innovation and open source software: Longitudinal insights from linux kernel development," in *Conference on e-Business, e-Services and e-Society*, pp. 299–313, Springer, 2015.

[26] R. Wetzker, C. Zimmermann, and C. Bauckhage, "Analyzing social bookmarking systems: A del. icio. us cookbook," in *Proceedings of the ECAI 2008 Mining Social Data Workshop*, pp. 26–30, 2008.

[27] Wikipedia contributors, "Adjacency list — Wikipedia, the free encyclopedia," 2022.

[28] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web.," tech. rep., Stanford InfoLab, 1999.

[29] E. W. Dijkstra, "A note on two problems in connexion with graphs," in *Edsger Wybe Dijkstra: His Life, Work, and Legacy*, pp. 287–290, 2022.

국문초록

Sparse matrix는 0이 아닌 원소만을 그 좌표 정보와 함께 저장한다. 따라서 모든 원소를 1차원 배열의 연속적인 메모리 공간상에 저장하는 dense matrix와는 달리, sparse matrix는 그 내부의 자료구조를 다양하게 선택할 수 있다. 따라서 sparse matrix의 관리를 위한 서로 다른 다양한 종류의 인-메모리 포맷이 존재할 수 있다.

현재까지 여러 종류의 sparse matrix 포맷이 제안되었고, 이들은 서로 다른 장단점을 가지고 있다. 선형대수 연산을 빠르게 지원할 수 있는 포맷은 행렬 갱신이 느리다는 단점이 있고, 그 반대로도 마찬가지이다. 그러나 sparse matrix의 많은 용례는 매우 동적이다. 다시 말해, sparse matrix의 행렬 갱신이 매우 빈번하게 발생한다. 따라서 sparse matrix는 선형대수 연산을 포함한 분석 질의를 주기적으로 처리하는 와중에 계속되는 행렬 갱신도 빠르게 처리할 수 있어야 한다.

그러므로 sparse matrix를 관리하는 주요 전략 중 하나는, 행렬 갱신이 계속되는 동안에는 이를 갱신이 빠른 포맷으로 관리하다가, 분석질의가 요청되었을 때 선형대수 연산처리가 빠른 포맷으로 변환하는 방식이다. 이 전략을 최대로 활용하기 위해, 본 논문은 *RBForest*라는 새로운 sparse matrix 포맷을 제안한다. *RBForest*는 각 행마다 하나의 레드-블랙 트리를 관리하여 0이 아닌 원소의 삽입, 삭제 이후 발생하는 자료구조 재조정 비용을 줄인다. 또한 해시 테이블을 관리하여 행렬이 관리하는 0이 아닌 원소에 즉시 접근할 수 있도록 하고, 이는 행렬 갱신에서의 초기 조회의 비용을 줄인다. 본 논문에서의 실험에 따르면 *RBForest*는 비슷한 전략으로 관리되는 sparse matrix 포맷과 비교했을 때, 현실의 동적 워크로드에서 더 효율적으로 동작한다는 것이 증명되었다.

주요어: 희소 행렬, 행렬 갱신, CSR 포맷

학　번: 2021-20552