Tree-based indexing

Team 02 - StudQL *

Mohamed Babana, Olivier Boivin, Guillaume Dupont
Yann Kervella, Maxime Lepeytre, Matthieu Lormeau
Soumaya Sabry, Alexandre Taffet, Alexandre Zajac

August 9, 2021

***Abstract*– Nowadays, the research on indexing data structures has reached a high peak. Due to the growth of data, multiple methods and techniques were invented or improved to satisfy the need for larger queries in a limited time. In this work, we present a fundamental type of data structure: *tree-indexing*. In particular, we will compare and implement B-trees and B+-Trees as well as R-trees.**

1 Introduction

During the last few decades, data volumes have skyrocketed ; as we can see on the graph below, data generated during the last two years is more than the entire human history before that. A lot of big industries using big data (Security[19], Healthcare[**?**], Banking, Media[3], Retail, Construction). Big data[18] solves a lot of issues, but for this it is necessary to store and easily access the data.

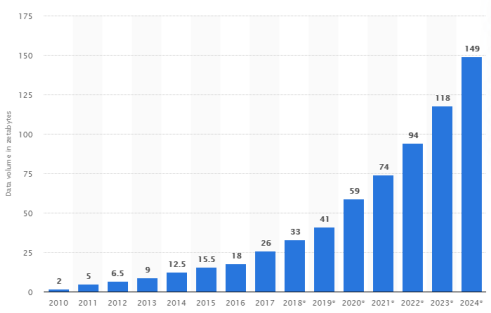
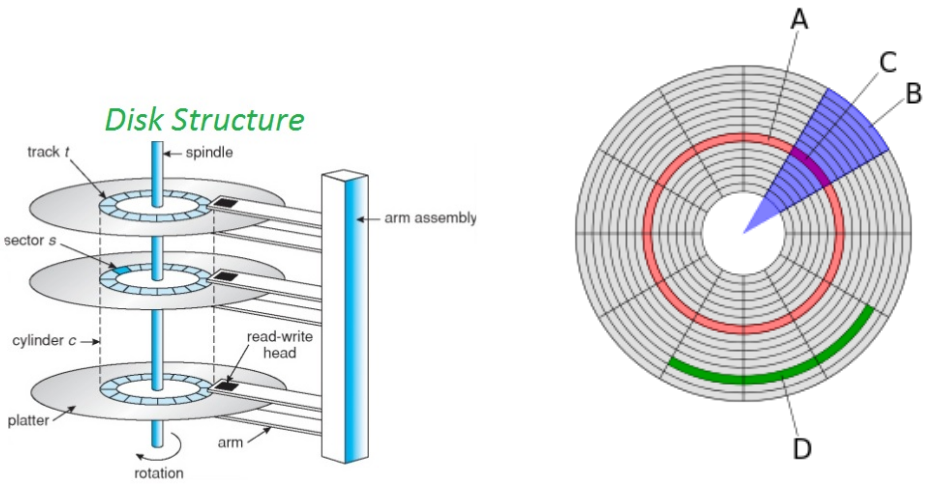


Figure 1: Volume of data/information created worldwide (Historical and Forecast)[1]

The data is converted into bytes which are stored in blocks. The blocks are concatenated into a track and stored on the disk. Each time we want to access a byte which is in a block, we have to read the whole block. After storing the data, now it is important to access quickly and easily data:
$access\,time = seek\,time + rotational\,delay + transfer\,time$

Indexing is a data structure technique which aims to improve the speed of data retrieval. It can be viewed as a table with two columns, the first column takes care of primary or candidate key of a table while the second column is made of pointers holding the location of the disk block where the key is stored.

