University of Alberta

FAR-Tree: A Spatial Index For Solid State Drives

by

Feng Jiang

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

Abstract

Solid state drives (SSDs) are becoming more common with their main advantage of faster reads compared to hard disk drives (HDDs). However, writes are relatively slower, indeed asymmetric with respect to reads, unlike in HDDs where read and write are comparable. Current indexing structures were designed for HDDs and aim at reducing the number of reads at query time. In SSDs, index writes may impact the overall query performance in the presence of updates. Thus, we focus on minimizing the number of writes during index update, considering the R-tree in particular, given that it is an ubiquitous and update-expensive indexing structure. We propose the FAR-tree (for Flash Aware R-tree) which aims at avoiding node splits by building a chain of nodes on leaf nodes. Our experiments using real and synthetic datasets show that the FAR-tree can yield a more update-efficient index at the cost of some overhead at query time.

Acknowledgements

I would like to express my sincere gratitude to my supervisor Dr. Mario A. Nascimento for his guidance and encouragement during my study and research as a master student.

I would also like to thank the department and the university who provide the great environment for me to study and enjoy my student life.

Furthermore, I like to thank my friends for their help and the great time we had together.

Last but not the least, I would like to thank my parents for their love and support all my life.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Many books come with an index section in their end to help readers navigate the book. The book index is a collection of headings and associated page number(s) where information about the heading can be found. The headings are typically titles, concepts, or subjects in the book and are usually categorized or alphabetically sorted. Readers can quickly locate their interested topics in a given book directed by the index.

Generally, an index is a data structure built on data to improve the performance of search queries. It provides efficient data retrieval of both sequential access and random lookups. An index is key to a database management system (DBMS) to enhance performance. For example the B-tree index, which is commonly used in relational databases. A B-Tree search starts from the root node and chooses the subtree by comparing the search value and the separation value in the node. A query on a number of n objects takes $O(\log n)$ with a B-Tree index, compared with a sequential scan without using an index that takes $O(n)$. A B-Tree example is showed in Figure 1.1. If we look for 17 from the numbers, without an index, we need to traverse each of the numbers until we meet 17; So in the given example we need 16 reads. With the index, we start from the root and descend to the right subtree since 17 is larger than 10, then we descend to the right subtree of 14 since 17 is between 14 and 18, and finally we find 17 on the leaf node; during this process, we find 17 after only 4 reads.

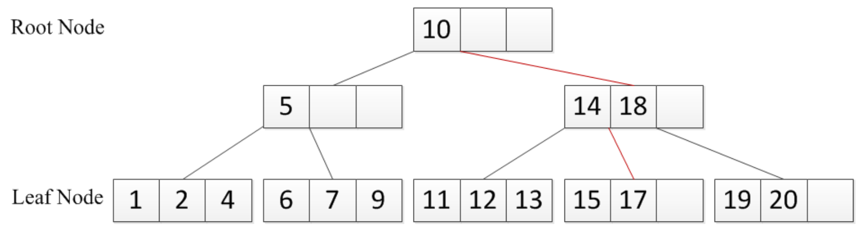


Figure 1.1: B-Tree index for data: 2, 14, 10, 5, 1, 4, 6, 11, 13, 19, 7, 20, 18, 9, 12, 17, 15

A spatial index is used to quickly locate and retrieve objects in space based on their locations. Over the past decades, intense researches have been done on spatial indexing to pursue a more efficient maintenance and usage of the spatial data. The most popular index structures is the R-Tree [4] and the many variants based on it [2] [6]. It is the most commonly used spatial structure and has been implemented in database products such as Oracle, MySQL, and PostgreSQL [3].

The R-Tree and its variants are mostly designed for the *hard-disk drives* (HDDs). Nowadays, solid-state drives (SSDs) are a new storage media for both personal computers and commercial servers, with its featured advantages of faster performance and lower power consumption than the HDDs [15]. Generally, there are three factors that affect the performance of a magnetic disk: the seek time, the rotational latency, and the data transfer time. For the SSD, however, besides a superior transfer rate [7], it does not need time to seek for tracks and there is no mechanical latency. A key component of an SSD is the flash memory. Most SSDs are built with NAND flash memory. Flash memory is an electronic memory and is composed of flash cells, which catches or insulates electrons to charge the voltage on the flash cells, and the flash memory then stores the information using arrays containing cells' voltage. Unlike HDDs, which require magnetic heads to access data on the rotating platters, the flash memory does not have to pay for the expensive disk seeks when doing random read access. Reading on SSDs thus costs less than on HDDs. Furthermore, without using the rotating platters, the SSDs do not consume power as much as the HDDs.

The advantageous features make SSDs an appropriate alternative to HDDs for storing and indexing data. At the same time, it is realistic nowadays considering

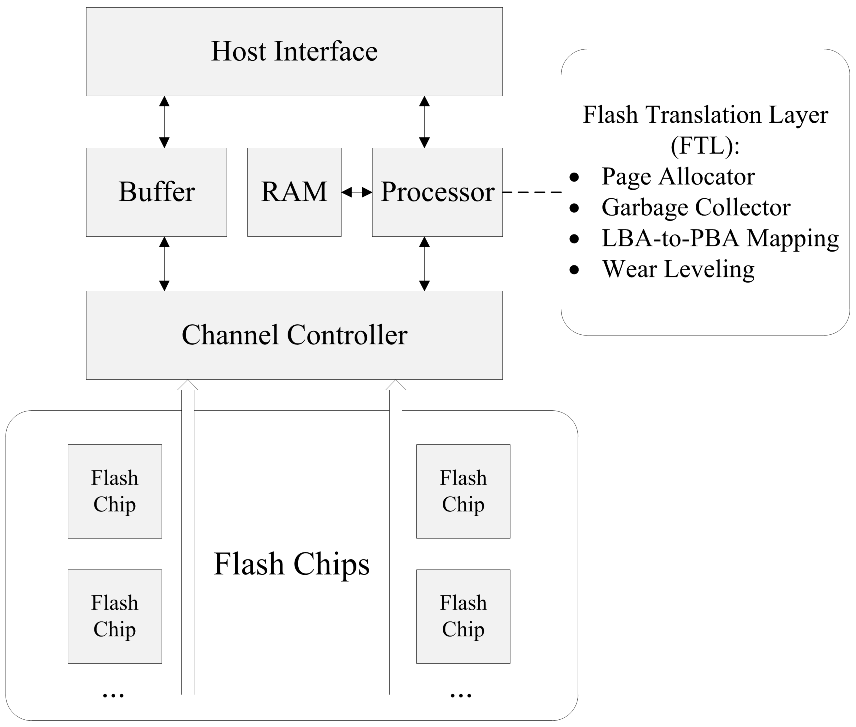
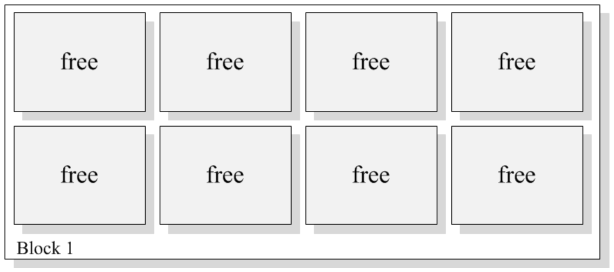


Figure 1.2: A general SSD architecture

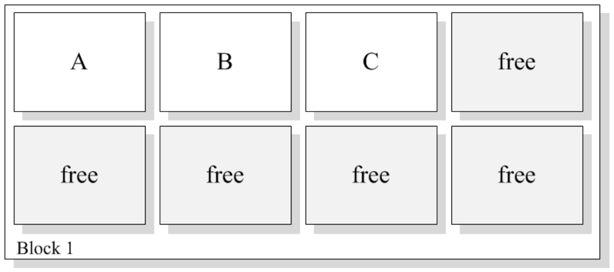
the affordable price and the increasingly available storage space of the SSDs [3]. Also, sharing the same I/O interface with traditional HDDs, the SDDs are easy to be employed on existing systems from the hardware aspect. A general architecture of an SSD is displayed in Figure 1.2. The Flash Translation Layer (FTL) maintains the disk's operation. The LBA-to-PBA mapper maps every logical page to a physical one; the page allocator find free pages to write data; the garbage collector does the flash disk garbage collection and the wear leveling is to swap blocks for longer disk life span, which will be explained in the next three paragraphs.

However, due to different mechanisms of the hardware implementation, the flash-based SSDs differ from the conventional magnetic-based HDDs on the I/O performance. A major difference is the SSDs' *I/O asymmetry*. By contrast, HDDs spend the same amount of time for reading and writing data, as both operations are performed by first moving the heads to the target location on the disk and then respectively read or write data. The access time for SSDs is typically faster than HDDs when reading data from the disks, while writing could be slower.

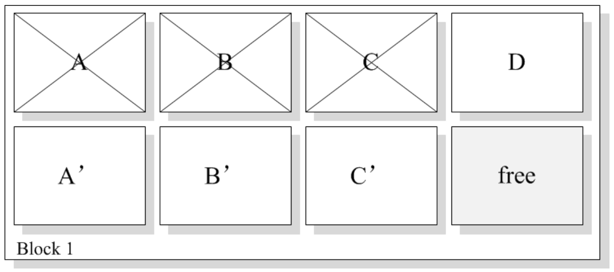
In an SSD, the *flash chips* are divided into *blocks*, and each block contains *pages*



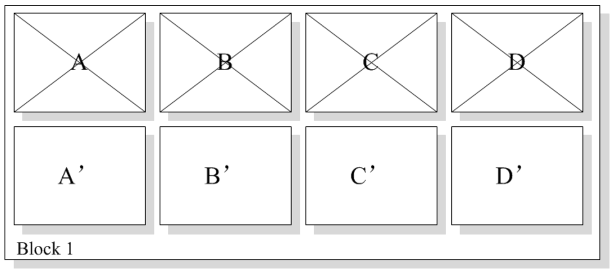
(a) A fresh block containing 8 pages.



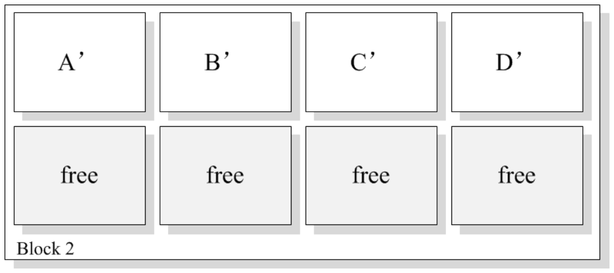
(b) Writing A, B, C to Block1.



(c) Writing D to Block1, and update A, B, C to A’, B’, C’.



(d) Updating D to D’.



(e) Valid contents are moved to Block2 after garbage collecting.

Figure 1.3: Examples of SSDs’ operations

where the data is stored. Typically, a page is 4 - 8 KB in size and a a block consists of 32 - 128 pages. An example of a flash block is showed in Figure 1.3a where there are 8 blank pages contained in Block1. Operations on the disks includes *read*, *program*, and *erase*: *read* is to read a page from the disk, *program* means first-time writing on a fresh page, and *erase* is to clear up all existing contents within a block. So the SSD reads and programs in pages while erases in blocks. The SSD page cannot be over-written. Thus, to update a page within a block, the flash memory will mark the old page as invalid and then look for a new fresh page to program the updated value. For example, we are to write data A, B, and C to the flash block in Figure 1.3a. Since all pages are fresh in this block, the data will be programmed to the first three pages, as showed in Figure 1.3b. Next we are to write D and update A, B, and C to A', B', and C'. On the disk, D is programmed to the next free page. However, for the reason that the flash pages cannot be over-written, to update A, B, and C, pages containing the old values will be marked as invalid, and the updated value A', B' and C' will be programmed to the next three free pages, as resulted in Figure 1.3c. Similarly when we update the data D, we need to first invalidate the page of D and program the new value D' on the last free page in this Block1, as showed in Figure 1.3d. Moreover, the SSDs will reclaim a block when the number of invalid pages is large enough; this reclaiming process is called *garbage collection*. During the garbage collection, the remaining valid pages of that block, if any, will be moved to a new block and the old block will be erased as a whole to be back to fresh. A whole block has to be erased even though it is partially dirty. Assume a block will be garbage collected when half of its pages are invalid in Figure 1.3d. To garbage collect Block1, the SSD will find a new fresh block named Block2 and move A', B', C', and D' to the Block2; afterwards, everything on Block1 will be erased and Block 1 is back to fresh as was in Figure 1.3a. Another strategy called *wear leveling* swaps intensely-used blocks with rarely-used ones. A flash block would wear out if programmed and erased a large number of times. By swapping blocks, as the writes are distributed such that every block can have their maximum utilization, the disk's life is extended. However, both the garbage collection and the wear leveling cause extra writes on disks. The amount of actual physical writes on flash disks is larger than the amount of logical commands from disk manager. This phenomenon is called *write amplification*.

In summary, the I/O asymmetry and the garbage collection lead to the fact that writing, especially randomly, on SSDs is slow and expensive. The SSDs' I/O asymmetry is due to the page-based write and block-based erase of flash chips. Garbage collection and wear leveling cause write amplification, which brings more disk writes.

In this thesis, we focus on a particular index, the R-Tree, and how it can benefit from the SSDs architecture.

An R-tree is a dynamically constructed and maintained spatial index structure. Operations on the R-Tree requires large amounts of reads and writes. Like other tree structures, R-Tree is made up of nodes, which are composed of the Minimum Bounding Rectangles (MBRs) of spatial objects as the entries of a node. We will explain the R-Tree in detail in Section 2.2. For an R-Tree, usually a node is stored as a single page. According to the SSDs' characteristics, each time we update a node or randomly insert entries to a node, all the information on this node has to be firstly invalidated and the updated node will be programmed on a fresh page afterwards. Thus, continuously updating nodes will increase the number of invalid pages within blocks. The garbage collection process will reclaim a block when invalid pages within it exceed some threshold. Furthermore, an insertion of a data item into the R-Tree can trigger node splits. One node split needs at least three writes: two on the split nodes and one on their parent node. One node split could further cascade onto upper levels and result in multiple node splits until reach the root; this will introduce even more writes on the disk.

When implementing the R-Tree spatial index on the SSDs, it becomes important to consider the index updates, as that may impact the overall query response time in the presence of updates. Therefore, to leverage the faster reads over the slow random writes of SSDs, modifications are needed on the original indexing scheme, with the goal being to reduce the expensive writes.

1.2 Contributions

In this thesis, we adapted the existing R-Tree index for SSDs. We investigated and designed an indexing approach, which was initially proposed in [11] for an efficient spatial indexing on the SSDs. It reduces the random writes of index updates so as to avoid the drawbacks of the SSDs. Although it introduces some extra reads on queries, it works well for a comparatively update-intensive spatial system, considering the cheap read costs of the SSDs. Also, it does not need special care for buffer management.

The contributions of this thesis are summarized as follows:

1. We analyzed the impacts on the R-Tree brought by the flash-based SSDs. Applying the characteristics of the flash storage, we conducted a detailed analysis for major operations on the R-Tree of their costs of disk I/Os, compared with those on the magnetic storage.

2. We designed the FAR-Tree, a flash-aware spatial index grounded on the R-Tree. Based on an imbalanced indexing idea in [11], we investigated to design and develop the FAR-Tree indexing approach, which reduces a large number of random writes by avoiding the R-Tree node splits.

3. We conducted a performance analysis of the FAR-Tree, for both insertions and queries. The analysis provides a theoretical comparison between the R-Tree and the FAR-Tree on their I/O costs.

4. We presented quantitative performance analysis of the FAR-Tree. We applied the index on real-world spatial data, as well as synthetic data with various attributes such as distributions and size of objects. We also tested the impact of bulk-loading the base R-Tree.

5. We measured the disk access time on a real SSD, besides simulations measured in logical I/Os.

1.3 Organization

The rest of this thesis is organized as follows. Chapter 2 reviews previous solutions of indexing for both regular and spatial databases on the SSDs. It also explains the R-Tree principles and performances when applied on the SSDs instead of the HDDs. Chapter 3 describes the details of our FAR-Tree structures and algorithms. It gives a cost analysis for the FAR-Tree, with a comparison to the R-Tree. Chapter 4 presents the experimental evaluation and analysis result of our approach, on both real-world data and synthetic data, with both logical I/Os and real disk time measurements. Chapter 5 summarizes the paper and discusses the future work.

Chapter 2

Related Work

This chapter gives an overview of existing studies leveraging the SSD characteristics on database storage and indexing, some of which are applicable or designed for the spatial data. Then it introduces the R-Tree spatial index and how the R-Tree operates on SSDs.

2.1 Databases on Flash Storage

Various approaches has been proposed to adapt database systems on SSDs. The approaches can be generally categorized into:

- system architecture design
- buffer management
- indexing algorithm adaption

A review on data management over flash memory can be found in [10].

2.1.1 System Architecture Design

SSDs enjoy fast reads but have slow writes. In a database system, SSDs provide benefits for rapid data queries but still not enough to directly replace the HDDs, due to its slow updates. In this case, the systems are designed to have both the SSDs and the HDDs at the same time and synthesize their respective strengths. So for the system design approaches, it is not only focused on the database application, but more generally on the system storage level.

HDDs and SSDs at the same storage level

One alternative of the design is to have HDDs and SSDs at the same level in the storage hierarchy [9]. SSDs and HDDs are both used as persistent storage. Whether to place incoming data on the SSD or on the HDD is determined by the workload on the data. The system will identify the workload to place a page at the right disk; read-intensive data will be placed on the SSD and write-intensive data will be placed on the HDD. Also, the system will monitor reads and writes operations on the data pages. If the workload changes, pages might migrate between disks, for example, a page will be moved from the SSD to the HDD if its workload changes from read-intensive to write-intensive.

HDDs and SSDs at different storage level

Another alternative is to put the disks at different storage hierarchy level: one works as persistent primary storage and another works as cache.

Using the SSD as primary storage, the authors of [20] add the HDD as a "write cache" to improve write throughput and reduce writes on the SSD. Incoming write operations will be firstly logged onto the HDD cache. When the cache is full, the logged changes will be merged and written to the SSD. The number of writes on the SSD is reduced, since multiple writes could be merged into one. Also, HDDs' latency and rotational delay are diminished since the log is a sequential write to the disk.

On the other hand, SSDs are adopted as read cache when the HDD is acting as the primary storage, considering the fast read of SSDs [13]. With efficient reading and larger capacity than the RAM, the SSD acts as an adequate tier between the buffer and the disk for reading. In this approach, part of the SSD cache also acts in logging writes, taking advantage that flash memory is good for large sequential writes.

2.1.2 Buffer Management Approaches

When only using SSDs as the storage, instead of hybridizing with magnetic disks, one key point is to manage the buffer to reduce the writes on the SSD. Buffer management approaches are proposed either with smart eviction algorithms or to aggregate random writes in the buffer and flush them in bulk, both are to avoid expensive writes as much as possible.

Eviction algorithms

The idea of smart eviction algorithms is to choose proper victim pages/blocks to minimize the disk writes in total. A clean first LRU (CFLRU) [17] approach divides the buffer pool into working regions and clean-first region; the clean pages in the clean-first region are always evicted before the dirty pages. Compared with the normal LRU, pages could stay in the buffer longer, and this reduces writes from evicting dirty pages. A cost-based buffer approach [9] estimates the costs on eviction of each pages. Similar to the CFLRU, evicting dirty pages leads to write costs while evicting clean pages does not. This approach has a cost region in the buffer pool where the LRUs are queued by their eviction costs; units in the cost region with least eviction costs will be the victim.

Aggregating random writes

On the other hand, aggregating random writes alleviates the SSD write costs, considering the comparably faster sequential writes to the random writes on SSDs. The block padding LRU (BPLRU) approach [8] buffers only write operations; reads are directly requested from the flash disk at the fast read rate. Also, blocks for the LRU or the MRU are queued, moved, and evicted in the buffer, where the blocks are in equal size with the blocks of the SSD. If a data block is not full on eviction, it will read the missing part to fill the block so that the eviction writing will be sequential in a whole block; updates will be performed by the disk erase unit on the flash disk, which diminishes the overhead from random page updates. The append and pack approach [21] create a layer between the DBMS buffer manager and the physical disk. On eviction, it groups dirty pages in multiples of the SSD's erase block size, so that it writes dirty pages to the disk in blocks sequentially.

2.1.3 Indexing Algorithm Adaption

Besides the system and the memory level adaption, research has been done on directly designing or modifying the database indexes. As database indexes need to be kept updated upon changes of the underlying data, but maintaining the index causes lots of expensive writes, when implemented on SSDs, the index should be designed to reduce the number of writes caused by index updates. Most index designs are combined with correlated flash-aware buffer management.

Logging

The methods in [25] and [19] allocate a space in memory for each tree node to store a list of update operations on that node. There is a mapping table between the tree nodes and the updates. Query on a node will be done by combining the original node information with the update list. Based on this, the approach in [25] will compact the update list when there are too many items in a list. The design in [19] keeps log of all operations in case of system crash. The update lists are adopted in the FlashDB design as well, but not for all nodes [16]. A node in the FlashDB is determined either in disk mode or in log mode, by the workload on the node. A disk node is stored the same as a normal index node; a log node is composed of a linked list of update logs for that node. Update-intensive nodes will be in log mode so that the writes are amortized. Furthermore, the FlashDB is a self-tuning index that each node can switch its mode to cater for dynamic workload.

Transforming random writes to sequential

Other approaches reduce write costs on flash disks by transforming random writes into sequential writes, an idea similar to the buffer management approaches. The LA-Tree divides a tree into subtrees and attaches buffers to each of the subtrees [1]. The buffer batches the updates to be performed on nodes of that subtree all at once. This reduces the number of writes on the disk and transforms the random writes into sequential ones. The FD-Tree has multiple levels in the index and when upper levels get full to merge with lower levels, writes are transformed from random to sequential [14].

In summary, the system architecture and buffer management approaches can be generally applied with all tree index structures in databases, or even for the file systems. Various indexes can observe similar performance. For the indexing algorithm adaption approaches, they are more specific and mostly focused on one certain index structure. For example, among the designs mentioned above, the approaches in [16] and [14] are designed for B^+-Tree, the approach in [25] is for R-Tree, while the approaches in [1] and [19] can be generally applied with multiple index structures.

2.2 The R-Tree

The R-Tree is the most commonly used spatial structure. Similar to the B-Tree, the R-Tree is a balanced tree structure, but designed for spatial objects [4].

2.2.1 The R-Tree Index Structure

In the R-Tree, spatial objects are represented by their minimal bounding rectangles (MBRs), with a unique identifier for each. With a structure similar to the B-Tree, the R-Tree has both internal nodes and leaf nodes. An R-Tree example is shown in Figure 2.1. Each node consists of entries to spatial objects or pointers to nodes in the lower level; the leaf node entries store pointers to the real object data; the index node entries store the address of nodes in its lower level. Entries in the leaf node are stored in the form of:

(Object's Identifier, Object's MBR),

and entries in the internal node are stored as:

(Child Node's Identifier, MBR of Child Node).

Nodes are stored corresponding to disk pages. Each node consists of the node head and the node content. The node head stores the profile information of the current node, such as the node ID and the node level; the node content refers to entries stored in the node. The MBR of a node is the combining MBR of all entries stored in it. Moreover, same as the B-Tree, each node can contain a maximum number of entries, which is called the node capacity, and a minimum number of

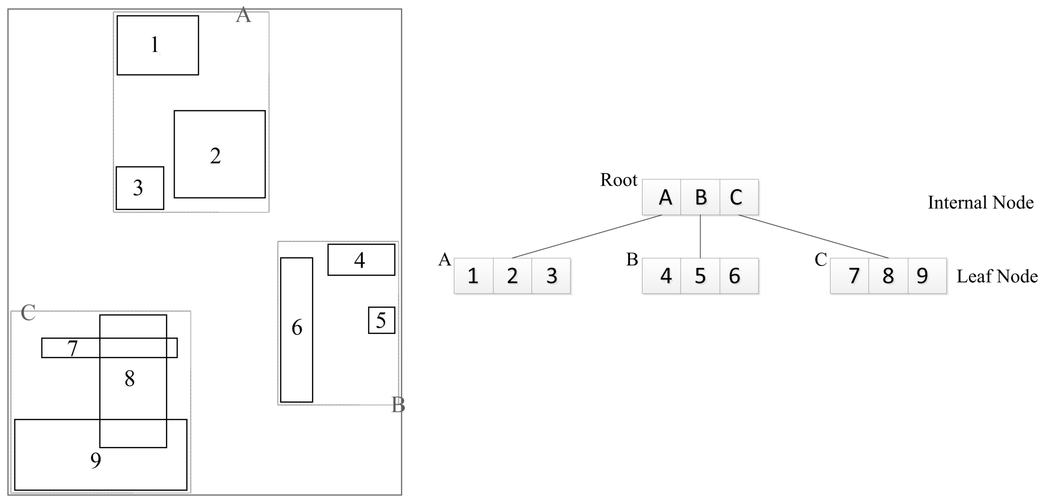


Figure 2.1: A spatial example and the R-Tree built on the distribution.

entries is enforced as well.

2.2.2 The R-Tree Operations

A spatial distribution example and its R-Tree index are shown in Figure 2.1. There are 9 objects in the space area, represented by their MBRs. When constructing an R-Tree, neighboring objects are grouped in one node upon their insertions; the R-Tree insertion will be explained later. An R-Tree index constructed for the spatial distribution in Figure 2.1 is shown on its right. We assume the node capacity is 3 in this example. In the R-Tree, there are three leaf nodes; the internal node, which is also the root node in this case, will hold the three leaf nodes as its entries.

As a balanced tree index structure, the R-Tree has basic operations of searching, insertion, deletion and updating. With the example in Figure 2.1, we now explain the R-Tree operations in detail.

1. The **insertion** will firstly find a leaf node to place the new object, with a *chooseLeaf* function. The *chooseLeaf* will recursively traverse the tree from the root node and descend by choosing the subtree with least MBR enlargement to include the new object. After inserting the object, the tree will adjust itself, that changes on the leaf node will be propagated upward to the root. If the leaf node is full before the insertion, the leaf node will split into two

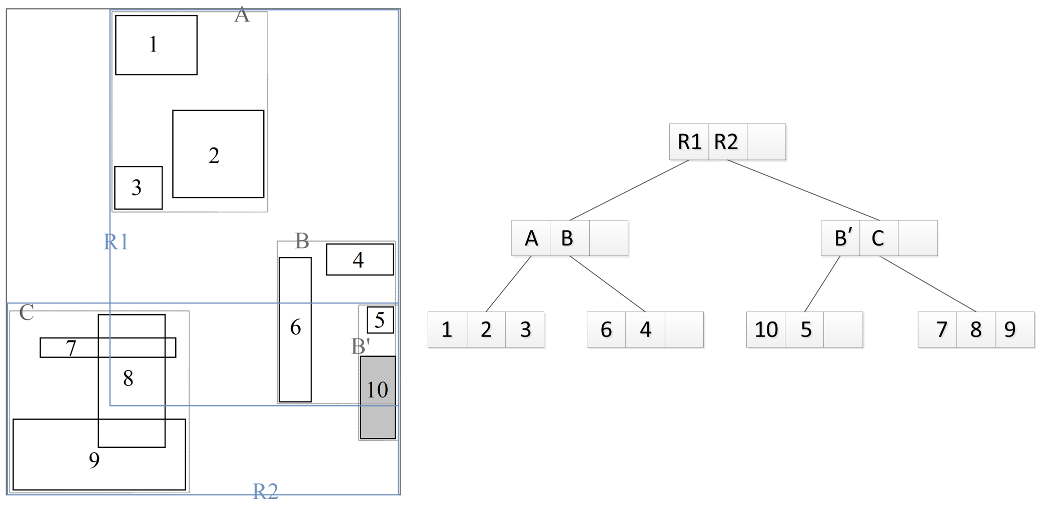


Figure 2.2: Insert object 10 to the spatial example in 2.1.

nodes and a new entry will be added into their parent node as a pointer to the new split node; again, the parent node will split if it does not have space for the entry to be added. Exhaustive algorithm, quadratic-cost algorithm and linear-cost algorithm are provided for the node splitting. In this thesis, we adopt the quadratic split.

2. The **searching** algorithm of R-Tree is similar to other tree structures. The R-Tree search starts from the root node. For an intersection query, it examines the query rectangle against each of the entry's MBR in the node to see if they intersect, so as to determine which subtree(s) to search inside. Recursively the searching descends until the leaf node(s). When reach a leaf node, the query rectangle will be examined against each of the entries in the leaf node for intersection, and the query results will be returned.

3. The **deletion** is similar to the insertion. It firstly finds the leaf node where the targeting node to be deleted is located. After deleting the node, the tree will be condensed that changes on the leaf node propagated upwards. The **updating** is operated by deleting the old value and re-inserting the updated one.

Fore more details about the R-Tree operations, refer to [4].

For the example in Figure 2.1, we are to insert an object 10 into this spatial area, as shown in Figure 2.2 on the left. The *choosesubtree* function determines to place the entry 10 in node B because B's MBR enlarges the least to include the new entry, compared with A or C. However, since the leaf node B is full, B needs to split before inserting the entry 10. Splitting the node B further leads to the split of the root node (which is the parent of node B) since the root node is full as well. In this case, the tree is formed into the structure on the right of Figure 2.2 according to the R-Tree insertion algorithm. The leaf node B is split into two leaf nodes B and B', each contains two object entries; the root node is also split into two internal nodes with each contains two leaf nodes; a new root node is created and the tree height is increased by one.

For a tree structure, the insertion and searching are more important. We will not focus on the deletion and the updating in this thesis.

2.2.3 Bulk-loading R-Tree

To provide efficient query on the R-Tree, the key point is to minimize the empty space covered by nodes' MBRs and the nodes' overlap. The more concise the MBRs, the fewer subtrees a query will need to process so that fewer reads are needed.

When constructing an R-Tree, the original idea was to insert one object at a time. The insertion algorithm chooses the leaf node with least MBR enlargement to put the new object; the choice is locally optimal. Thus, constructing an R-Tree by inserting, there are disadvantages of sub-optimal space utilization, which may reduce the query performance. The R-Tree structure is thus highly influenced by the order that the entries are inserted.

Solutions have been proposed to optimize the structure of an R-Tree when it is built: one is to perform changes to the insertion algorithm and another is bulk-loading the R-Tree. R*-Tree is one of the examples modifying the insertion algorithm. It forces a re-insertion on node overflow [2]. When inserting an entry to a node which is full, its entries are firstly removed from the node and then re-inserted into the tree. Entries may be placed at a more proper place than their original location so that entries in a node are better clustered with less node MBR coverage. The general idea of bulk-loading is to firstly sort the spatial objects based on their locations and then pack and distribute them in nodes according to the sorting results to build the tree. The objective is also to have objects within one node closer. Various packing algorithms have been proposed for bulk-loading the R-Tree, such as the Nearest-X [18], the Hilbert Sort [5], and the STR [12]. However, none of the algorithms is optimal for all kinds of datasets. Thus, in this thesis, for fairness of comparison, we do not focus on the R-Tree bulk loading; although we will mention the bulk-loading about its influence on our FAR-Tree approach.

2.2.4 Disk Write of R-Tree Insertion

As a dynamic balanced tree, the R-Tree updates itself upon insertions, which leads to write operations on the disk. When implementing the R-Tree on SSDs, we care much about the write costs brought by the R-Tree's construction and maintenance.

According to the insertion algorithm, inserting an entry to a full leaf node leads to the node to split. Whenever a node is split, at least three nodes are updated: the split node, the new created node, and their parent node. Moreover, if the split is propagated, more node updates will be needed. As the example in Figure 2.1 and Figure 2.2, the leaf node splits and the split is propagated upwards, the MBRs are adjusted and the updates ascend to upper levels. Inserting object 10 costs five node writes: two for splitting B into B and B', two for splitting the root node, and one for creating a new root node. In the case when the tree is large enough with a lot more levels, the cost of disk writes brought by tree adjustment during insertion will be considerable. We will have a detailed disk access analysis for the insertion when comparing with the FAR-Tree in Chapter 3. A model for the R-Tree query performance on disk access is presented in [22], which is an analysis generally applicable for SSDs as well.

Chapter 3

The FAR-Tree

The FAR-Tree is a flash-aware variant of the R-Tree spatial index. It is designed to reduce the expensive disk writes on SSDs caused by the R-Tree index updates. In this chapter, we will introduce the idea and explain the detailed implementation of the FAR-Tree approach.

3.1 The FAR-Tree Design

The original R-Tree index is a balanced tree structure with dynamic node insertions, deletions, and updates. As discussed in Chapter 1, when implementing the R-Tree on SSDs, problems come up due to the read-write asymmetry of the flash storage. Inspired by [11], the idea of the FAR-Tree is to improve the disk access efficiency for the R-Tree index maintenance. When inserting data into an R-Tree, instead of splitting nodes to accommodate new inserting data, the FAR-Tree appends data to the overflowing leaf nodes. This results in an imbalanced tree structure but reduces the number of node writes.

3.1.1 The FAR-Tree Structure With A Running Example

A FAR-Tree example is shown in Figure 3.1. There are three types of nodes in the FAR-Tree: the internal node, the leaf node, and the chain node. The internal node and the leaf node are as defined in the R-Tree; the chain node are nodes attached on a leaf node. Entries in the leaf node and the chain node represent real spatial objects and entries in the internal node point to the corresponding children nodes. Besides,

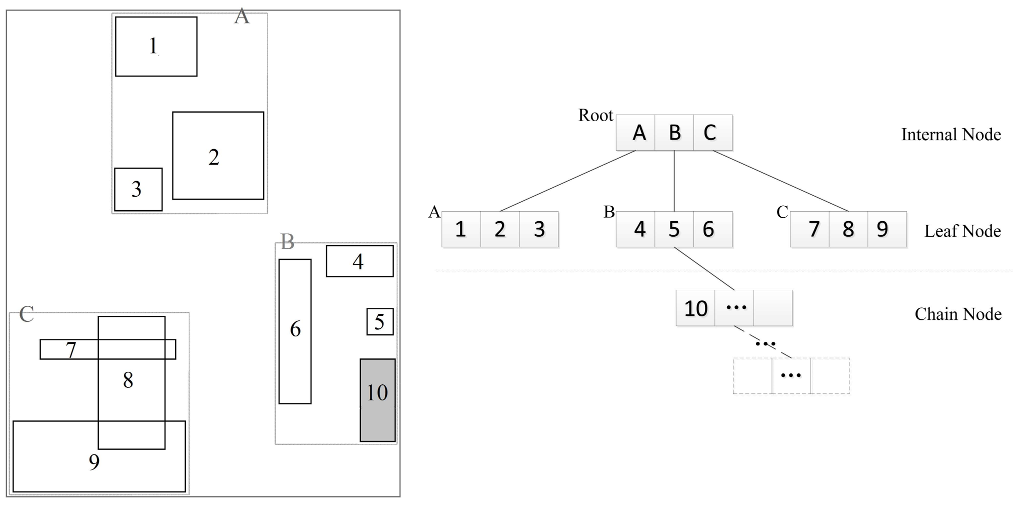


Figure 3.1: Insert object 10 with the FAR-Tree insertion algorithm to the spatial example in Figure 2.1.

every leaf node has two pointers to the first and the last chain node attached on it, and every chain node has a pointer to its following chain node on the same chain; for all of them, the pointer is set to NULL if no node is attached.

The basic idea of the FAR-Tree for saving disk writes is that there is no node split when inserting new objects. The FAR-Tree insertion will firstly look for a proper leaf node to place the new object. If the leaf node has space to include the object, an entry representing the object will be inserted into the leaf node. However, if the chosen leaf node is full, unlike the R-Tree where we split the leaf node, we will create a chain node to place the new entry and append the chain node to that leaf node. All subsequent objects to be inserted on this leaf node's branch will be put into the chain node; furthermore, when the chain node is full, a new chain node will be created and appended on the previous one in the same way.

This appending insertion method leads to an imbalanced structure of the index. Accordingly, an alternative query approach is enforced on this imbalanced index structure. A query in the FAR-Tree starts from the root node and descends to subtrees. When the query arrives at a leaf node, entries in the leaf node will be traversed sequentially. Furthermore, if there are any chain node(s) appended on this leaf node, the query will keep traversing entries in the chain node(s) sequentially. Query results are returned while traversing entries of both the leaf node and the chain node(s).

The detailed algorithms for the FAR-Tree operations will be explained in Section 3.1.3, while here we give an overview of the the FAR-Tree operations with an example of the same scenario we used for explaining the R-Tree in Chapter 2. In the example, we are to add a new object 10 to the spatial area shown in Figure 2.1. With the R-Tree insertion, the node B will split as it has no room for the new entry. With the FAR-Tree method, instead, a chain node is created and appended to node B, in which we place the entry 10. The FAR-Tree index is constructed in Figure 3.1, as a comparison with the R-Tree in Figure 2.2. At the same time, the leaf node's MBR will be updated to include all chain nodes appended. The internal nodes' MBR on that path will be adjusted as well for any enlargement. In this example, the MBR of node B is updated after the insertion of entry 10 while the root node does not need to be updated, as shown in Figure 3.1. Without any node splits in the FAR-Tree, during the insertion of entry 10, there are two node writes in total: one for creating the chain node, and one for updating the MBR of node B. Compared to the 5 writes by using the R-Tree insertion, we save 3 writes.

Overall, the number of writes saved offsets the additional reads at query time. With the imbalance structure of the FAR-Tree, we may need to read more nodes for a query. Query on chain nodes is executed sequentially. After traversing all entries in the leaf node, we also need to traverse entries in the chain nodes. For example when we are searching for the object 10 in Figure 3.1, we start from the root node, descend to the leaf node B, and after traversing entries 4, 5, and 6, we traverse the chain node and find the entry 10 inside; so we spend 3 node reads. In this example, there is only one chain node attached. However, when more objects have been inserted such that the chains are getting long, we will need to traverse more chain nodes for one query. With the growing chains, the extra reads for chain nodes traversal increase, and the savings gained from insertions may be offset by the queries. Therefore we propose a re-balancing for the FAR-Tree. The re-balancing re-organizes the FAR-Tree to a balanced status. It re-inserts all entries from the chain nodes to the R-Tree index using the R-Tree insertion algorithm. After the re-balancing, all chain nodes' entries are inserted back to the R-Tree and a balanced

Node ID	MBR	hasChain
Root	d0[low],d0[high],d1[low],d1[high]	1
A	d0[low],d0[high],d1[low],d1[high]	0
B	d0[low],d0[high],d1[low],d1[high]	1
C	d0[low],d0[high],d1[low],d1[high]	0

• • •

Figure 3.2: The In-memory table *T* for the FAR-Tree in Figure 3.1. The MBR is represented by coordinate pairs of all dimensions; a coordinate pair (di[low], di[high]) specifies the minimum and the maximum coordinates of the ith dimension. The hasChain label is a boolean stamp.

R-Tree index is constructed. Queries thereafter are on a balanced R-Tree index, until some subsequent insertions were to cause an overflow.

3.1.2 An In-memory Stamp Table

Every time we insert a new entry to a chain node, its corresponding leaf node's MBR needs to be updated if the new inserted entry enlarges the leaf node's bounding rectangle. The updates will propagate upwards until the root node for any internal nodes' MBR changes. The adjustment can take lots of node writes, as the MBRs are recalculated and written back to the nodes.

To eliminate the node writes caused by adjustment for chain nodes insertions, we set up a table *T* in memory storing the latest MBRs for leaf nodes and internal nodes. Adjustments caused by inserting to the original leaf nodes are executed on the tree index. For insertions to the chain node, we do not update MBRs in disk; all MBR changes will be reflected in the table. The table *T* is structured as in Figure 3.2, which corresponds to the FAR-Tree in Figure 3.1. In the table, each record represents a leaf node or an internal node, containing their latest MBR and a label for whether there is a chain attached. The MBR is represented in space coordinates and the hasChain label is a boolean stamp. We define that: (1) a leaf node *L* has a chain when there is at least one chain node attached to *L* and (2) an internal node *I* has a chain when any of the leaf nodes under this *I* has a chain attached. That is, for a leaf node, if it has at least one chain node attached, the "hasChain" label will be 1, and 0 otherwise; for an internal node, if any of the leaf nodes as its direct or indirect children has a chain, the “hasChain” label will be marked 1, and 0 otherwise. Initially, the table *T* is constructed from the existing base R-Tree; all nodes are marked 0 for their “hasChain” label and the MBRs are equal to those of the existing base R-Tree nodes. When any insertion creates a chain onto a leaf node, all nodes on the path from the leaf node to the root node will be marked 1 for “hasChain” and the nodes’ MBR along the path will be updated as needed in the table *T*. As the table shows in Figure 3.2, by attaching a chain node to the node B, both B and B’s parent node, the Root, are marked 1 for “hasChain”. When a query wants to get the MBR of a node, it will refer to the table *T* for the latest MBR if the node is marked 1 for “hasChain”, otherwise the query will refer to the index node on the disk directly. Upon the FAR-Tree re-balancing, the table *T* will be refreshed and rebuilt based on the MBRs of the re-constructed R-Tree. With this table, adjustments for chain nodes’ insertions are done in memory and thus we eliminate the disk writes.

The space taken by the table *T* is acceptable. Each record contains an *integer* for the node ID, four *double* for the MBR coordinates, and a *boolean* for the “hasChain” label. In addition, we only keep the leaf nodes and the internal nodes in the table; in real applications, the node capacity is usually large, which makes the number of nodes much smaller than the number of spatial objects. For example, the table for an index built on 1 million data objects takes less than 1 Megabyte.

3.1.3 The FAR-Tree Algorithms

Below we list the key algorithms for the FAR-Tree implementation. Our method modifies the insertion and the query. For the query, we focus on the intersection query.

Insertion

Algorithm 1 gives the pseudo code for the FAR-Tree insertion. The spatial object to be inserted is represented as an entry, which is a structure storing the object’s property information such as its unique ID and its coordinates in the space area. The insertion is processed in three steps. The first step is to choose a leaf node to place the inserting entry; the chosen leaf node is the one with least MBR enlargement to include the new entry. The second step is to insert the entry to either the chosen leaf node or the appended chain node; if the leaf node has space, the entry will be placed in the leaf node, otherwise it will call the *overflowNode* function in Algorithm 2 to insert the entry to the chain node. The third step is to adjust the leaf node and the internal nodes for any MBR enlargement.