*Here is the available organization on Github `https://github.com/StudQL`
[1]This figure was taken from `https://www.statista.com/statistics/871513/worldwide-data-created/`



(a) Disk structure (b) Track & block

Figure 2: Taken from dbmsinternals website

1. **Input** : Search Key
2. **Output** : Collection of matching records

Databases have been evolving exponentially for the last 5 decades, considering the Big Data evolution as well, we needed to think of ways to store all the information. From hierarchical databases in the 1960s to NoSQL nowadays, databases have received a lot of improvements throughout the years. Here is a quick chronological reminder of databases evolution :

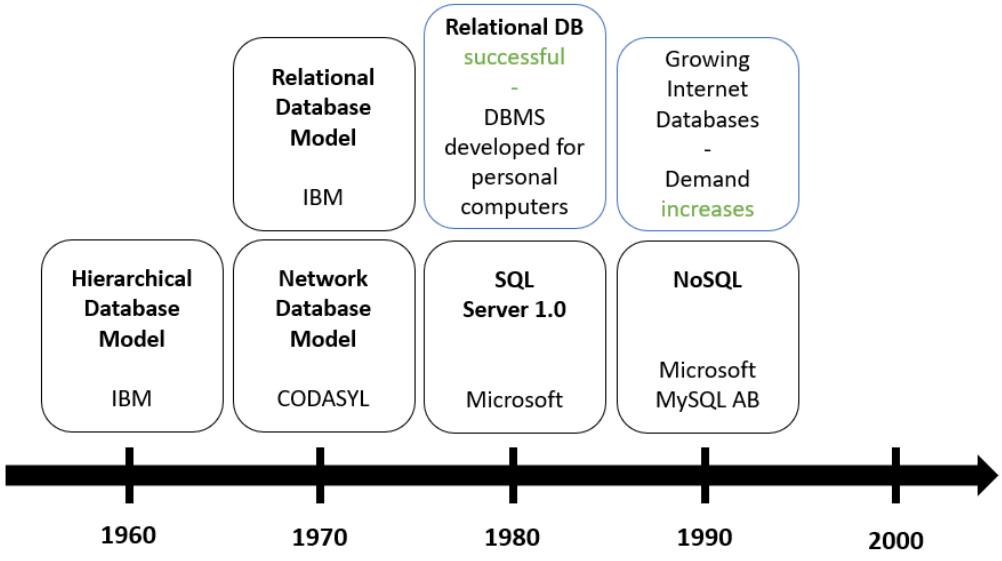


Figure 3: Database systems Timeline

At the beginning, databases came as a solution to the problem of storing information, sometimes publicly or privately, and the delivery of that information at a decent speed. Over time, their impact on society became

more and more important, and we interact with them throughout any application. Fields like health, security, technology couldn't have had such innovations over the last decades without Big Data and Databases.

2 State of art

A **Binary Search Tree** [14] is one of the most simple and easy to understand tree structure and is the foundation of tree indexing data structures. This tree is made of nodes that contain one key value only and has 2 children. The BST is an ordered tree, the left child of parent node has a smaller key value while the right child has a higher key value than the key value of the parent node.
This way, we can explore the tree in such a manner that we can have an ordered list of key value as an output, which make data retrieval an easier job and reduces complexity on database management operations.

M-Way Trees [10] were designed to overcome the weaknesses of a Binary Search Tree. As all nodes can only store a single key and can have at most 2 children, the height of the BST is highly increased when we add some data. Moreover, a tree data structure has a complexity for operations like insert, search and delete that depends on the height. Here are the properties of a m-way tree :

1. Keys in each node are sorted
2. A node with k keys can have at most k+1 sub-trees, sub-trees can be empty
3. The i-th subtree of a node may only hold values j in the range of $key(i) < j < key(i+1)$

Index Sequential Access Method (ISAM) [23] is a static index structure based on the m-way tree structure. It's most efficient when the file we want to store will not be manipulated (insert, search, delete) frequently. All data entries are contained inside the leaf nodes, like the B+-Tree structure that we will explore later in this paper. However, if the data we want to insert exceeds the leaf node capacity, it is stored in an overflow chain that can cause lower performances for data retrieval. We need to reconstruct the files after inserting new records to maintain the sequence (static). This method allows us to manipulate, create and monitor computer files, so we can access the data in a sequential (in the order the data were entered) or random manner (with an index). The nodes of an ISAM structure are like other tree nodes, they contain indexes and pointers. However, the indexes are fixed and pointers don't change during inserts and deletes.