Algorithm 1 Insert an object to the FAR-Tree

```
function INSERT(Entry E)
    CHOOSELEAF(E, Root);   ▷ Select a leaf node L in which to place E. Input the root node as a start.
    INSERTDATA(E, L);                                   ▷ Insert E to L.
end function

/* Choose the leaf node to place a new entry E. */
function CHOOSELEAF(E, R)
    N ← root node;
    if N is a leaf node then
        Return N;
    else
        Find the entry F in N whose rectangle needs least enlargement to include E, resolve ties by choosing the entry with the rectangle of smallest area;
        N ← the child node pointed to by F;
        CHOOSELEAF(E, N);      ▷ Descend recursively until reach a leaf node.
    end if
end function

/* Insert a new entry E to the chosen leaf L. */
function INSERTDATA(E, L)
    if L has room for E then
        Put E into L;
        Adjust the the MBRs of upper level(s) if needed;
        if the MBR of any nodes changed then
            Update its MBR record in $T$;
        end if
    else
        OVERFLOWNODE(E, L);                             ▷ When L is full.
    end if
end function
```

Algorithm 2 deals with the leaf node overflow, that the chosen leaf node does not have space for a new inserting entry. In the case when the leaf node's pointer

Algorithm 2 Node Overflow

```
1:  /* Deal with the case when a leaf node is overflow, where the number of entries
    is larger than the node capacity. */
2:  function OVERFLOWNODE(E, L)
3:      if L.next is NULL then      ▷ When there exists no chain node attached to L.
4:          Create a new chain node S;
5:          Put E into S;
6:          L.next ← S;
7:          Update the "hasChain" label in the table $T$ for all nodes on the path
8:      else
9:          C ← the last chain node C on L;
10:         if C is full then
11:             Create a new chain node S;
12:             Put E into S;
13:             C.next ← S;
14:         else
15:             Put E into C;
16:         end if
17:     end if
18:     UPDATEPATH(L, C, L.pathBuffer) ▷ update the MBRs on the path from the
    leaf node up to the root in table $T$
19: end function
20:
21: /* update MBR of all nodes along the path from the leaf node up to the root;
    input is the leaf node L, the chain Node C, and the pathBuffer of L. pathBuffer
    is a stack of the parent nodes on upper levels from L to the root. */
22: function UPDATEPATH(L, C, pathBuffer)
23:     Read from the table $T$ to get the MBR of L       ▷ Update leaf node's MBR;
24:     Combine the MBR of C and L;
25:     Update the MBR of L in table $T$;
26:     while pathBuffer is not empty do              ▷ Update internal nodes' MBR
27:         P ← Pop from pathBuffer to get the parent node;
28:         Read from the table $T$ to get the MBR of P;
29:         Combine the MBR of P and L;
30:         Update the MBR of P in table $T$;
31:     end while
32: end function
```

is NULL, which means there is no chain node attached on it, a chain node will be created and the pointer will be set to the newly created chain node. When there already exist one or more chain nodes attached, the new entry will be put to the last chain node; similarly, if the last chain node is full, a new chain node will be created and appended at the last place on that chain to accommodate the inserting entry. Finally, the *updatePath* function is to adjust the information in the table *T*.

Query

Algorithm 3 explains how an intersection query is executed in a FAR-Tree. The search area is represented using its space coordinates, which is in the same format as an MBR.

Algorithm 3 Query

```
1: /* Find all spatial objects whose rectangle intersect with a search rectangle S,
   input the root node as a start. */
2: function QUERY(S, R)
3:     N ← R;
4:     Check “hasChain” label of N from table T;
5:     if N has chain then
6:         Read from table T to get the MBR of N;
7:     else
8:         Read the MBR of N from the index itself;
9:     end if
10:    if N does not intersect in shape with S then
11:        Return;
12:    end if
13:    if N is not a leaf then
14:        Check each entry E in N to determine if intersecting with S;
15:        QUERY(S, the child node pointed to by E); ▷ Descend recursively with
   the intersecting node as input
16:    else
17:        Traverse all entries in N and check whether they intersect with S;
18:        Traverse all entries in the chain attached to N and check whether they
   intersect with S;
19:        Return those entries intersecting with S;
20:    end if
21: end function
```

A query starts from the root node and recursively descends to subtree(s) intersecting with the search area. When reading the entry’s MBR to determine an intersection, if the corresponding node is labeled 1 for "hasChain", the MBR will be read from the table *T* and otherwise from the index node. When the query reaches a leaf node, all entries in the leaf node and the attached chain nodes will be traversed for intersecting objects. Moreover, in one query, more than one subtree may intersect with the search area so the query may search more than one leaf nodes.

Re-balancing

Algorithm 4 gives the pseudo code for re-balancing the FAR-Tree. At the start, it retrieves all the entries from the chain nodes chain by chain. At the same time, the node pointer of all the leaf nodes will be set to NULL, which means there is no chain node attached on them. All the entries will then be inserted back to the tree using the balanced R-Tree insertion function *balancedInsertData* [4]. The last step is to refresh the table T based on the new constructed R-Tree.

Algorithm 4 Re-balancing

```
1: /* Re-insert all the chain nodes' entries to the R-Tree part and reconstruct a
   balanced R-Tree */
2: function REBALANCE( )
3:     LS ← a list of all leaf nodes of the tree
4:     while LS is not empty do
5:         Read one leaf node L from LS ;
6:         L.next ← NULL;
7:         for all Chain node C attached on L do
8:             for all Entry E in C do BALANCEDINSERTDATA(E);                ▷
   Insert each chain entries attached to L. The balancedInsertData is the insertion
   approach of the original R-Tree.
9:             end for
10:        end for
11:    end while
12:    Reconstruct table T based on the newly constructed tree;
13: end function
```

We may recall the re-insertion of the R*-Tree [2] from the FAR-Tree re-balancing, as both of them insert entries for a second time. As a variant of R-Tree, the R*-Tree forces to re-insert entries during the insertion routine when a node overflows. Entries of the overflowing node are sorted according to their locations and re-inserted in order. Remind that different sequences of insertions will result in different trees.

It is evidenced in the R*-Tree that the re-insertion improves the query performance as it decreases the nodes overlap. Similar to the R*-Tree, in the FAR-Tree re-balancing, entries are inserted chain by chain so they can be generally regarded as locationally sorted. Their difference is that the R*-Tree re-insertion sorts entries locally, i.e., within the overflowing node; while the FAR-Tree re-balancing sorts entries globally, i.e., over the entire tree. With the sorted re-inserting, we can expect that the resulting tree of the FAR-Tree re-balancing has smaller overlap and hence yields a slightly better query performance than the corresponding normally constructed R-Tree.

3.2 The FAR-Tree Disk Analysis

In this section, we analyze the costs on disk access for the FAR-Tree. The analysis is conducted with a comparison against the R-Tree. We discuss about the costs for both the insertion and the query. For the insertion, we show how the FAR-Tree saves costs compared to the R-Tree based on the insertion algorithms. For the query, we give a cost prediction model based on an existing model for the R-Tree [22].

3.2.1 The Insertion Cost Analysis

Applicable to both R-Tree and FAR-Tree, the insertion is composed of the following processes:

- Choosing a leaf node
- Adding an entry
- Adjusting the tree
- Dealing with node overflow

We will analyze the insertion cost through each of the process.

We list the notations in Table 3.1 for our analysis.

In the following analysis, the R-Tree and the FAR-Tree are inserting the same number of objects *NI* to the same base R-Tree containing *B* objects; the base R-Tree is the initial R-Tree that we will do insertions and queries and we call it the base tree in the following explanation. We have $NI = NI_{FAR1} + NI_{FAR2} = NI_{R1} + NI_{R2}$.

Symbols	Definitions
B	Number of entries in the base tree
M	Maximum node capacity
NI	Number of entries to be inserted in total
h	Height of the R-Tree
NI_{FAR1}	Number of entries inserted to the leaf nodes of the base tree (in the FAR-Tree)
NI_{FAR2}	Number of entries inserted to the chain nodes when the leaf nodes overflow (in the FAR-Tree)
NI_{R1}	Number of entries inserted before the tree height increase (in the R-Tree)
NI_{R2}	Number of entries inserted after the tree height increase (in the R-Tree)
C_i	Read cost for choosing leaf node on a tree of height *i*, per node insertion
avg	Average cost of MBR adjustment per each node insertion (quantified by the number of node adjusted)
f	average node capacity

Table 3.1: Notations for the insertion analysis

Choosing A Leaf Node

All disk accesses for choosing a leaf node are disk reads. Choosing a leaf node for insertion is to find a single leaf node with the least MBR enlargement to include the new inserting entry. Starting from the root node to the leaf node, with a tree of height h, the *chooseLeaf* will spend h disk reads. For the FAR-Tree, the base tree will never change its height unless a re-balance happens. For the R-Tree, however, with the node splits and adjustments, the structure and the height of the tree may change. For the reason that different sequences of insertions will build up different trees, we cannot determine at which point the height may increase in the R-Tree and thus we cannot decide whether NI_{R1} is smaller or larger than NI_{FAR1}; more explanations will be given in the following discussion.

When $NI_{R1} \geq NI_{FAR1}$, the proportion of *NI* is shown in Figure 3.3. An example for this case is given in Figure 3.4. In this example, we are to insert object 5 to 10 to the base tree shown on the top; assume the objects are inserted to node C according to their locations. In the R-Tree, as shown in Figure 3.4a, during the insertion of object 5 to 8, node C is split into C and D but the height does not change; the resulting tree for these operations is shown in the middle. Then we

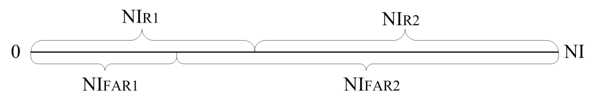
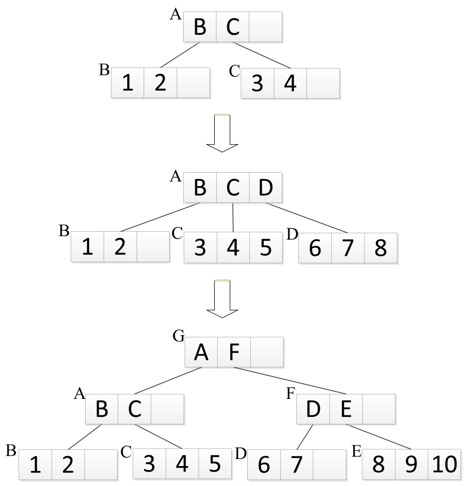
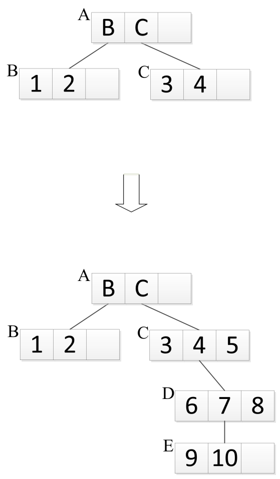


Figure 3.3: $NI_{R1} \geq NI_{FAR1}$



(a) R-Tree: $NI_{R1} = 5, NI_{R2} = 1$



(b) FAR-Tree: $NI_{FAR1} = 1, NI_{FAR2} = 5$

Figure 3.4: An example when $NI_{R1} \geq NI_{FAR1}$

insert object 9 to node D, and this increases the tree height, as the resulting tree shown at the bottom in Figure 3.4a. So five nodes are inserted before the tree height increase. Afterwards, the insertion of object 10 is to the tree with increased height. By contrast, in the FAR-Tree, as shown in Figure 3.4b, the object 5 is inserted to the original leaf node C while all other insertions are to the chain nodes. So for this example, $NI_{R1} = 5, NI_{R2} = 1, NI_{FAR1} = 1, NI_{FAR2} = 5$.

In this case, the tree height will not increase if we insert a number of NI_{FAR1} objects to an R-Tree with the R-Tree insertion; the R-Tree and the FAR-Tree have the same height and thus the costs for choosing a leaf during this number of insertions are the same. Then when we insert the NI_{FAR2} objects using the R-Tree insertion, $NI_{R1} - NI_{FAR1}$ are inserted to the tree with the original height *h*, while NI_{R2} are inserted to the grown tree with height *h*+*x*, where *x* is a positive integer representing the increment of the tree height. By contrast, with the FAR-Tree insertion, all the *NI* objects are inserted with the base tree being of height *h*. However, NI_{FAR2} insertions are to the chain nodes in the FAR-Tree and each insertion needs one more read of the last chain node to place the inserting object.

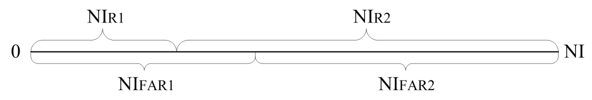
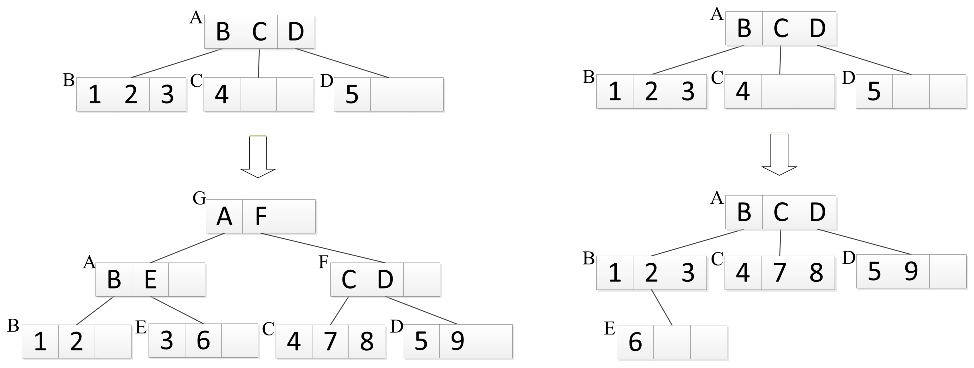


Figure 3.5: $NI_{R1} < NI_{FAR1}$



(a) R-Tree: $NI_{R1} = 1, NI_{R2} = 4$ (b) FAR-Tree: $NI_{FAR1} = 4, NI_{FAR2} = 1$

Figure 3.6: An example when $NI_{R1} < NI_{FAR1}$

So, comparing the disk reads of choosing leaf node for insertions, for the NI_{FAR1} objects, the R-Tree and the FAR-Tree cost the same; for the $NI_{R1} - NI_{FAR1}$ objects, the FAR-Tree costs 1 more read than the R-Tree per insertion; and for the NI_{R2} objects, the FAR-Tree cost x-1 less read than the R-Tree per insertion.

When $NI_{R1} < NI_{FAR1}$, the proportion of NI is shown in Figure 3.5. A corresponding example is given in Figure 3.6. In this example, we are to insert object 6 to 9 to the base tree shown at the top of the figure; assume the insertions are to the node B by their locations. In the R-Tree, as shown in Figure 3.6a, inserting object 6 triggers the node B to split; the split propagates upwards and causes the tree hight increase. So insertions of the objects 7 to 9 afterwards are to the tree with increased height (in this example, assume objects 7 to 9 are inserted to node C and D by their locations). In the FAR-Tree, by contrast, as shown in Figure 3.6b, the object 6 is inserted to the chain node, which does not change the tree height; objects 7 to 9 are inserted to the original leaf node C and D. So for this example, $NI_{R1} = 1$, $NI_{R2} = 4$, $NI_{FAR1} = 4$, $NI_{FAR2} = 1$.

In this case, using the R-Tree insertion, the height of the tree will increase at some point during the insertion of NI_{FAR1} objects; NI_{R1} are inserted before the height changes and $NI_{FAR1} - NI_{R1}$ are inserted after the height changes. Then, for the

NI_{FAR2} objects, when using the R-Tree insertion, all are inserted to the tree with height increased. Using the FAR-Tree insertion, the tree height keeps unchanged, but inserting to a chain node needs one more read of the chain tail.

So, comparing the disk reads of choosing leaf node for insertions, for the NI_{R1} objects, the R-Tree and the FAR-Tree cost the same; for the $NI_{FAR1} - NI_{R1}$ objects, the FAR-Tree costs x less reads than the R-Tree per insertion; and for the NI_{FAR2} objects, the FAR-Tree costs x-1 less reads than the R-Tree.

In summary, in the process of choosing a leaf node, when $NI_{R1} \geq NI_{FAR1}$, the R-Tree costs:

$$\begin{aligned} R &= NI_{FAR1} \cdot C_h + (NI_{R1} - NI_{FAR1}) \cdot C_h + NI_{R2} \cdot C_{h+x} \\ &= NI_{FAR1} \cdot h + (NI_{R1} - NI_{FAR1}) \cdot h + NI_{R2} \cdot (h + x) \end{aligned}$$

The FAR-Tree costs:

$$\begin{aligned} FAR &= NI_{FAR1} \cdot C_h + (NI_{R1} - NI_{FAR1}) \cdot (C_h + 1) + NI_{R2} \cdot C_{h+1} \\ &= NI_{FAR1} \cdot h + (NI_{R1} - NI_{FAR1}) \cdot (h + 1) + NI_{R2} \cdot (h + 1) \end{aligned}$$

To compare the R-Tree and the FAR-Tree, we have:

$$R \text{ - } FAR = NI_{FAR1} - NI_{R1} + NI_{R2} \cdot (x - 1)$$

When $NI_{R1} < NI_{FAR1}$, the R-Tree costs:

$$\begin{aligned} R &= NI_{R1} \cdot C_h + (NI_{FAR1} - NI_{R1}) \cdot C_{h+x} + NI_{FAR2} \cdot C_{h+x} \\ &= NI_{R1} \cdot h + (NI_{FAR1} - NI_{R1}) \cdot (h + x) + NI_{FAR2} \cdot (h + x) \end{aligned}$$

The FAR-Tree costs:

$$\begin{aligned} FAR &= NI_{FAR1} \cdot C_h + NI_{FAR2} \cdot (C_h + 1) \\ &= NI_{FAR1} \cdot h + NI_{FAR2} \cdot (h + 1) \end{aligned}$$

To compare the R-Tree and the FAR-Tree:

$$\begin{aligned} R \text{ - } FAR &= x \cdot (NI_{FAR2} + NI_{FAR1} - NI_{R1}) - NI_{FAR2} \\ &= NI_{R2} \cdot x - NI_{FAR2} \end{aligned}$$

For the case when $NI_{R1} \geq NI_{FAR1}$, we cannot determine whether R - FAR is larger or smaller than 0, with $NI_{R2} \cdot (x-1) \geq 0$ but $NI_{FAR1} - NI_{R1} \leq 0$. However, if inserting large amount of objects such that the height of the R-Tree increases a lot, i.e., x is large, *R-FAR* would be positive, which means the FAR-Tree costs less than the R-Tree. Even when $R\text{-}FAR < 0$, the cost difference is within $NI_{R1} - NI_{FAR1}$ reads; the costs for this process are read access, which is cheap for SSDs. The excessive reads of the FAR-Tree in this case may be easily set off by write savings. For the case when $NI_{R1} < NI_{FAR1}$, since $NI_{FAR2} < NI_{R2}$ and $x \geq 1$, the FAR-Tree always costs less than the R-Tree.

Adding An Entry

After choosing a node, the selected node is in memory, so there is no disk access for adding an entry to the node. The costs for the FAR-Tree and the R-Tree are the same for this process.

Adjusting Tree

The adjustment requires node updates, which includes both disk reads and disk writes. On the FAR-Tree, NI_{FAR1} objects are inserted into the original leaf nodes of the base tree. The adjustment for this part of insertions is performed the same as the R-Tree, which updates the MBRs of internal nodes in disk. For the remaining NI_{FAR2} objects, which are inserted to the chain nodes, all MBR changes are reflected on the in-memory table, so there is no disk access for this part. On the R-Tree, by contrast, all the NI insertions may cause disk access for adjustments.

We thus denote avg_{FAR} as the average number of disk access for adjustment per insertion in the FAR-Tree (avg_{FAR} is only applicable to the objects inserted to the leaf nodes of the base tree, because the NI_{FAR2} adjustments are using the in-memory table T and do not interact with the disk), and avg_R as the average costs for adjustment per insertion in the R-Tree.

So the cost of adjustment on the FAR-Tree is:

$$FAR = NI_{FAR1} \cdot avg_{FAR} + NI_{FAR2} \cdot 0 = NI_{FAR1} \cdot avg_{FAR}$$

The cost of adjustment on the R-Tree is:

$$R = NI \cdot avg_R$$

For the avg_{FAR} and the avg_R, we cannot get a precise value for them, but as an estimation, the avg_{FAR} is no larger than the avg_R. The maximum number of nodes to be adjusted in an insertion equals to the height of the tree, in which case we assume all nodes on the path from the leaf node toward the root node need to be updated. The height of the R-Tree may increase with entries inserted, while the height of the FAR-Tree keeps unchanged, so it is likely that the R-Tree needs to update more nodes for adjustment. With the estimation that $avg_{FAR} \leq avg_R$, and the fact that $NI_{FAR1} \leq NI$, the FAR-Tree costs no more disk writes than the R-Tree on adjusting the tree. Furthermore, when most of the entries are inserted to the chain nodes, *NI* would be far larger than NI_{FAR1}, and in this case the FAR-Tree would yield more savings for adjustment.

Node Overflow

When inserting an entry to a leaf node which is full, in the R-Tree the leaf node will split and accordingly in the FAR-Tree a chain node will be created. The number of node splits should be equal to the number of newly created nodes. Disk access for splitting or creating nodes are about disk writes.

For better understanding, we analyze a single leaf node *L* as a start, considering both the R-Tree and the FAR-Tree. We assume that there are originally a number of *a* entries in *L* and we are to insert *b* entries. With the FAR-Tree insertion, *M - a* entries are inserted to *L* itself and *b - (M - a)* entries are inserted to the newly created chain nodes. With the R-Tree insertion, *L* will split into multiple leaf nodes. In the FAR-Tree, the number of newly created chain nodes is $\lceil \frac{b\text{ - }(M\text{ - }a)}{M} \rceil$. In the R-Tree, besides creating new leaf nodes upon splitting, the leaf node splitting may cause the internal node to split. The number of leaf node splits is $\lceil \frac{a+b}{M} \rceil - 1$, which is the number of leaf nodes minus one. The number of internal node splits is just the number of all internal nodes created, that is $\lceil \frac{a+b}{M^2} \rceil + \lceil \frac{a+b}{M^3} \rceil + ... + \lceil \frac{a+b}{M^n} \rceil$, where *n* represents the height after the insertions. The number of newly created nodes is the sum of these two parts: $\lceil \frac{a+b}{M} \rceil - 1 + \lceil \frac{a+b}{M^2} \rceil + \lceil \frac{a+b}{M^3} \rceil + ... + \lceil \frac{a+b}{M^n} \rceil$. Although nodes in the R-Tree are not always full, for simplicity of analysis, we conservatively assume that all leaf nodes are full in the R-Tree after insertions, which is a worst case for our comparison that the R-Tree insertions cause least number of node splits.

Based on the analysis for a single node above, we calculate the costs for the whole tree. We assume the entries are inserted evenly: after all the insertions in the FAR-Tree, the number of chain nodes attached to each leaf node tends to be the same; and accordingly in the R-Tree, the number of entries inserted toward each leaf node branch is generally the same. So for each leaf node, we will insert $\frac{NI}{\frac{B}{f}} = \frac{NI \cdot f}{B}$ entries, with $\frac{B}{f}$ leaf nodes in the base tree.

The number of splits in the R-Tree is:

$$S_R = \frac{B}{f} \cdot (\lceil \frac{\frac{NI \cdot f}{B} + f}{M} \rceil - 1 + \lceil \frac{\frac{NI \cdot f}{B} + f}{M^2} \rceil + ... + \lceil \frac{\frac{NI \cdot f}{B} + f}{M^n} \rceil)$$

The number of newly created chain nodes in the FAR-Tree is:

$$S_{FAR} = \lceil \frac{NI_{FAR2}}{M} \rceil = \lceil \frac{NI - NI_{FAR1}}{M} \rceil = \lceil \frac{NI - (M - f) \cdot \frac{B}{f}}{M} \rceil$$

Then, for the disk access, each split in the R-Tree costs 3 disk writes: one for the split node, one for the newly created node, and one for their parent node. So the cost for the R-Tree overflow is:

$$R = 3 \cdot S_R$$

Each overflow in the FAR-Tree costs 2 writes: one for creating the new chain node and one for updating the pointer of the previous node. So the cost for the FAR-Tree overflow is:

$$FAR = 2 \cdot S_{FAR}$$

$$
\begin{aligned}
FAR\text{ - }R &= 2\cdot\frac{NI-(M-f)\cdot\frac{B}{f}}{M}-\frac{3B}{f}\cdot\left(\frac{\frac{NI\cdot f}{B}+f}{M}-1+\frac{\frac{NI\cdot f}{B}+f}{M^2}+\ldots+\frac{\frac{NI\cdot f}{B}+f}{M^n}\right)\\
&= 2\cdot\frac{NI-(M-f)\cdot\frac{B}{f}}{M}-\frac{3B}{f}\cdot\left(\frac{NI\cdot f}{B}+f\right)\cdot\left(\frac{1}{M}+\frac{1}{M^2}+\ldots+\frac{1}{M^n}\right)+\frac{3B}{f}\\
&= \frac{3\cdot B\cdot M+2NI\cdot f-2B\cdot(M-f)}{M\cdot f}-3(NI+B)\cdot\frac{\frac{1}{M}(1-\frac{1}{M^n})}{1-\frac{1}{M}}\\
&\left(let\ \frac{1}{M^n}\approx 0\right)\\
&= \frac{B\cdot M+2NI\cdot f+2B\cdot f}{M\cdot f}-\frac{3(NI+B)}{M-1}\\
&< \frac{B\cdot M+2NI\cdot f+2B\cdot f}{M\cdot f}-\frac{3(NI+B)}{M}\\
&= \frac{B\cdot M-B\cdot f-NI\cdot f}{M\cdot f}\\
&= \frac{B\cdot M-B\cdot f-\frac{NI}{\frac{B}{f}}\cdot B}{M\cdot f}\\
&= \frac{B\cdot(M-f-\frac{NI}{\frac{B}{f}})}{M\cdot f}
\end{aligned}
$$

For the inequation above, $\frac{NI}{\frac{B}{f}}$ is the number of entries inserted to each leaf node of the base tree, as we assume that entries are inserted to the tree evenly. For both the R-Tree and the FAR-Tree, when a node overflows, the number of existing entries f in the node plus the inserted entries is larger than the node's maximum capacity, that is, $\frac{NI}{\frac{B}{f}}+f > M$; so we have $\frac{B\cdot(M-f-\frac{NI}{\frac{B}{f}})}{M\cdot f} < 0$. Therefore, $FAR-R<0$, which means that the FAR-Tree costs less writes than the R-Tree for dealing with node overflows.

Summarizing all the processes of insertions above, the FAR-Tree insertion costs less than the R-Tree on the whole.

3.2.2 The Query Cost Analysis

A model for predicting the R-Tree query performance is given in [22]. Our FAR-Tree query analysis is conducted based on that R-Tree model.

The notations used in analyzing the query performance are listed in Table 3.2.

As per the R-Tree model, the number of disk access for an R-Tree query is 1

Symbols	**Definitions**
B	Number of entries in the base tree
NI	Number of entries to be inserted
NI_{FAR1}	For the FAR, number of entries inserted to the leaf nodes of the base tree
N	Amount of objects in the R-Tree to be queried
n	Number of dimensions
D	Density of a dataset
$q = (q_1, ...q_n)$	Query window
M	Maximum node capacity
f	Average node capacity
h	Height of the R-Tree
N_j	Nunber of R-Tree nodes at level j
$s_j = (s_{j,1}, ...s_{j,n})$	Average size of an R-Tree node at level j
DA	Number of disk access for a query window q

Table 3.2: Notation for the query analysis

(one access for the root) plus the number of intersected nodes at every level j (j = 1, ..., h - 1). The root node is defined at level h and the leaf nodes is at level 1. With an R-Tree in height h, the disk access to answer a query $q = (q_1, ..., q_n)$ is expected to be:

$$DA = 1 + \sum_{j=1}^{h-1} intersect(N_j, s_j, q) \tag{3.1}$$

where $intersect(N_j, s_j, q)$ is the number of intersected nodes at level j.

For the FAR-Tree, we can view it as two parts if dividing by the leaf level: the upper part is in the same shape as the base tree (including the internal nodes and the leaf nodes) and the lower part are the chains. The query cost for the FAR-Tree is composed of the cost for searching on the base tree part and the cost for traversing the chain nodes.

The cost for searching on the base tree index can be estimated with Equation 3.1. For the chain nodes traversal, the number of chains needed to be traversed is equal to the number of nodes at level 1 intersected by the query window q (the intersected leaf nodes). Still, we assume the inserted entries are distributed evenly such that the number of chain nodes attached to each leaf nodes are the same. So we will insert $\frac{NI}{\frac{B}{f}} = \frac{NI \cdot f}{B}$ number of entries for each leaf node. Then the average length of the chains is $\left\lceil \frac{\frac{NI\cdot f}{B}+f}{M} - 1 \right\rceil = \left\lceil \frac{NI\cdot f - B\cdot M + B\cdot f}{B\cdot M} \right\rceil$.

Thus the expected number of disk access to answer a query q in the FAR-Tree is:

$$\begin{aligned} DA &= 1 + \sum_{j=1}^{h-1} intersect(N_j, s_j, q) + intersect(N_1, s_1, q) \cdot chainLength\_average \\ &= 1 + \sum_{j=1}^{h-1} intersect(N_j, s_j, q) + intersect(N_1, s_1, q) \cdot \left\lceil \frac{NI \cdot f - B \cdot M + B \cdot f}{B \cdot M} \right\rceil \end{aligned} \tag{3.2}$$

where $intersect(N_1, s_1, q)$ is the number of intersected leaf nodes.

A detailed analysis is given in the R-Tree model [22]:

1. With a unit work space $[0,1)^n$, $intersect(N,s,q) = intersect(N,s',0) = D(N,s') = N \cdot \prod_{i=1}^{n}(s'_i)$ where $s'_i = s_i + q_i$. The average number of intersection is equal to the density of the nodes' bounding rectangles inflated by q_i at each direction.

$$intersect(N,s,q) = N \cdot \prod_{i=1}^{n}(s_i + q_i) \tag{3.3}$$

2. The height of an R-tree storing N objects with an average capacity is:

$$h = 1 + \left\lceil \log_f \frac{N}{f} \right\rceil \tag{3.4}$$

3. With every node containing f entries on average, we can assume that the average number of leaves is $N_1 = \frac{N}{f}$, based on which we get the average number of nodes on the higher level is $N_2 = \frac{N_1}{f}$. Following this idea, the average number of nodes at level j is

$$N_j = \frac{N}{f^j} \tag{3.5}$$

4. Assuming that the node sides are equal (i.e., $s_{j,1} = s_{j,2} = \ldots = s_{j,n}, \forall j$), then the density of nodes at level j (j = 1, , h-1) is:

$$D_j = N_j \cdot \prod_{i=1}^{n} s_{j,i} = \frac{N}{f^j}(s_{j,i})^n \Rightarrow s_{j,i} = (D_j \cdot \frac{f^j}{N})^{\frac{1}{n}} \tag{3.6}$$

5. Using Eq.3.3 through Eq.3.6, the disk access for an R-Tree is:

$$\begin{aligned} DA &= 1 + \sum_{j=1}^{h-1} intersect(N_j, s_j, q) \\ &= 1 + \sum_{j=1}^{\lceil \log_f \frac{N}{f} \rceil} \{ \frac{N}{f^j} \cdot \prod_{i=1}^{n} \sqrt[n]{D_j \cdot \frac{f^j}{N}} + q_i \} \end{aligned} \tag{3.7}$$

In our analysis, for the R-Tree, we insert *NI* data to the base tree containing *B* objects. So after the tree is constructed, the total amount of entries *N* is *B + NI*. Therefore, the disk access for the R-Tree is:

$$DA = 1 + \sum_{j=1}^{\lceil \log_f \frac{B+NI}{f} \rceil} \{ \frac{B+NI}{f^j} \cdot \prod_{i=1}^{n} \sqrt[n]{D_j \cdot \frac{f^j}{B+NI}} + q_i \} \tag{3.8}$$

For the FAR-Tree, the number of data entries in the original index (internal nodes and leaf nodes) is $N = B + NI_{FAR1}$. So the disk access for the FAR-Tree is:

$$\begin{aligned} DA = 1 + \sum_{j=1}^{\lceil \log_f \frac{B+NI_{FAR1}}{f} \rceil} \{ \frac{B+NI_{FAR1}}{f^j} \cdot \prod_{i=1}^{n} \sqrt[n]{D_j \cdot \frac{f^j}{B+NI_{FAR1}}} + q_i \} + \\ \frac{B+NI_{FAR1}}{f} \cdot \prod_{i=1}^{n} (\sqrt[n]{D_1 \cdot \frac{f}{B+NI_{FAR1}}} + q_i) \cdot \left\lceil \frac{NI \cdot f - B \cdot M + B \cdot f}{B \cdot M} \right\rceil \end{aligned} \tag{3.9}$$

The *f* value in the FAR-Tree should be larger than that in the R-Tree, because that without node splits, the nodes in a FAR-Tree tends to be full, and thus the average number of entries in each node should be larger than the R-Tree; the *f* value in the FAR-Tree model is more closer to the *M* value.

In summary, based on the R-Tree performance prediction model, we now get a model for the FAR-Tree query, which can be estimated by using only the dataset properties *N* and *D*, the typical index parameter *f* and the query window *q*.

Chapter 4

Experiment and Result

In this chapter, we discuss experimental results performed to test the I/O performance for the FAR-Tree insertion, query, and re-balancing, with a comparison to the R-Tree. We examine both the logical I/O and the real disk access time. Agreeing with the cost analysis in Chapter 3, we observe that the FAR-Tree costs less for insertions while more for queries. Also, the cost for re-balancing is acceptable as the memory is very well utilized in the FAR-Tree when re-inserting the chain entries. Our experiments are conducted on both synthetic data and real data; using the synthetic data allows us to investigate several data properties, such as the distribution and the size of the spatial object.

4.1 Experiment Setup

The FAR-Tree approach was implemented by modifying the R-Tree implementation by Marios Hadjieleftheriou in C++[1] under Windows. We conduct the experiments on our FAR-Tree against the R-Tree for comparison. Important parameters of the experiment settings are listed in Table 4.1, and we will explain in detail about the settings in the following.

4.1.1 Dataset

The synthetic datasets are generated using the Spatial Data Generator by Yannis Theodoridis[2]. The datasets are in a 2-dimensional space area in size of 10,000 $\times$

[1]R-Tree implementation libspatialindex-1.7.1: http://libspatialindex.github.io/index.html
[2]Spatial Data Generator: http://www.chorochronos.org/?q=node/49

Synthetic Dataset	Distribution: Uniform, Gaussian, Zipf Object Size: 0.01%, 0.1%, 1%
Real Dataset	Germany, Greece
Page Size	4096 bytes
Node Capacity	40
Buffer Size (B)	10, 25, 50 pages
Logical Write/Read Ratio	7

Table 4.1: Parameters of the experiment settings

10,000. We have the following three distributions of centers of the rectangles:

- The Uniform distribution
- The Gaussian distribution. The mean μ is set to 5,000 and the variance σ is set to 2,000.
- The Zipfian distribution. The exponent p is set to 1, which means a moderate skew.

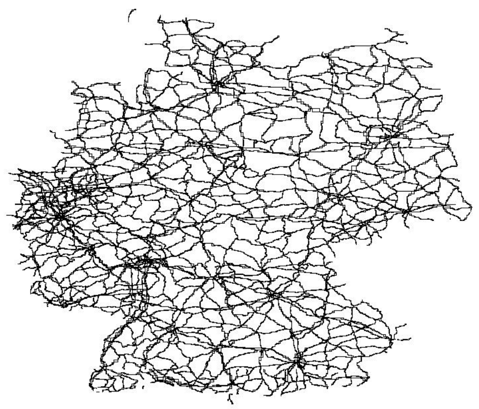
At the same time, for each distribution, we set different sizes for the spatial objects. Measured in the percentage of the space area taken by a single object to the entire space area on average, we have 0.01%, 0.1%, and 1%; corresponding to the side length ranging between 1 to 100, 1 to 316, and 1 to 1,000.

For the real geographical data, we use the streets of Germany dataset[3] and the streets of Greece dataset[4]. The Germany dataset has 30,674 spatial objects in total and the Greece dataset has 23,268 objects. Their spatial distribution are displayed in Figure 4.1. Without loss of generality, we shuffled the data before we use, because the original datasets are given sorted by location.

In the experiments, for all datasets, we use 5,000 objects to construct the base tree. We will insert the remaining data to the base tree to examine the insertion. For the query data, the synthetic query data has the same distribution and object size as the data in the tree; the query data for the real dataset are randomly extracted from the Germany dataset and the Greece dataset respectively.

[3]Germany Dataset: http://www.chorochronos.org/?q=node/54

[4]Greece Dataset: http://www.chorochronos.org/?q=node/55


(a) The streets of Germany with 30,674 spatial objects

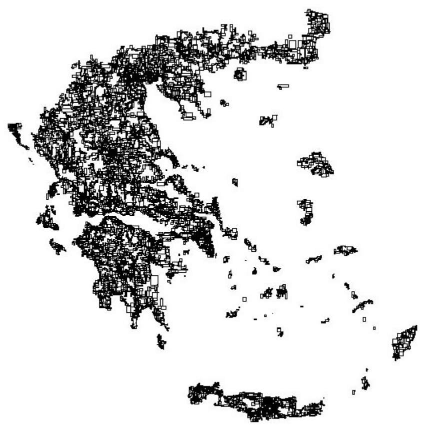

(b) The streets of Greece with 23,268 spatial objects

Figure 4.1: The real dataset distribution

4.1.2 Page Size and Buffer

A node is stored in one disk page and we choose the page size to be 4096 bytes, since having a node in size corresponding to the size of a disk page makes good performance [4] and the SSD we use in our experiments has a page size of 4096 bytes. Also, using larger page size leads to similar performances as for smaller dataset, and vice versa, as the number of nodes should be the same.

In the implementation, a full node is composed of both the information about itself and the information of all the entries contained in the node. The size of a node is calculated as below, with the correlating data type in parenthesis (The *Next Pointer* only exists in the leaf node and the chain node of FAR-Tree.):

$$\begin{aligned} \mathit{Node} = &\mathit{Node\ Type(int32)} + \mathit{Node\ Level(int32)} + \mathit{Next\ Pointer(int64)} + \\ &\mathit{Number\ of\ Children(int32)} + \mathit{MBR} + \mathit{Capacity} \times (\mathit{MBR} + \\ &\mathit{Entry\ ID(int64)} + \mathit{Data\ Length(int32)} + \mathit{Data}) \end{aligned}$$

The MBR is represented in coordinates as:

$$\mathit{MBR} = \mathit{Dimension} \times (\mathit{High\ Coordinate(double)} + \mathit{Low\ Coordinate(double)})$$

Adding up the size of data types used by each components, for a two-dimensional space area, we can calculate the size of one node in byte as:

$$\textit{Node} = \textit{52} + \textit{Capacity} \times \textit{(44 + Data)}$$

The *Data* stores the location of each entry in bytes for disk stream reading and writing. It has a variable length depending on the number of digits in the objects' coordinates. In the Germany dataset and the synthetic datasets, it is 34 bytes on average; in the Greece dataset, which comes with longer coordinates, it is 54 bytes on average. So, a page in size of 4096 bytes can hold 42 objects for the Greece dataset and 51 objects for the synthetic or the Germany datasets. For consistency, we set the maximum node capacity as 40 for all nodes in our experiments.

Despite the experiment settings, one node can also be stored in multiple pages when the node size is larger than the page size; the node will be divided into separate pages in this case.

As for the buffer management, we simply adopt the random eviction strategy, since our approach does not particularly manage the buffer. However, to investigate the effects of the buffer size, we set buffer size (denoted as B) to 10 pages, 25 pages, and 50 pages.

4.1.3 Other Experiment Parameters

When constructing the base tree, we set the node's fill factor as 40%, because the fill factor cannot be larger than 50% for linear and quadratic split method. As we observe for all the datasets, the constructed base tree has a utilization around 70%, which means the leaf nodes are 70% full on average. We adopt the quadratic split method when doing the R-Tree insertion and we use the intersection query.

We measure both the logical I/Os and the real disk access time. Measuring the logical I/Os eliminates the impact of different SSDs' I/O performances. By the I/O performances we mean the write/read ratio, which is the proportion of the speed of writing to the speed of reading. In our experiments, we examine the number of disk reads and writes. We regard the cost of one read as the cost unit, and the cost of one write is multiple times the cost of a read since writing on SSD is slower than reading. Assuming C_r and C_w are the number of disk reads and disk writes, then the total I/O cost is normalized into $1 \times C_r + \textit{ratio} \times C_w$, where the *ratio* is the write/read ratio of some specific disk. Typically the write/read ratio of SSDs ranges from 2 to 30, for example recent work for B$^+$-Tree use the value 5 for the ratio [23]. We assume a ratio of 7 in ours, which means that write operation is 7 times more expensive than read operation.

Also, we tested the real disk access time using a real SSD device[5]. Experiments are conducted on the 64-bit *Windows 7 Professional* operating system. The processor is *Intel Core i7-3770 3.4GHz*, the memory is 32GB, and the file system is *NTFS*. The *Process Monitor*[6] returns the duration of disk file access. We add up the read and write durations as the total disk access time.

4.2 Results In Logical I/Os

4.2.1 Insertion Evaluation

To evaluate the insertion, we inserted data to the base tree and compare the costs of the R-Tree and the FAR-Tree.

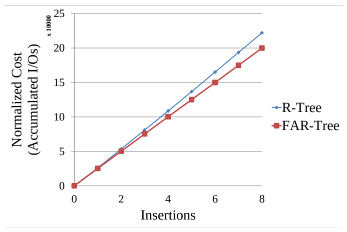
For the synthetic datasets, we insert 20,000 objects to the corresponding base trees. For the Germany data, we insert 25,000 objects to the base tree, and for the Greece dataset we insert 17,500 objects. Figure 4.2, Figure 4.3, and Figure 4.4 show the costs for the synthetic datasets. Figure 4.5 and Figure 4.6 show the costs for the Germany dataset and the Greece dataset. For all the charts, the vertical axis is the accumulated I/O costs for all insertions, represented in the normalized I/O, i.e. $1 \times C_r + \textit{ratio} \times C_w$; the horizontal axis represents the round of operations. We divide the insertions into several rounds, with each round containing 2,500 operations; in the figures, we note the accumulated I/O costs after a corresponding round.