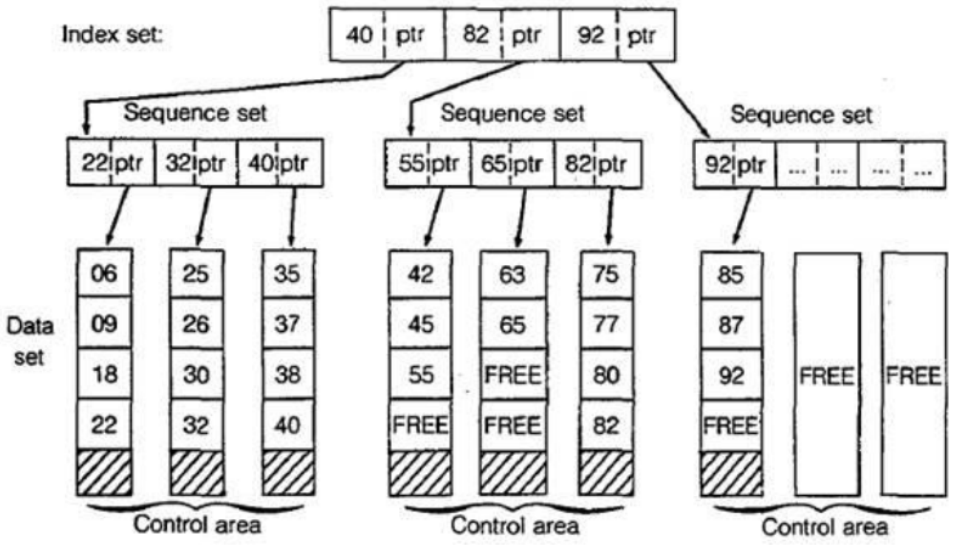


Figure 4: ISAM static data structure example

B-Trees, on the other hand, are dynamic. With operations like insertion and deletion, the tree will adapt itself according to its properties, the height might change (grow or shrink). We will see in further details in the next section the properties of a B-tree.

Let's compare the ISAM structure to the B-tree structure :

1. ISAM is better for static tables while B-Tree handles tables that are growing quickly much better.
2. ISAM requires fewer disk operations than a B-tree to visit a data page
3. ISAM is better for small tables as B-Tree requires a lot of pages no matter the table size.
4. ISAM requires no locking in the index pages, while B-Tree incurs index locking.
5. B-Tree concurrent performance is better than ISAM if ISAM has long overflow chains.
6. B-Tree is better when sorting over the key, because the sequential access is automatic, there is no need to add sort clauses to queries.

To conclude, in the context of Big Data, where the volume and the velocity of data is more and more important, B-Tree structures are to be privileged over ISAM structure.

After seing briefly the ISAM and B-Tree structure, let's talk about **B+-Trees**, which combine features of ISAM and B-Trees. They contain index pages and data pages :

- $root + internal\ nodes = index\ pages$
- $leaves = data\ pages$

It has the dynamic structure of a B-Tree while having the property of storing the data only in the leaf nodes of ISAM. It is the most commonly used data structure in the OS for metadata indexing. We've chosen for this project to implement B-Trees and B+-Trees, and we will see in further details their properties in this paper. We will also see the use of spatial data indexing with the R-tree data structure. The different variants of R-trees will be explained in their respective part later.

3 Materials and Methods

We choose to work on $B-$tree, B^+-tree and $R-$tree. For each of them we will do Single thread, Multi-thread and Spark implementation. The following section's goal is to explains these implementations.

3.1 Global ideas for implementation ::

Tree Concurrency Control Algorithms

Multi-threading allows to executes multiple threads concurrently. To explain these algorithms it is important to understand find(), insert() and delete(), these functions are the most important (cf Introduction).

1. Coarse Grained Locking:

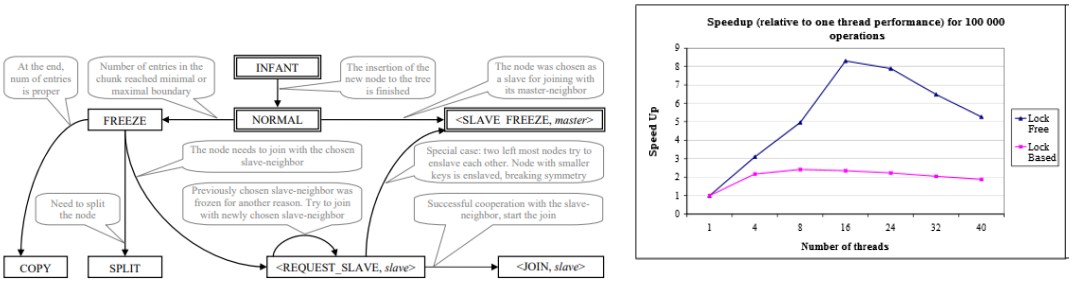
   This method is the easiest but it is the less optimal because find() = read-lock, insert() and delete() = write-lock. Each of these three functions block all the tree. This method is detailed in [11] chap 2.