As an overview, we observe that the FAR-Tree insertion costs less than the R-Tree insertion for all the testing datasets. This confirms with our insertion analysis in Chapter 3.

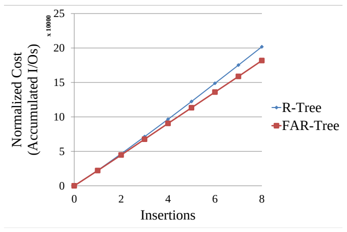
Furthermore, by comparing the results among the synthetic datasets, we evaluate the impact of the dataset property and the buffer size. Figure 4.2, Figure 4.3,

[5]SSD model: OCZ Vertex 3 VTX3-25SAT3-480G

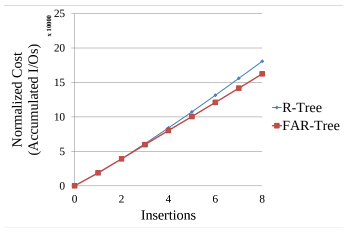
[6]Windows Sysinternals: http://technet.microsoft.com/en-ca/sysinternals/bb896645



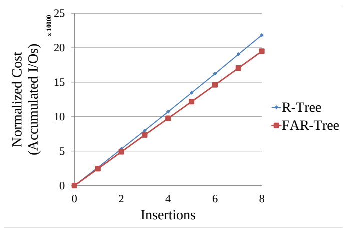
(a) Uniform, *B*=10



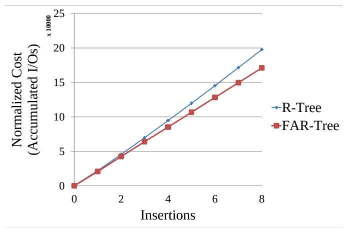
(b) Uniform, *B*=25



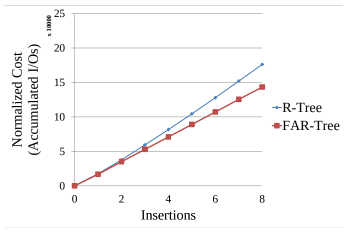
(c) Uniform, *B*=50



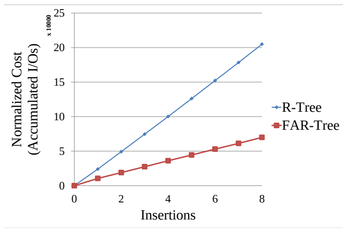
(d) Gaussian, *B*=10



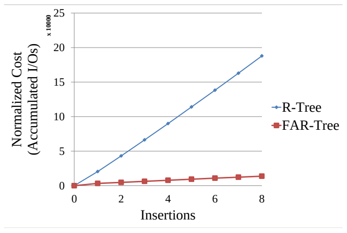
(e) Gaussian, *B*=25



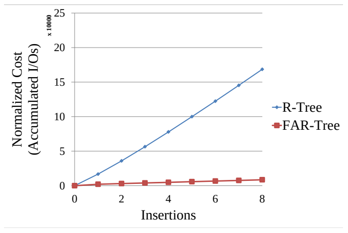
(f) Gaussian, *B*=50



(g) Zipf, *B*=10

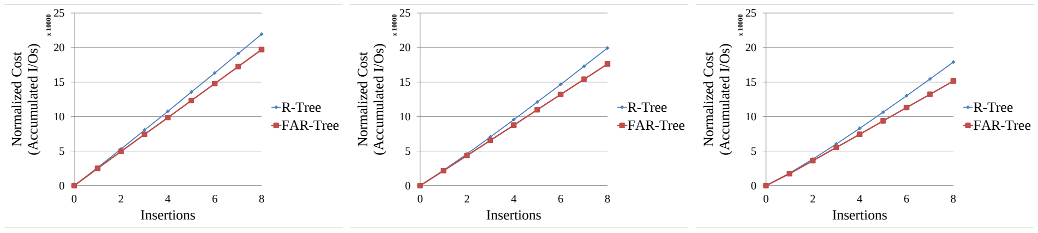


(h) Zipf, *B*=25

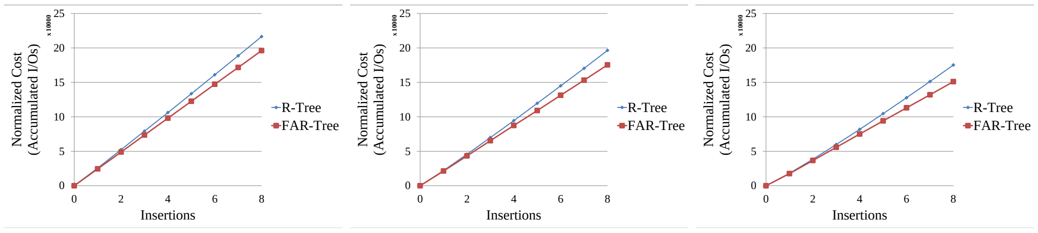


(i) Zipf, *B*=50

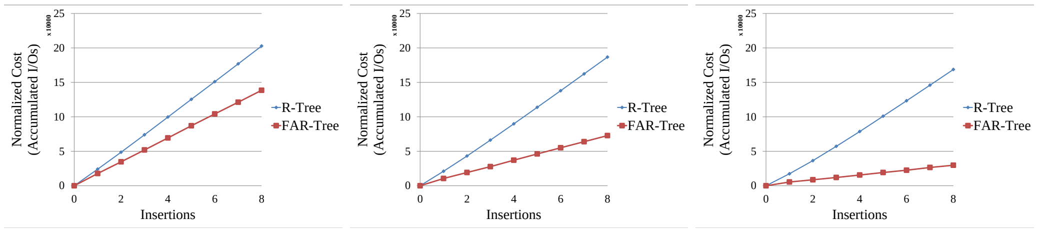
Figure 4.2: Insertion evaluation in logical I/Os; average MBR size: 0.01%



(a) Uniform, *B*=10 (b) Uniform, *B*=25 (c) Uniform, *B*=50



(d) Gaussian, *B*=10 (e) Gaussian, *B*=25 (f) Gaussian, *B*=50



(g) Zipf, *B*=10 (h) Zipf, *B*=25 (i) Zipf, *B*=50

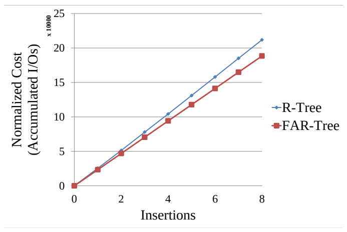
Figure 4.3: Insertion evaluation in logical I/Os; average MBR size: 0.1%

and Figure 4.4 are results for different object size, and within each of the figures are results for different distributions and buffer sizes. Each row corresponds to one kind of distribution; charts on the same row have the same distribution but different buffer size.

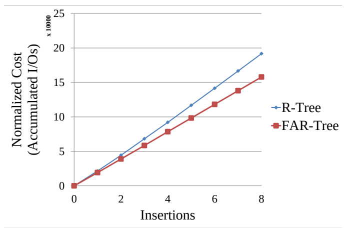
For the impact of data distribution, we can observe that the more skewed the dataset is, the more savings the FAR-Tree insertion is able to achieve. For example in Figure 4.2 using $B = 10$, the FAR-Tree insertion costs 10% less than the R-Tree on the Uniform dataset, on the Gaussian dataset it costs 15% less, and on the Zipf dataset it costs 65% less. The reason is that, for more skewed datasets, the spatial objects are more concentrated within a small area; correspondingly, insertions on the tree index are focused on some few branches. In this case, most of the insertions are done on chain nodes and the benefit of FAR-Tree insertion is the best. More node splits are eliminated. Lots of disk updates are saved, since inserting entries to the chain nodes does not cost disk access for adjusting internal nodes' MBRs (adjustment operates on the in-memory table *T*). Also, with the insertions more concentrated, the buffer can be better utilized. Successive insertions may target the same node, which is mostly available in the buffer and hence no disk interaction is needed. Corresponding to the results in Figure 4.2, after inserting the 20,000 objects, the longest chain in the FAR-Tree for the Uniform dataset is 14 in length, the Gaussian has the longest chain with 20, and the Zipf has the longest chain with 178. As for the leaf node *utilization*, defined as the ratio of the sum of entries in the leaf nodes to the sum of capacity of the leaf nodes, which express the fullness of the leaf nodes, the Uniform has a utilization of 98%, the Gaussian is 92% and the Zipf is 72%. Having long chains but small leaf node utilization is an evidence that most of the entries are inserted to the chain nodes attached on few of the leaf nodes and most leaf nodes does not have chains attached, for example the Zipf distribution. Conversely, for the Uniform distribution, the leaf nodes are almost full and the longest chain is only 20 in length, in which case insertions are evenly distributed over the leaf nodes.

For the object size, we cannot observe an direct influence. The result is still related to the locality of insertions. For example with the Zipf dataset, we compare the results among Figure 4.2, Figure 4.3, and Figure 4.4. The dataset with object size 0.01% has smaller costs than the 0.1% and the 1%, where costs for the 0.1% and the 1% are similar. The FAR-Tree for the 0.01% has a utilization of 72% with the longest chain in length of 178, the 0.1% has a utilization of 79% with a longest chain in 120, and the 1% has a utilization of 74% with a longest chain in 107. We can see that the insertion for the 0.01% is more concentrated and this explains its smallest costs.

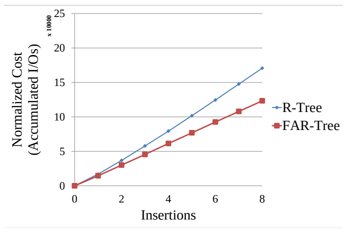
For the impact of buffer size, increasing the buffer size will decrease the costs in general for both the R-Tree and the FAR-Tree. If we compare the three charts of a same row, for all the cases, the costs for both the R-Tree and the FAR-Tree decrease, which is a reasonable influence of a buffer in general. Moreover, the savings by the FAR-Tree increase as the buffer grows and this is more obvious on more the skewed data. The buffer is more important in the FAR-Tree because of the chain nodes insertion. The skewed data better utilizes the buffer for its concentrated



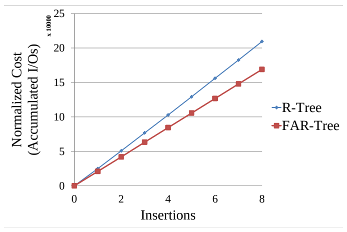
(a) Uniform, *B*=10



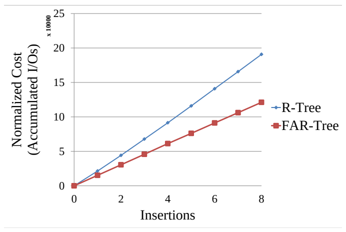
(b) Uniform, *B*=25



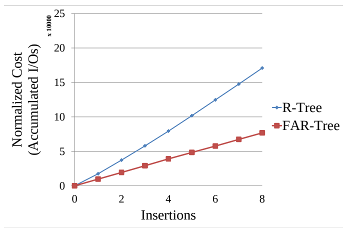
(c) Uniform, *B*=50



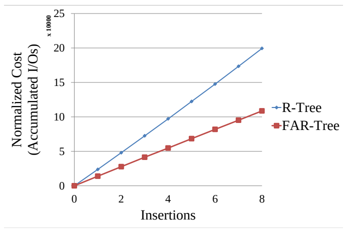
(d) Gaussian, *B*=10



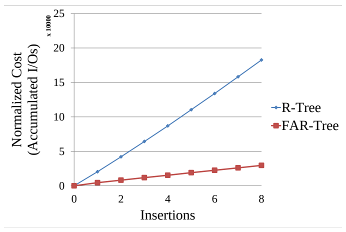
(e) Gaussian, *B*=25



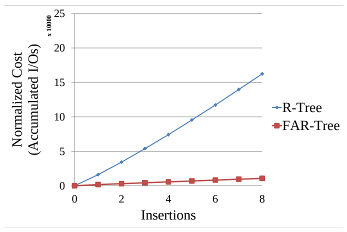
(f) Gaussian, *B*=50



(g) Zipf, *B*=10

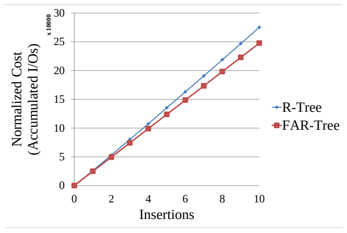


(h) Zipf, *B*=25

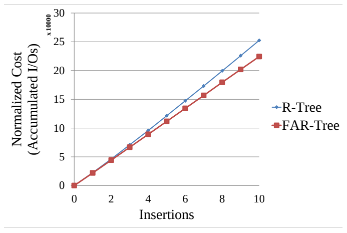


(i) Zipf, *B*=50

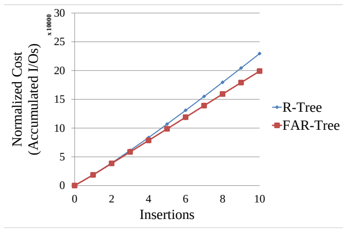
Figure 4.4: Insertion evaluation in logical I/Os; average MBR size: 1%



(a) *B*=10

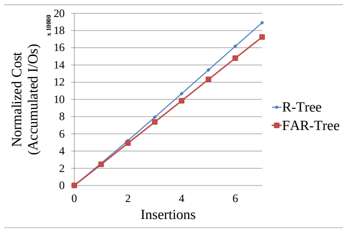


(b) *B*=25

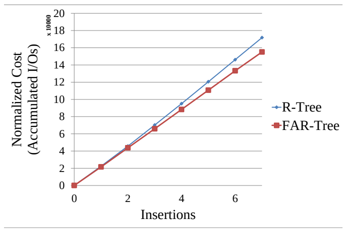


(c) *B*=50

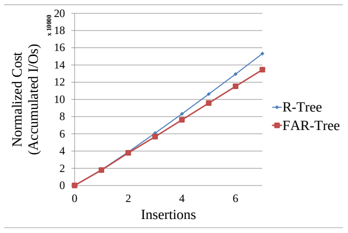
Figure 4.5: Insertion evaluation in logical I/Os on Germany dataset



(a) *B*=10



(b) *B*=25



(c) *B*=50

Figure 4.6: Insertion evaluation in logical I/Os on Greece dataset

insertions.

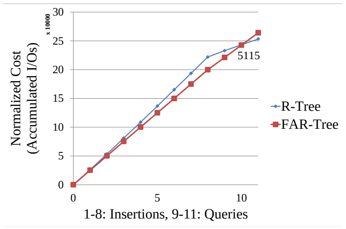
Moreover for the real datasets, the skewness of both the Germany dataset and the Greece dataset should be between the Uniform and the Gaussian, though the Greece is more skewed than the Germany. Accordingly, we can observe from Figure 4.5 and Figure 4.6 that they have similar trend as the Uniform and the Gaussian dataset, as well as a similar susceptibility to the buffer size.

4.2.2 Query Evaluation

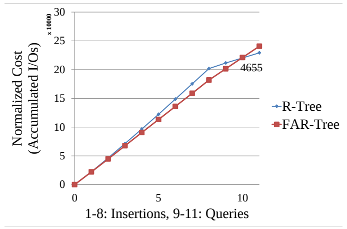
To evaluate the query, we search on the R-Tree and the FAR-Tree after inserting data with the corresponding insertion approaches, and then compare their query costs. We insert the same amount of data as in the insertion evaluation, and then we do queries for the query performance. From this insertion-query workload, we may think of applications where there are data coming all the time and we do queries at the end of the day. In this scenario, we have streaming operations that we do insertions and afterwards we do a bunch of queries; the insertion-query workload can be regarded as a window of such streaming. Figure 4.7, Figure 4.8, and Figure 4.9 show the costs for synthetic datasets. Figure 4.10 and Figure 4.11 show the costs for the Germany dataset and the Greece dataset. Again, the vertical axis represents the normalized accumulated I/O costs and the horizontal axis represents the round of operations, with the insertions at first and the queries followed up. For both the insertions and queries, we divide the operations into rounds, with each round containing 2,500 operations; we note the accumulated I/O costs after a corresponding round in the figures.

For all the cases in the result charts, we can observe a steeper slope for the FAR-Tree query cost than the R-Tree, which means that the FAR-Tree costs more than the R-Tree on doing queries. When doing the insertions, the FAR-Tree has saved an amount of I/O compared to the R-Tree, however, such savings will be used up by the queries at some point. The cross point in the result chart means that the accumulated cost of the FAR-Tree becomes equal to that of the R-Tree at that time, after which more queries will makes the FAR-Tree more expensive than the R-Tree. The number noted by the cross point is the amount of queries the insertion savings can afford. For example, in Figure 4.7a, the FAR-Tree can afford 5,115 queries until it becomes less cost efficient than the R-Tree.

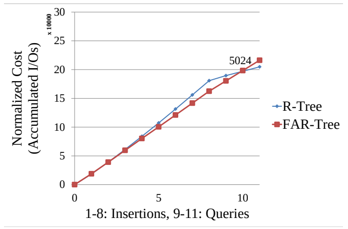
The costs of the FAR-Tree queries is decided by the length of the chains, as a query needs to traverse all the chain nodes attached to a target leaf node. Comparing the results on different distributions, we find that the skewed data costs more for queries. For example in Figure 4.7, the slope for the query cost is getting steeper row by row. Although this can be observed for both the R-Tree and the FAR-Tree, the FAR-Tree shows a very obvious influence. For the instance with size of the buffer being 10 in Figure 4.7, the FAR-Tree queries on the Gaussian dataset cost 2 times of the Uniform dataset, and queries on the Zipf data costs more than 30 times of the Uniform data. In this case, the number of queries that the insertion savings can afford decrease with the growing skewness of the data distribution, which can be observed by comparing within columns (except for the Zipf distribution in Figure 4.9, as the R-Tree is too much influenced by the large overlap that diminishes its performance). The reason is about the length of the chain resulted by different distributions. As discussed in Section 4.2.1, a skewed dataset means concentrated insertions, which will lead to long chains. Since our query data has the same dis-



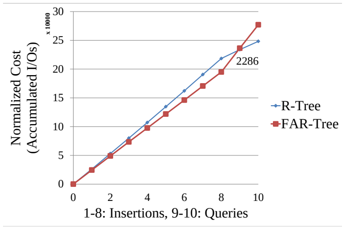
(a) Uniform, *B*=10



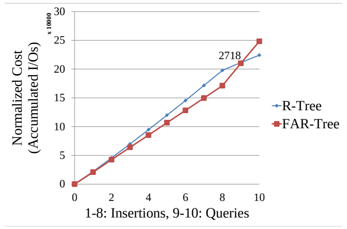
(b) Uniform, *B*=25



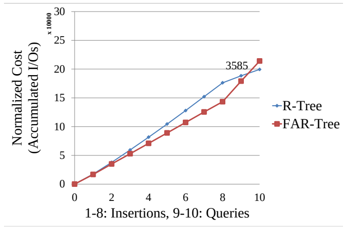
(c) Uniform, *B*=50



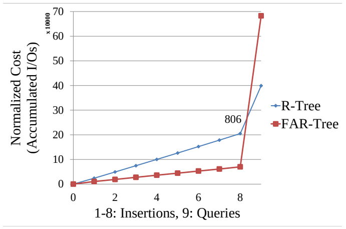
(d) Gaussian, *B*=10



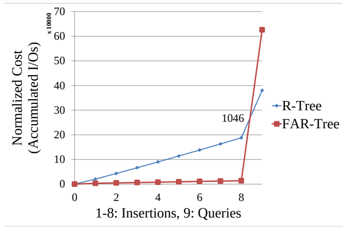
(e) Gaussian, *B*=25



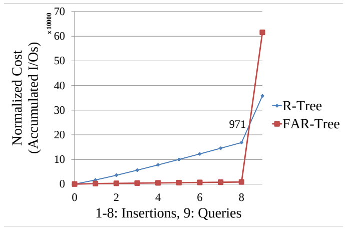
(f) Gaussian, *B*=50



(g) Zipf, *B*=10

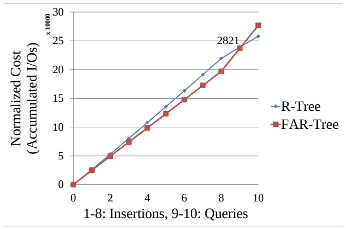


(h) Zipf, *B*=25

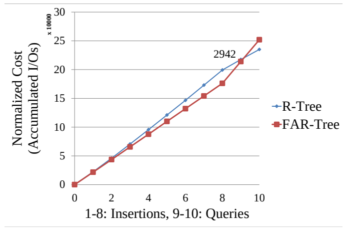


(i) Zipf, *B*=50

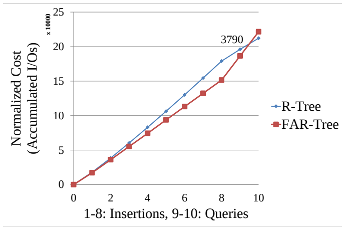
Figure 4.7: Query evaluation in logical I/Os; average MBR size: 0.01%



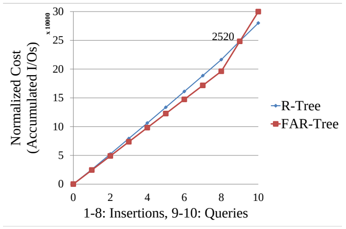
(a) Uniform, *B*=10



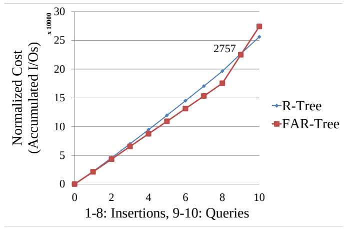
(b) Uniform, *B*=25



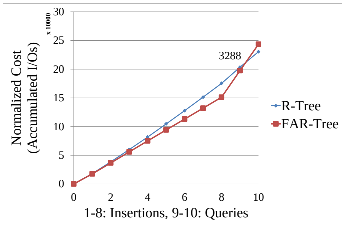
(c) Uniform, *B*=50



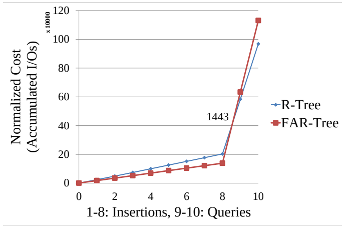
(d) Gaussian, *B*=10



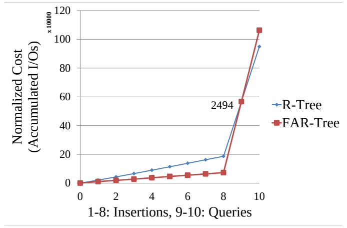
(e) Gaussian, *B*=25



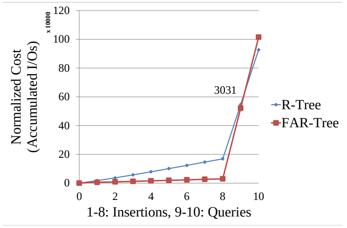
(f) Gaussian, *B*=50



(g) Zipf, *B*=10

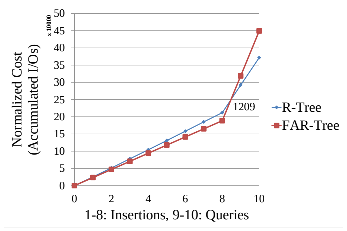


(h) Zipf, *B*=25

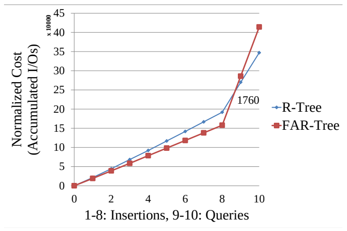


(i) Zipf, *B*=50

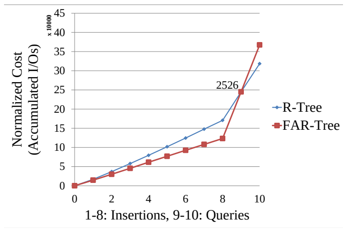
Figure 4.8: Query evaluation in logical I/Os; average MBR size: 0.1%



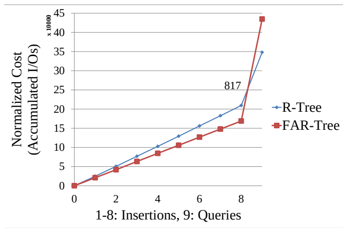
(a) Uniform, *B*=10



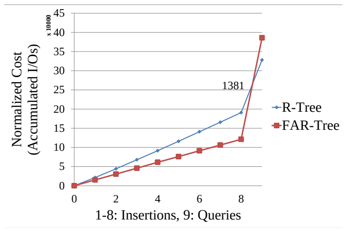
(b) Uniform, *B*=25



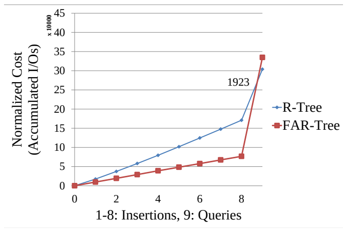
(c) Uniform, *B*=50



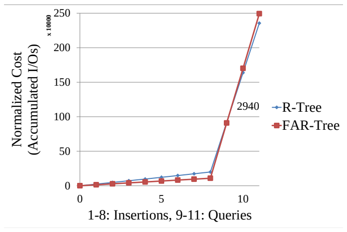
(d) Gaussian, *B*=10



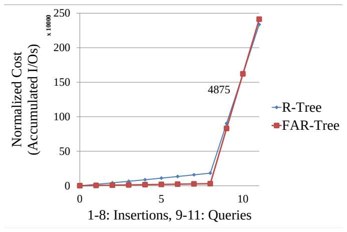
(e) Gaussian, *B*=25



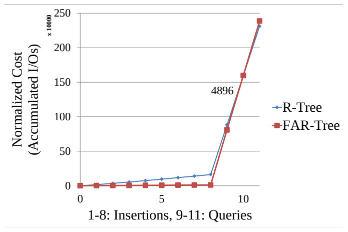
(f) Gaussian, *B*=50



(g) Zipf, *B*=10

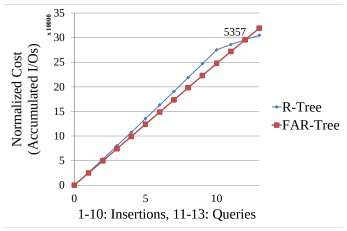


(h) Zipf, *B*=25

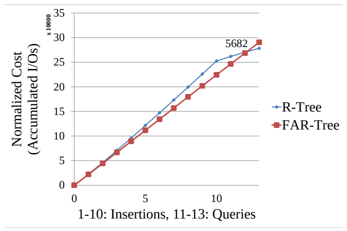


(i) Zipf, *B*=50

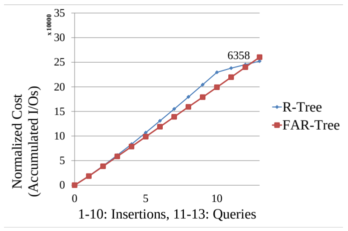
Figure 4.9: Query evaluation in logical I/Os; average MBR size: 1%



(a) *B*=10

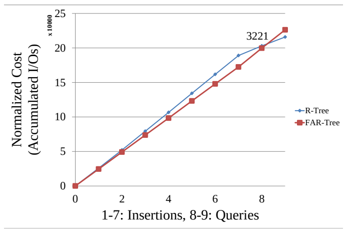


(b) *B*=25

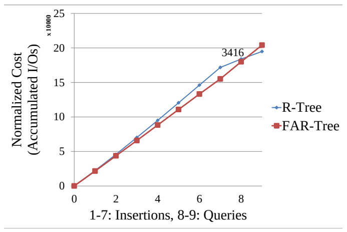


(c) *B*=50

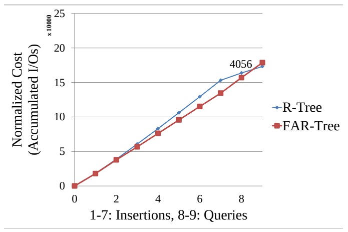
Figure 4.10: Query evaluation in logical I/Os on Germany dataset



(a) *B*=10



(b) *B*=25



(c) *B*=50

Figure 4.11: Query evaluation in logical I/Os on Greece dataset

tribution as the indexed data, the queries are concentrated as well. In this case, for the skewed data, queries will need to read long chains and thus costs large amount of disk reads. With similar idea, if we change the base tree to a bulk-loaded one, we will get less costs, and the reason is that the bulk-loaded base tree tends to have more precisely and evenly distributed leaf nodes, which will result in the tree with shorter chains. We will discuss about the effects of bulk-loading in Section 4.2.3.

If we compare among Figure 4.7, Figure 4.8, and Figure 4.9, we can see that the query costs for both the R-Tree and the FAR-Tree increase as the object size grows. At a glance, we can see that the range of the vertical axis grows from 300,000 to 500,000 and 450,000 for the Uniform and the Gaussian datasets and from 700,000 to 2,500,000 for the Zipf dataset. For the example of the Uniform dataset with B = 10 as shown in Figure 4.7a, Figure 4.8a, and Figure 4.9a, the query costs for doing 5,000 queries on the FAR-Tree of the 0.1% and the 1% object size are 2 and 7 times of the costs on the 0.01% and accordingly the costs for the R-Tree grows to 2 and 8 times. The reason is that, with larger objects, the objects overlap heavily; with more overlap, the query shape may intersect with more subtrees such that more branches need to be searched to finish a query. With the same reason, the costs difference between the R-Tree and the FAR-Tree becomes smaller with increasing overlaps, as the R-Tree query's overhead increases for querying multiple branches. This phenomenon can be more obviously observed in the results of the Zipf distribution.

Due to the influence of the buffer size, we can observe that the number of queries that the insertion savings can afford grows with the increase of the buffer size in general. The influence of buffer size is independent of dataset properties, as the queries are distributed over all branches and the chances of nodes being caught in memory are equal.

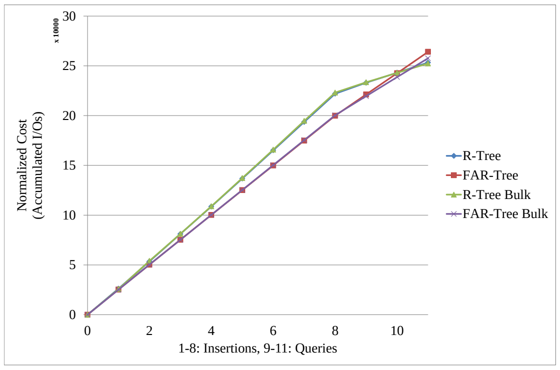
For the real datasets, we can observe that both of them have similar cost trends and similar susceptibility to buffers in between the results of the Uniform and the Gaussian dataset. With the Greece dataset being more skewed than the Germany dataset, the FAR-Tree for the Greece dataset has longer chains and costs more on queries and hence the insertion savings can afford less queries. For example, in the case of buffer in size of 10, the Germany dataset can afford 5,357 queries while the Greece dataset can only afford 3,221 queries.

4.2.3 Bulk-loading's Effect On Insertion And Query

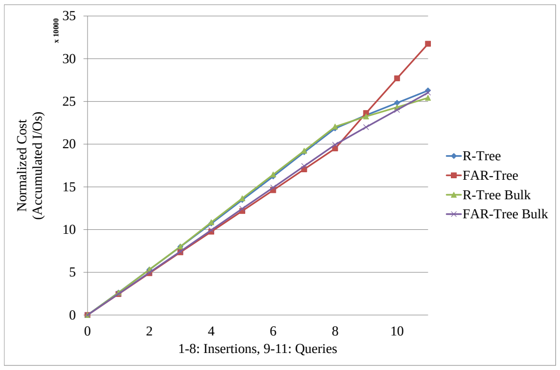
As introduced in Section 2.2.3, constructing an R-Tree by bulk-loading leads to a better index arrangement globally with less overlap. The base trees we use in our experiment are constructed by sequentially inserting the data; we call it the insert-constructed tree. As a comparison, under the case of $B = 10$ with objects in size of 0.01% of the space area, we construct the base tree by bulk-loading with the same data to see its effects on insertions and queries.

As shown in the results of our experiment, bulk-loading the base tree does not have much influence for the R-Tree operations. In Figure 4.12, the costs for the R-Tree and the R-Tree Bulk are almost the same. For the FAR-Tree, however, bulk-loading the base tree leads to a different costs, which is especially obvious on the more skewed data. Firstly, for the insertions, the FAR-Tree on bulk-loaded base tree costs more than that on insert-constructed base tree. As observed in Figure 4.12, the cost of the FAR-Tree Bulk is always higher than the FAR-Tree. Secondly, for the queries, the FAR-Tree on bulk-loaded base tree costs less than the FAR-Tree on insert-constructed base tree. In Figure 4.12, the slope of the FAR-Tree is sharper than the FAR-Tree Bulk; also, the difference is more obvious on the more skewed data.

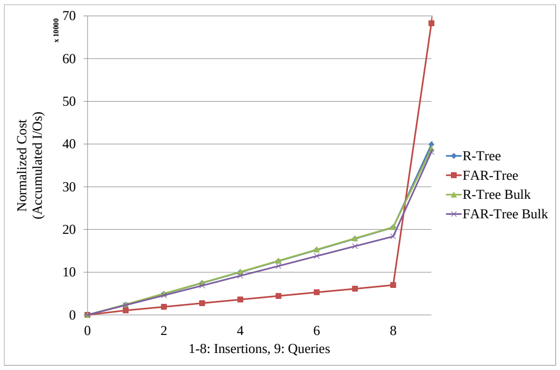
The key reason is that, the FAR-Tree insertion will result in shorter chains on the bulk-loaded base tree for that it is more precisely arranged. The length of the longest chain in the FAR-Tree after the insertions are listed in Table 4.2. With the bulk-loaded base tree, the FAR-Tree will have shorter chains, and the chain length decrement is more obvious on the skewed data. With the decreased chain length, the advantages of the FAR-Tree chain nodes insertion can not be well utilized, so the insertion costs more than that on the insert-constructed base tree which makes longer chains. On the other hand, the query costs decrease as a result of the shorter chains, since there are fewer chain nodes to traverse.



(a) Uniform



(b) Gaussian



(c) Zipf

Figure 4.12: Bulk-loading's effect on insertion and query

Length of the longest chain	Uniform	Gaussian	Zipf
Bulk-loaded base tree	10	7	83
Insert-constructed base tree	14	20	178

Table 4.2: The length of the longest chain in the FAR-Tree

4.2.4 Re-balancing Evaluation

To evaluate the re-balancing, we first insert data to the base tree with the R-Tree and FAR-Tree insertion approaches, then we re-balance the FAR-Tree, and lastly do queries on the R-Tree and the FAR-Tree respectively. For the first step, we insert the same amount of data as in the insertion evaluation. After the FAR-Tree re-balancing, we do 5,000 queries on both indexes.

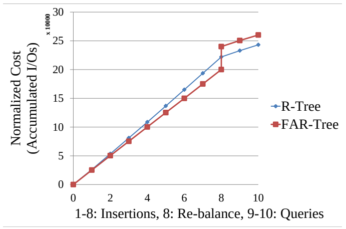
Figure 4.13, Figure 4.14, and Figure 4.15 show the costs for synthetic datasets. Figure 4.16 and Figure 4.17 show the costs for the Germany dataset and the Greece dataset. As before, the vertical axis represents the normalized accumulated I/O costs and the horizontal axis represents the number of operations, with the insertions at first and a re-balancing performed before performing the queries. We do insertions and queries in rounds, with each round containing 2,500 operations; in the figures, we note the accumulated I/O costs after a corresponding round. In the charts, the cost jump in the FAR-Tree is for the FAR-Tree re-balancing.

Essentially, the re-balancing is doing a set of R-tree insertions which re-insert all the entries from the attached chain nodes back to the tree index, so the number of node updates for re-balancing should be almost the same as constructing the corresponding R-Tree. However, as we can see from all the results, the FAR-Tree re-balancing costs less than constructing the R-Tree. For example, in Figure 4.13a, the re-balancing costs about 50,000 I/Os while constructing the R-Tree costs more than 200,000 I/Os. A minor reason for this is that the number of objects re-inserted is smaller than the number of objects inserted in the corresponding R-Tree, since the objects inserted to the leaf nodes in the FAR-Tree does not need to be re-inserted during the re-balancing. The major reason is that the buffer is better utilized because of the “ordered” insertion in the re-balancing. The re-balancing retrieves the attached entries chain by chain. Entries within the same chain are located close to each other, and are close to the entries in their appending leaf nodes as well. For this reason, when inserting these entries back to the tree, successive insertions are usually on the same leaf node or adjacent leaf nodes. In this case, most insertions are executed when the target nodes can be found inside buffer; the buffer hit increases and therefore the number of disk access is reduced. If we set the buffer size to 1 page to simulate the case that there is no buffer utilization for the case in Figure 4.13a, we will find that the re-balancing cost is as large as the R-Tree construction. The comparison is shown in Figure 4.18. The efficient buffer utilization makes the re-balancing cost acceptable. We can observe from results of both the synthetic datasets and the real datasets that, with the increasing buffer size, the cost of re-balancing decreases evidently. The total costs of the FAR-Tree insertion plus the re-balancing can be less than constructing the R-Tree with the same data. For all cases in our experiment, the total cost of FAR-Tree is more than the R-Tree when the buffer size is 10, but as the buffer size grows, the FAR-Tree costs less than the R-Tree in total.

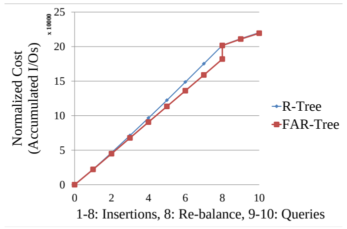
At the point just after a re-balancing, the FAR-Tree actually becomes a balanced R-Tree index. The R-Tree built from the FAR-Tree re-balancing has similar shape as the comparing R-Tree, although they may be slightly different as the order of insertions matters when inserting data to construct an R-Tree. This explains why the costs for the queries after re-balancing are almost parallel. On the other hand, as discussed in Section 3.1.3, the sorted re-insertion results in a better structured R-Tree with less MBR overlap. In this case, although not obviously observed, queries on the re-balanced tree yields a better performance, for the results in Figure 4.13c as an example. This phenomenon can be more obviously observed in the results with real access time measurement in Section 4.3.

4.3 Results In Real Disk Access Time

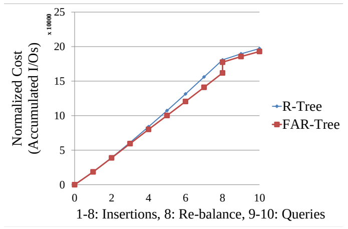
For all the experiments above, we run them on a real SSD and measure their real disk access time. This section lists all the corresponding results, and we can observe that the results generally agree with the results measured in logical I/Os.



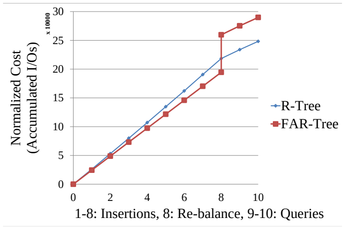
(a) Uniform, *B*=10



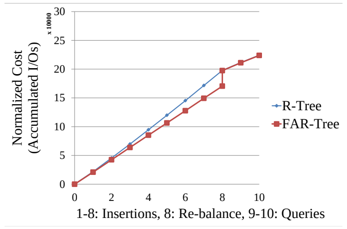
(b) Uniform, *B*=25



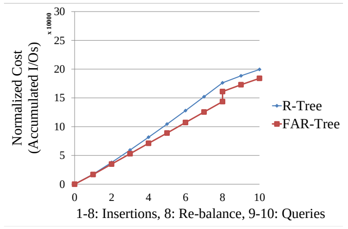
(c) Uniform, *B*=50



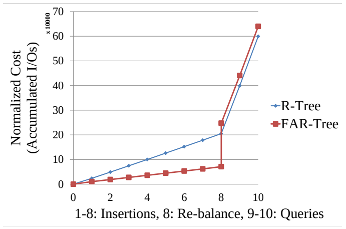
(d) Gaussian, *B*=10



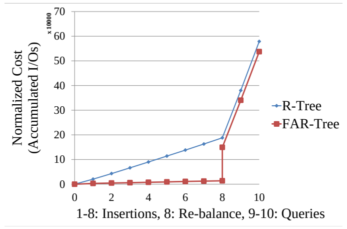
(e) Gaussian, *B*=25



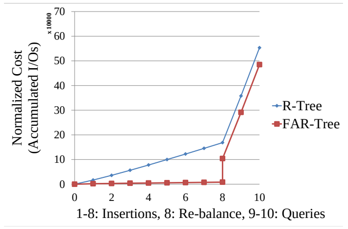
(f) Gaussian, *B*=50



(g) Zipf, *B*=10

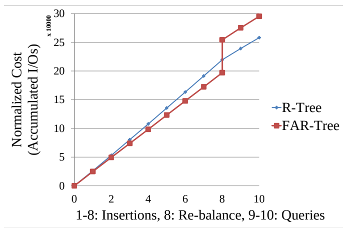


(h) Zipf, *B*=25

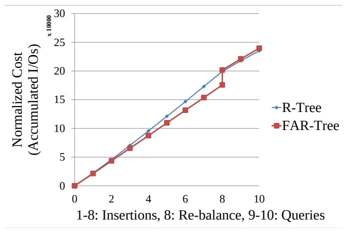


(i) Zipf, *B*=50

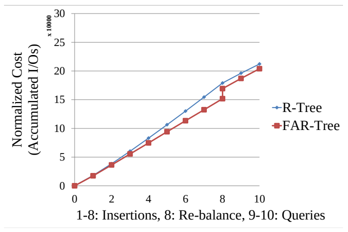
Figure 4.13: Re-balancing evaluation in logical I/Os; average MBR size: 0.01%



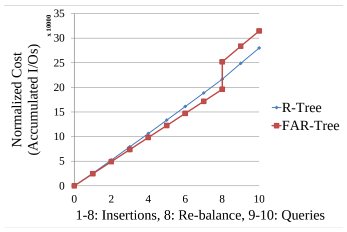
(a) Uniform, *B*=10



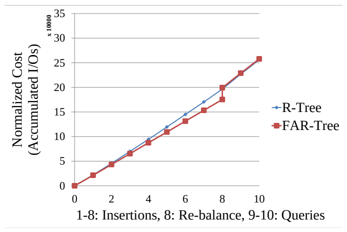
(b) Uniform, *B*=25



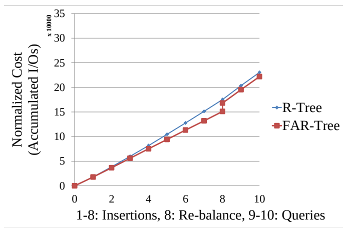
(c) Uniform, *B*=50



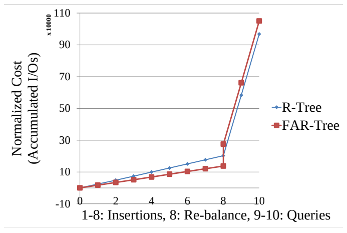
(d) Gaussian, *B*=10



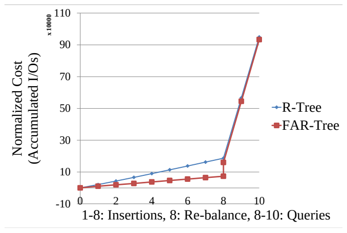
(e) Gaussian, *B*=25



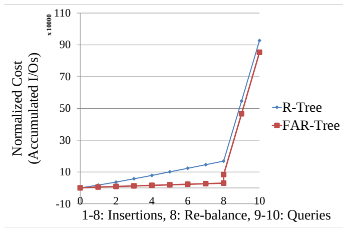
(f) Gaussian, *B*=50



(g) Zipf, *B*=10

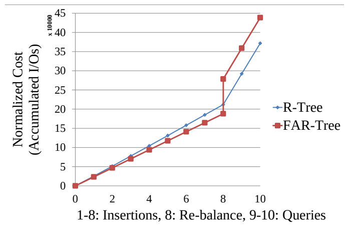


(h) Zipf, *B*=25

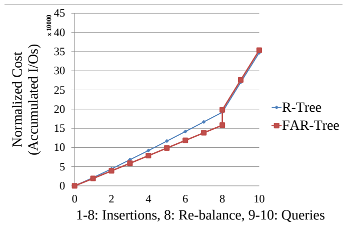


(i) Zipf, *B*=50

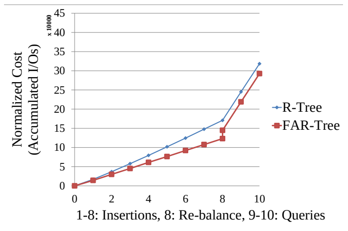
Figure 4.14: Re-balancing evaluation in logical I/Os; average MBR size: 0.1%



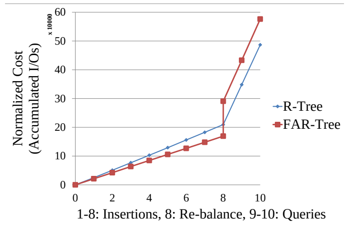
(a) Uniform, *B*=10



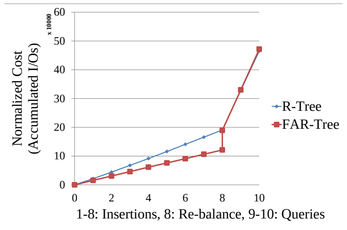
(b) Uniform, *B*=25



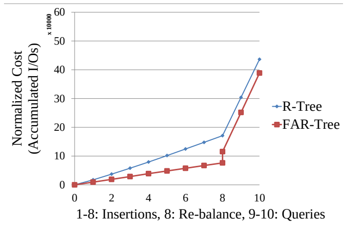
(c) Uniform, *B*=50



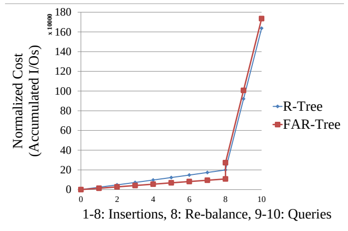
(d) Gaussian, *B*=10



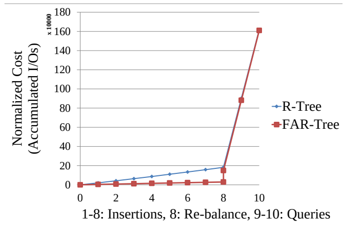
(e) Gaussian, *B*=25



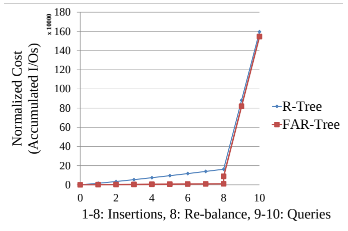
(f) Gaussian, *B*=50



(g) Zipf, *B*=10

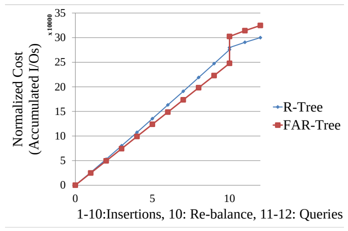


(h) Zipf, *B*=25

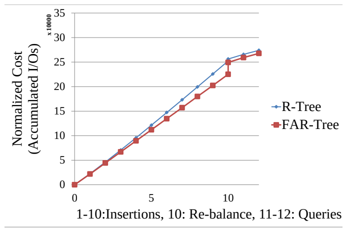


(i) Zipf, *B*=50

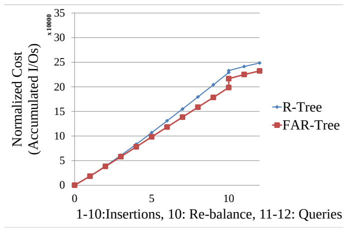
Figure 4.15: Re-balancing evaluation in logical I/Os; average MBR size: 1%



(a) *B*=10

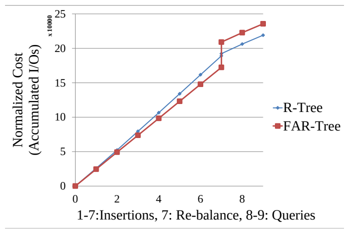


(b) *B*=25

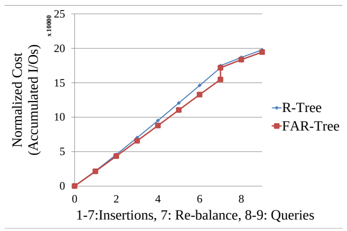


(c) *B*=50

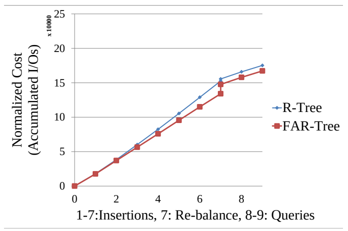
Figure 4.16: Re-balancing evaluation in logical I/Os on Germany dataset



(a) *B*=10

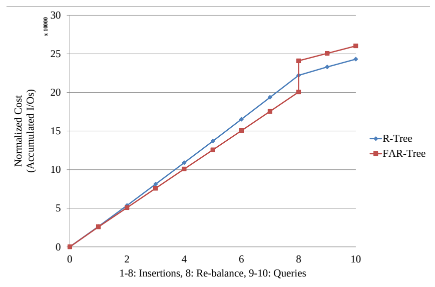


(b) *B*=25

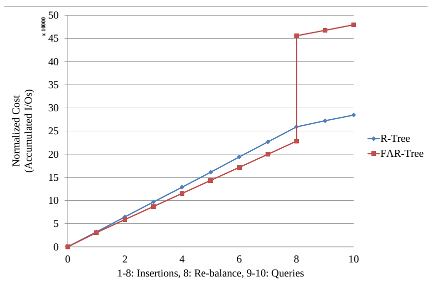


(c) *B*=50

Figure 4.17: Re-balancing evaluation in logical I/Os on Greece dataset



(a) *B*=10



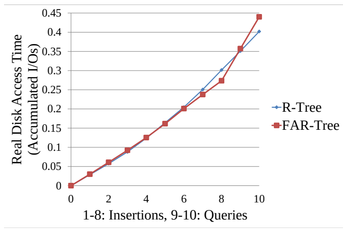
(b) *B*=1

Figure 4.18: The result in Figure 4.13a with a 10-page buffer size is zoomed in on the left. The corresponding result with a 1-page buffer size is shown on the right.

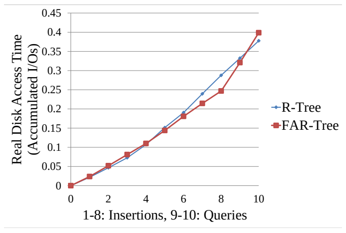
Figure 4.19, Figure 4.20, and Figure 4.21 show the costs of insertions and queries for the synthetic datasets. Figure 4.22 and Figure 4.23 show the results for the Germany dataset and the Greece dataset respectively. We display the insertions and queries together for simplicity.

For the insertion, the FAR-Tree costs less than the R-Tree. The saving is more obvious with more skewed distribution or larger buffer size. For the query, the FAR-Tree costs more than the R-Tree because of the chain nodes traversal, so that the savings from insertions are offset by the queries. The more skewed the dataset distribution is, the more costs will be spent by the FAR-Tree queries. However, an exception is shown in Figure 4.20 and Figure 4.21 that for the results of the Zipf distribution, where the R-Tree query costs more than the FAR-Tree. This is because that the R-Tree cannot work well with intense overlap on the datasets with large objects. In the logical I/O measurement, the number of queries afforded by the insertion savings also increases with the enlarging object size for the Zipf distribution, although on the real disk it is more obvious.

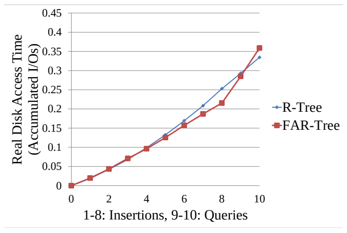
Figure 4.24, Figure 4.25, and Figure 4.26 show the costs of re-balancing for the synthetic data. Figure 4.27 and Figure 4.28 show the results for the Germany dataset and the Greece dataset respectively. According to the results in logical I/Os, the cost for re-balancing decreases with the growth of the buffer size, and the total cost of insertions plus queries become less than constructing the comparing R-Tree when the buffer is relatively large. In addition, we can observe that the query on the re-balanced FAR-Tree costs less than the comparing R-Tree due to the benefits



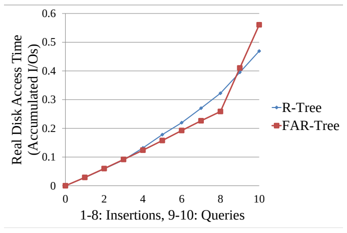
(a) Uniform, *B*=10



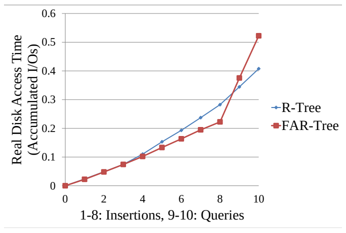
(b) Uniform, *B*=25



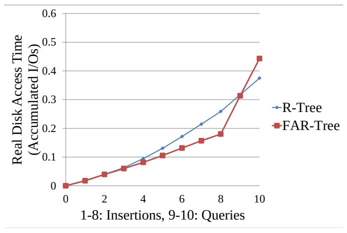
(c) Uniform, *B*=50



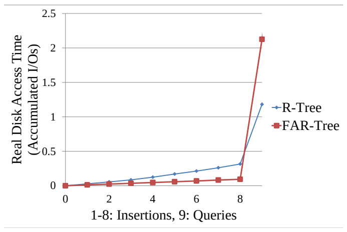
(d) Gaussian, *B*=10



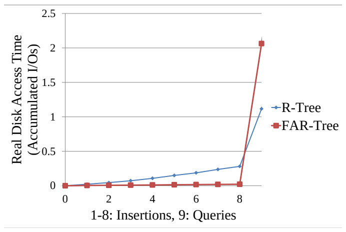
(e) Gaussian, *B*=25



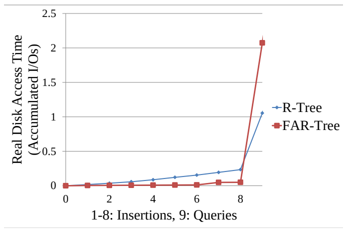
(f) Gaussian, *B*=50



(g) Zipf, *B*=10



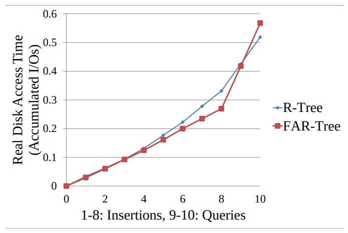
(h) Zipf, *B*=25



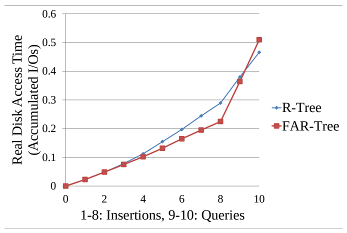
(i) Zipf, *B*=50

Figure 4.19: Insertion and Query evaluation in real disk access time; average MBR size: 0.01%

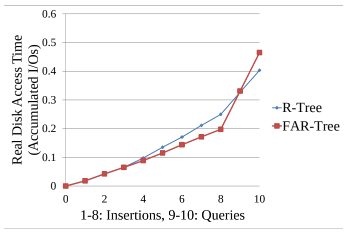
of sorted re-insertion. The results in the real disk access time is more obvious than the logical number of I/Os. A possible explanation is that the re-balancing re-insert the chain entries in a "ordered" sequence, such that adjacent nodes (adjacent leaf nodes and nodes on the same branch) tend to be stored on the disk in sequence. In the case when one query reads more than one branch, some reads can be sequential. The rate of sequential read and the random read on SSDs are different. The SSD we use has the sequential disk read in 500MB/s and the random read in 29,000IOPS (115MB/s). The logical measurement cannot reveal this subtle difference.



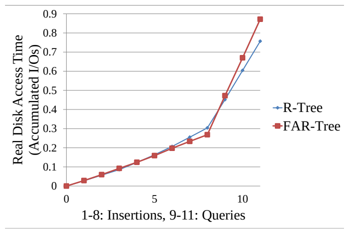
(a) Uniform, *B*=10



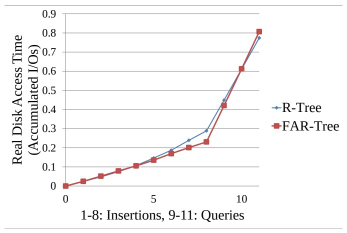
(b) Uniform, *B*=25



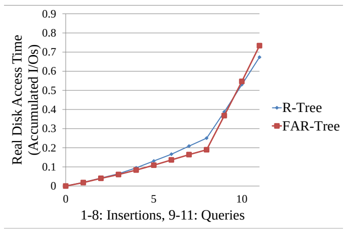
(c) Uniform, *B*=50



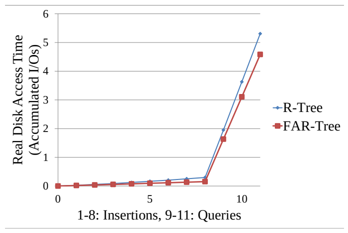
(d) Gaussian, *B*=10



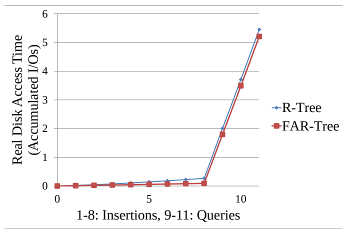
(e) Gaussian, *B*=25



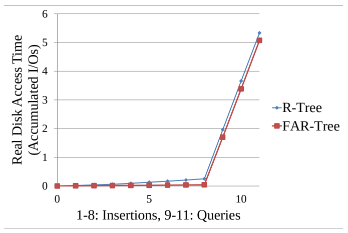
(f) Gaussian, *B*=50



(g) Zipf, *B*=10

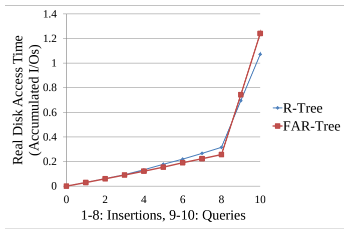


(h) Zipf, *B*=25

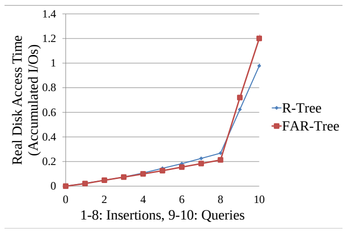


(i) Zipf, *B*=50

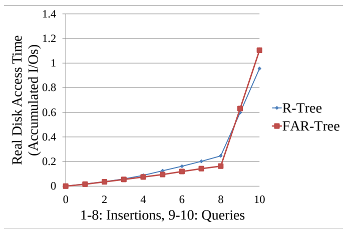
Figure 4.20: Insertion and Query evaluation in real disk access time; average MBR size: 0.1%



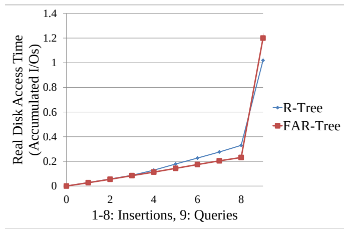
(a) Uniform, *B*=10



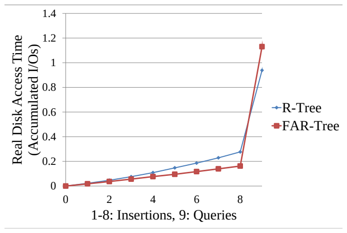
(b) Uniform, *B*=25



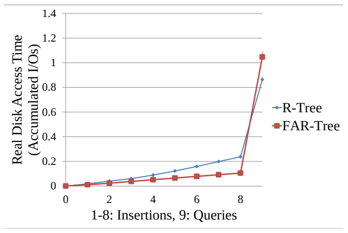
(c) Uniform, *B*=50



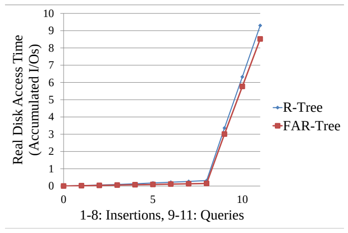
(d) Gaussian, *B*=10



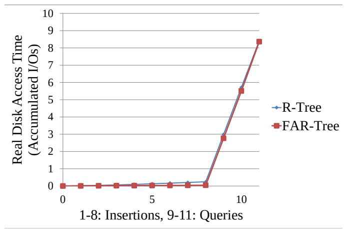
(e) Gaussian, *B*=25



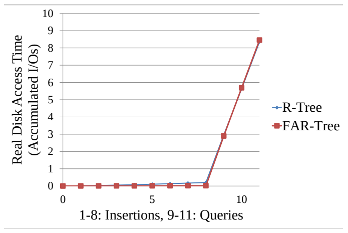
(f) Gaussian, *B*=50



(g) Zipf, *B*=10

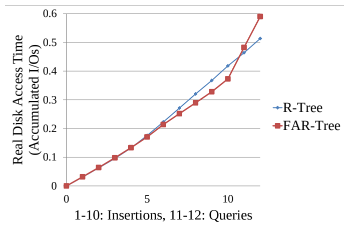


(h) Zipf, *B*=25

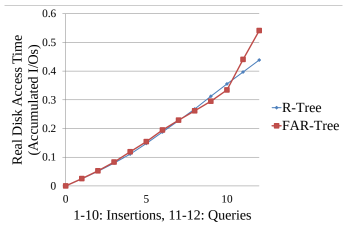


(i) Zipf, *B*=50

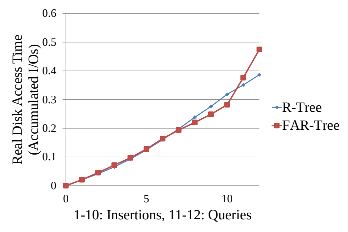
Figure 4.21: Insertion and Query evaluation in real disk access time; average MBR size: 1%



(a) *B*=10

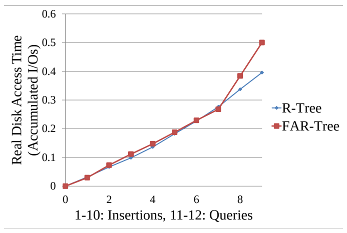


(b) *B*=25

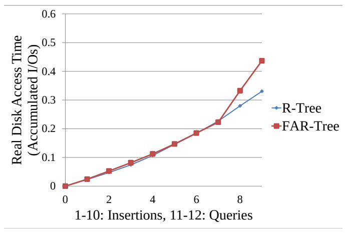


(c) *B*=50

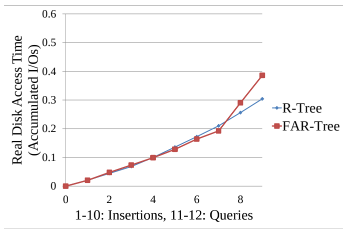
Figure 4.22: Insertion and Query evaluation in real disk access time on Germany Dataset



(a) *B*=10

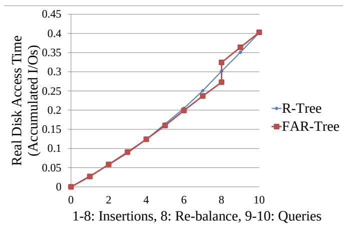


(b) *B*=25

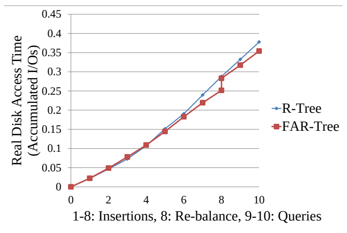


(c) *B*=50

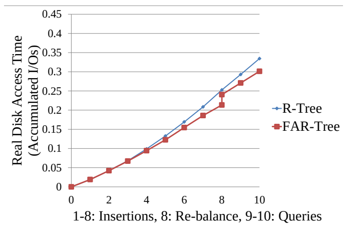
Figure 4.23: Insertion and Query evaluation in real disk access time on Greece Dataset



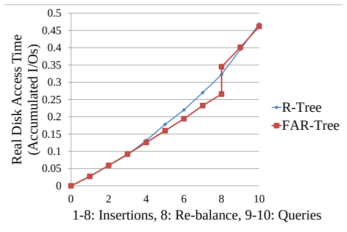
(a) Uniform, *B*=10



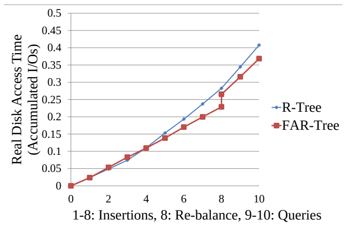
(b) Uniform, *B*=25



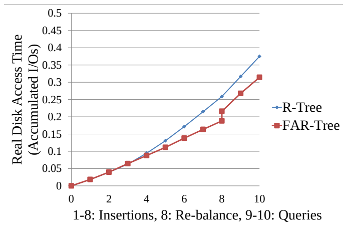
(c) Uniform, *B*=50



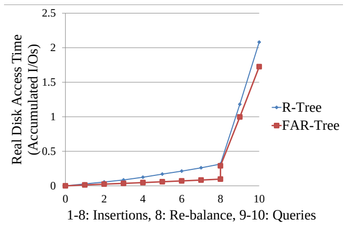
(d) Gaussian, *B*=10



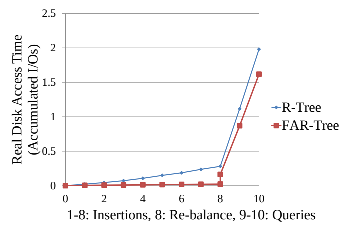
(e) Gaussian, *B*=25



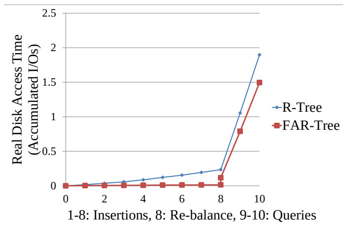
(f) Gaussian, *B*=50



(g) Zipf, *B*=10

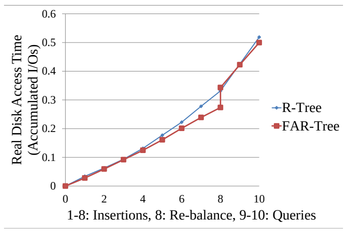


(h) Zipf, *B*=25

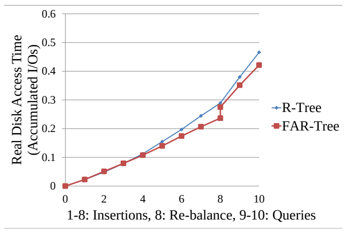


(i) Zipf, *B*=50

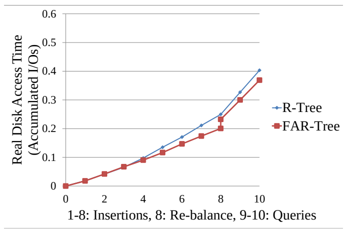
Figure 4.24: Re-balancing evaluation in real disk access time; average MBR size: 0.01%



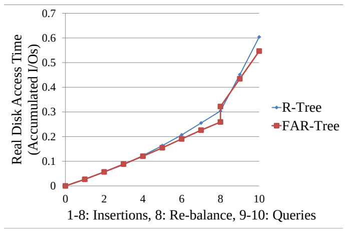
(a) Uniform, *B*=10



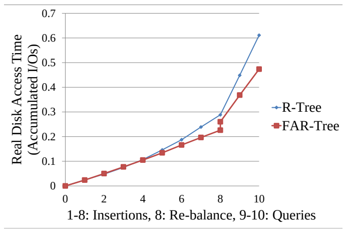
(b) Uniform, *B*=25



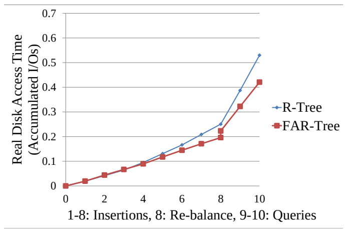
(c) Uniform, *B*=50



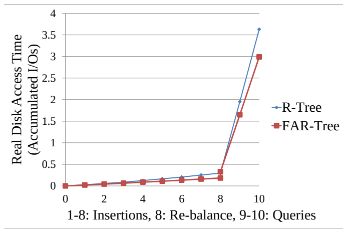
(d) Gaussian, *B*=10



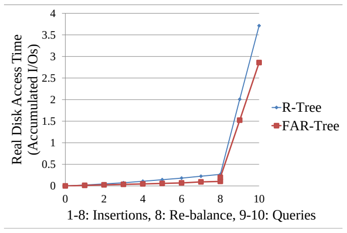
(e) Gaussian, *B*=25



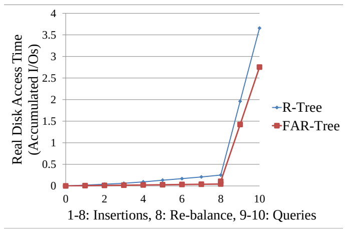
(f) Gaussian, *B*=50



(g) Zipf, *B*=10

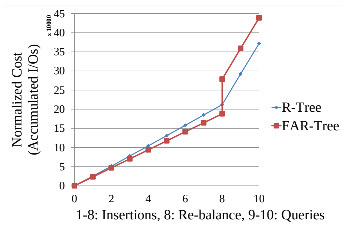


(h) Zipf, *B*=25

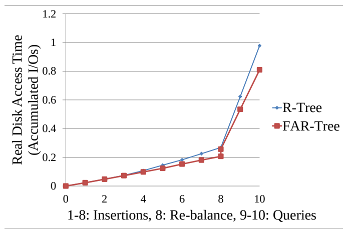


(i) Zipf, *B*=50

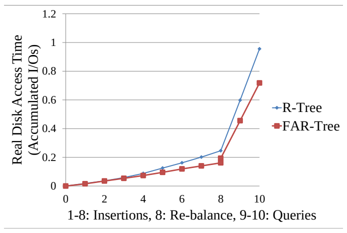
Figure 4.25: Re-balancing evaluation in real disk access time; average MBR size: 0.1%



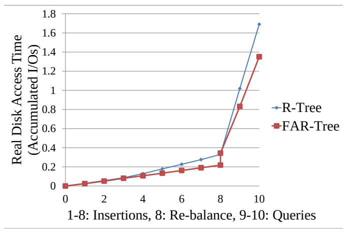
(a) Uniform, *B*=10



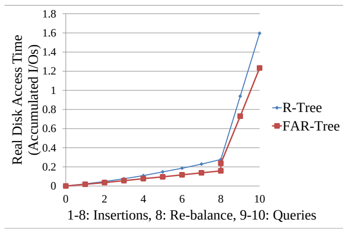
(b) Uniform, *B*=25



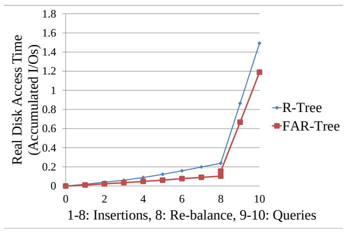
(c) Uniform, *B*=50



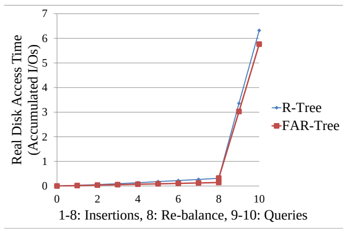
(d) Gaussian, *B*=10



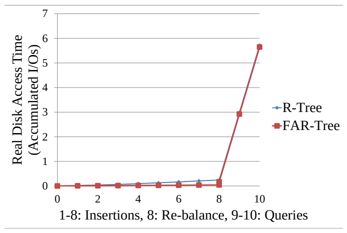
(e) Gaussian, *B*=25



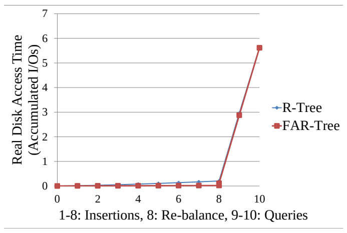
(f) Gaussian, *B*=50



(g) Zipf, *B*=10

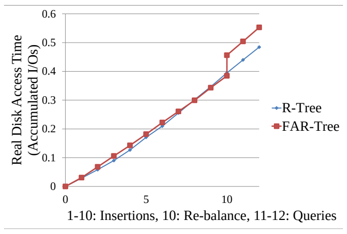


(h) Zipf, *B*=25

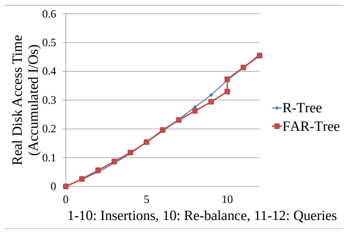


(i) Zipf, *B*=50

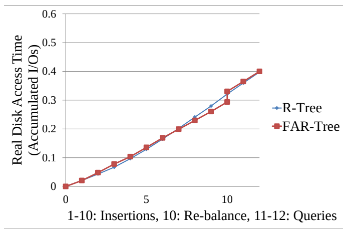
Figure 4.26: Re-balancing evaluation in real disk access time; average MBR size: 1%



(a) *B*=10

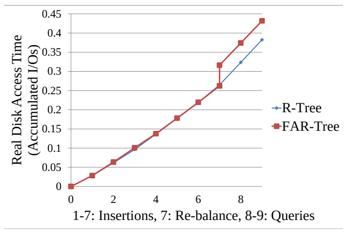


(b) *B*=25

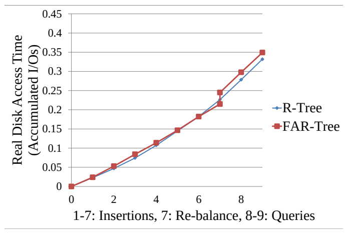


(c) *B*=50

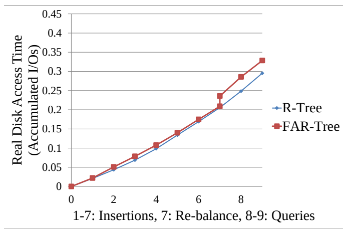
Figure 4.27: Re-balancing evaluation in real disk access time on Germany Dataset



(a) *B*=10



(b) *B*=25



(c) *B*=50

Figure 4.28: Re-balancing evaluation in real disk access time on Greece Dataset

Chapter 5

Conclusion and Future Work

5.1 Conclusion

As a new storage media, SSDs are featured with faster I/O performance and lower power consumption. Considering the affordable price and the increasingly available storage space, SSDs now become a feasible alternative to HDDs for database storage. In this research, we focused on the spatial data storage and presented a flash aware R-Tree based solution for indexing spatial data on SSDs.

The R-Tree is a dynamically constructed and maintained index structure. It was originally designed for HDDs. Generally, HDDs have the same rate for doing reads and writes, and both are relatively slow because of the seek time and rotational latency. SSDs are not well suited for implementing the R-Tree efficiently because of the different hardware mechanisms compared to magnetic disks. SSDs act asymmetrically on reading and writing; writing on SSDs is expensive because of the I/O asymmetry, the garbage collection, and the wear leveling. When implementing an R-Tree on SSDs, it becomes important to consider the index update, as it will lead to large amount of expensive disk writes, and the overall index performance may be impacted in the presence of updates.

In this thesis, we adapted the R-Tree to efficiently work for SSDs. The goal was to minimize the number of disk writes during index update. Based on an unbalanced indexing idea in [10], we proposed our solution to append chains to the leaf nodes and place the new inserting data into the chain nodes, instead of triggering node splits as in a typical R-Tree. By avoiding the splits, we reduced a number of node writes. At the same time, an in-memory table was maintained for MBR adjustment upon chain node insertions, such that the disk access for adjusting the index was eliminated. Compared to the original R-Tree, the FAR-Tree saved I/Os for inserting new objects to the index. On the other hand, the chains may result in more disk reads when searching the index, as the chain nodes need to be traversed; the reads increase as the chains get long. In this case, we proposed to re-balance the FAR-Tree index. The re-balancing is to re-insert the attached entries to the upper R-Tree index sequentially. The FAR-Tree re-balancing utilized the buffer well because of the "ordered" re-inserting, and therefore the overhead of re-balancing was diminished.

We conducted a detailed cost analysis for the FAR-Tree insertions and queries theoretically, with a comparison to the R-Tree. We then evaluated the FAR-Tree performance by doing experiments on both the synthetic data with various properties and the real-world data. We measured the logical I/Os in our experiments, which is the number of node reads and writes. The results conformed with our cost analysis. Experiments showed that, the FAR-Tree insertions always cost less than the R-Tree insertions. The FAR-Tree queries cost more than the R-Tree, and thus the insertion savings were offset after doing a number of queries. For the re-balancing, experiments showed that, with a proper buffer size, the total costs of insertions plus re-balancing for the FAR-Tree can be smaller than constructing the corresponding R-Tree, in which case the overhead for the FAR-Tree re-balancing was acceptable. In addition, we ran our experiments on a real SSD and measured the real disk access time. The results of real access time agreed with those of logical measurement, which showed savings on inserting new data and an acceptable re-balancing overhead, though more costs on queries.

In summary, the FAR-Tree was shown as an update-efficient index at the cost of some overhead at query time. The re-balancing addressed the FAR-Tree's problem of having larger costs for doing queries, and the re-balancing overhead is acceptable. When implemented on SSDs, compared to the R-Tree, the FAR-Tree improves the I/O performance in the presence of index updates. For an intensely updating index, the FAR-Tree will yield a good performance. Moreover, compared to previous approaches for indexing data on SSDs, as we reviewed in Chapter 2, our approach does not change the hardware configuration and does not require special care for buffer management.

5.2 Future Work

The FAR-Tree approach gave an efficient solution for adapting the R-Tree spatial index to work on SSDs. It improves the I/O performance when the R-Tree is relatively update-intensive. However, there are still more questions to be answered for the FAR-Tree.

In the FAR-Tree approach, we need to re-balance the tree when the chains get long. In our experiments, we re-balanced the index after we inserted all the insertion data, and thus the re-balancing happens in cases with various length of chains (depending on the dataset distributions). The results for all cases yielded a small cost for re-balancing when adopting a proper buffer size, such that the costs for insertions plus re-balancing is smaller than the R-Tree. However, there may be a optimal point to re-balance so that it will yield better savings. Especially when we do queries and insertions together, it is important to know when to re-balance the tree for a minimum overall cost. One could think about a dynamic strategy to find a threshold for the chain length; when the length of the chains reach the threshold, the tree will re-balance by itself. This could be done by tracking the costs history of the index and statistically choosing a threshold with best performance.

In addition, a good cost estimation model can be an aid to determine the threshold. We have given a detailed analysis for insertions and queries, however, a more precise estimation model is expected so that we can estimate the best time for re-balancing. Also, the costs for operations are expected to be estimated in advance only by referring to the properties of the indexing data and the query data.

Moreover, the FAR-Tree approach did not consider the buffer management, as in our experiments, we simply adopted the random eviction strategy. However, a better eviction algorithm and buffer management could help to make more savings.

Another direction for the future work is to reduce the number of reads resulted from the chain node traversal when doing queries. To address this problem, a smarter buffer manager can be adopted. On the other hand, a sub-index could be constructed on the chains to avoid the sequential traversal. Based on the idea of the VA-File [24], which calculates a compact binary approximations for spatial objects. One could calculate approximations for the chain nodes' locations and utilize the approximations to filter out an amount of chain nodes instead of traversing them all.

Bibliography

[1] Devesh Agrawal, Deepak Ganesan, Ramesh Sitaraman, Yanlei Diao, and Shashi Singh. Lazy-Adaptive Tree: an optimized index structure for flash devices. *Proc. VLDB Endow.*, 2(1):361–372, August 2009.

[2] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The R*-tree: an efficient and robust access method for points and rectangles. In *Proceedings of the 1990 ACM SIGMOD international conference on Management of data*, SIGMOD '90, pages 322–331, New York, NY, USA, 1990. ACM.

[3] Tobias Emrich, Franz Graf, Hans-Peter Kriegel, Matthias Schubert, and Marisa Thoma. On the impact of flash SSDs on spatial indexing. In *Proceedings of the Sixth International Workshop on Data Management on New Hardware*, DaMoN '10, pages 3–8, New York, NY, USA, 2010. ACM.

[4] Antonin Guttman. R-trees: a dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, SIGMOD '84, pages 47–57, New York, NY, USA, 1984. ACM.

[5] Ibrahim Kamel and Christos Faloutsos. On packing R-trees. In *Proceedings of the second international conference on Information and knowledge management*, CIKM '93, pages 490–499, New York, NY, USA, 1993. ACM.

[6] Ibrahim Kamel and Christos Faloutsos. Hilbert R-tree: An improved R-tree using fractals. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, pages 500–509, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[7] Vamsee Kasavajhala. Solid state drive vs. hard disk drive price and performance study. Dell Technical White Paper, Dell Power Vault Storage Systems, May 2011.

[8] Hyojun Kim and Seongjun Ahn. BPLRU: a buffer management scheme for improving random writes in flash storage. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, FAST'08, pages 16:1–16:14, Berkeley, CA, USA, 2008. USENIX Association.

[9] Ioannis Koltsidas and Stratis D. Viglas. Flashing up the storage layer. *Proc. VLDB Endow.*, 1(1):514–525, August 2008.

[10] Ioannis Koltsidas and Stratis D. Viglas. Data management over flash memory. In *SIGMOD Conference*, pages 1209–1212, 2011.

[11] Ioannis Koltsidas and Stratis D. Viglas. Spatial data management over flash memory. In *Advances in Spatial and Temporal Databases*, volume 6849 of *Lecture Notes in Computer Science*, pages 449–453. Springer Berlin Heidelberg, 2011.

[12] Scott T. Leutenegger, J. M. Edgington, and Mario A. Lopez. STR: A simple and efficient algorithm for R-Tree packing. In *Proceedings of the Thirteenth International Conference on Data Engineering*, ICDE '97, pages 497–506, Washington, DC, USA, 1997. IEEE Computer Society.

[13] Adam Leventhal. Flash storage memory. *Commun. ACM*, 51(7):47–51, July 2008.

[14] Yinan Li, Bingsheng He, Robin Jun Yang, Qiong Luo, and Ke Yi. Tree indexing on solid state drives. *Proc. VLDB Endow.*, 3(1-2):1195–1206, September 2010.

[15] Mark Moshayedi and Patrick Wilkison. Enterprise SSDs. *Queue*, 6(4):32–39, July 2008.

[16] Suman Nath and Aman Kansal. FlashDB: dynamic self-tuning database for NAND flash. In *Proceedings of the 6th international conference on Information processing in sensor networks*, IPSN '07, pages 410–419, New York, NY, USA, 2007. ACM.

[17] Seon-yeong Park, Dawoon Jung, Jeong-uk Kang, Jin-soo Kim, and Joonwon Lee. CFLRU: a replacement algorithm for flash memory. In *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, CASES '06, pages 234–241, New York, NY, USA, 2006. ACM.

[18] Nick Roussopoulos and Daniel Leifker. Direct spatial search on pictorial databases using packed R-trees. In *Proceedings of the 1985 ACM SIGMOD international conference on Management of data*, SIGMOD '85, pages 17–31, New York, NY, USA, 1985. ACM.

[19] Mohamed Sarwat, Mohamed F. Mokbel, Xun Zhou, and Suman Nath. FAST: a generic framework for flash-aware spatial trees. In *Proceedings of the 12th international conference on Advances in spatial and temporal databases*, SSTD'11, pages 149–167, Berlin, Heidelberg, 2011. Springer-Verlag.

[20] Gokul Soundararajan, Vijayan Prabhakaran, Mahesh Balakrishnan, and Ted Wobber. Extending SSD lifetimes with disk-based write caches. In *Proceedings of the 8th USENIX conference on File and storage technologies*, FAST'10, pages 8–8, Berkeley, CA, USA, 2010. USENIX Association.

[21] Radu Stoica, Manos Athanassoulis, Ryan Johnson, and Anastasia Ailamaki. Evaluating and repairing write performance on flash devices. In *DaMoN*, pages 9–14, 2009.

[22] Yannis Theodoridis and Timos Sellis. A model for the prediction of R-tree performance. In *Proceedings of the fifteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, PODS '96, pages 161–171, New York, NY, USA, 1996. ACM.

[23] Stratis D. Viglas. Adapting the B^{+}-tree for asymmetric i/o. In *Proceedings of the 16th East European conference on Advances in Databases and Information Systems*, ADBIS'12, pages 399–412, Berlin, Heidelberg, 2012. Springer-Verlag.

[24] Roger Weber and Stephen Blott. An Approximation-Based Data Structure for Similarity Search. Technical Report Zurich, Switzerland, 1997.

[25] Chin-Hsien Wu, Li-Pin Chang, and Tei-Wei Kuo. An efficient R-tree implementation over flash-memory storage systems. In *Proceedings of the 11th ACM international symposium on Advances in geographic information systems*, GIS '03, pages 17–24, New York, NY, USA, 2003. ACM.