2. Fine Grained Locking:

   This method[9] is the trade-off between Coarse Grained Locking and Lock-Free methods. Find() only lock one node at each time (lock the root then lock the child node and release the root ...). Delete() and insert() lock the all tree. This method is detailed in [11] chap 2.

3. Lock Free:

   Lock-free dynamic is a way to ensure the algorithm always progresses; at least one thread has to progress. The first implementation was in 2012 and everything is explained in this article [2]. A Braginsky and E Petrank used their previous study on lock-free chunks (blocks of memory that contain keys) to respect upper/lower bound for each nodes. This mechanism allows to keep the tree balanced. The main idea of this process is to freeze a chunk when an upper or lower bound is in danger and process the join or the split on a frozen chunk.



(a) Freeze process

(b) Difference between Lock Free and Lock Based

Figure 5: Taken from [2]

In our implementation, we used Coarse Grained or Fine Grained Locking techniques as we wanted to iterate and test quickly the performance of our algorithms. However, one improvement to our indexing structures would be to implement the the lock-free.

Tree Spark Implementation

Spark is a massively parallel processing system, its structured is based on RDDs[22]. Spark & RDD were introduced in response to the MapReduce cluster computing paradigm. The aim of Spark is to parallelize various operations on the same cluster or multiple cluster.

For our subject, we will shift our view about tree indexing[16] and consider to have a system which will have more reads than writes. Here, Spark would allow multiple users to make simultaneous queries to the same database, in a distributed way. We will more consider an approach where the data is bulk loaded into our tree indexes, and we will test and evaluate the performance of our queries.

For this, Spark provides an abstraction of data representation through RDD objects (Resilient Distributed Dataset) which are container-like elements partitioned across the nodes of the cluster and can be re-partitioned or act upon. There are essentially two kinds of transactions you can make on an RDD:

1. Transformations:

   Transformations create one or more new dataset (new RDD) based on another existing dataset (old RDD). Transformations are computed only when needed (when an action has been requested, see below), this is why there are lazy-evaluated. Map() and Filter() are two examples of transformations.

2. Actions (return a value):

   Actions are used when it is necessary to perform or collect insights from the data inside an RDD. Actions return non-RDD values in the form of an iterator, and they trigger the computation of all the needed RDDs traversed through the DAG of transactions (directed acyclic graph) and there are returned the back to the driver. Count() and collect() are two examples of actions.

3.2 B-Tree

B-Trees [6, 7] are data structures that have been widely used in files systems and DBMS. A *B*-Tree is a self-balancing tree that enables records to be retrieved, inserted and deleted in **logarithmic time**[13]. They are a generalization of *Binary* Search Trees, allowing nodes to store more keys, and also have more than 2 children.

The **advantages** of *B*-Trees over other Trees are :

- Maximizing the number of children per nodes reduces tree's height, and reduces the amount of balancing operations we will perform
- As data is also stored in internal nodes (non-leaf nodes), when we execute a query and data is stored in a internal node, the query succeeds and ends faster than a B+ Tree query.

Here are the rules to respect when building a *B*-Tree of order m :

1. Keys in each node are in ascending order
2. At any given node, the following is true :
   - Sub-tree starting at record $Node.branch[0]$ has only keys that are less than $Node.Key[0]$
   - Sub-tree starting at record $Node.Branch[1]$ has only keys that are greater than $Node.Key[0]$ and less than $Node.Key[1]$ ...
   - Sub-tree starting at record $Node.Branch[last-key]$ has only keys that are greater than $Node.Key[last-key]$
3. All leaves are at the same level
4. All internal nodes have at least $m/2$ number of children
5. Root node has a minimum of 2 children
6. Each leaf node must contain at least $(m/2 - 1)$ keys
7. Internal nodes with k number of children contains at most $k - 1$ keys

To illustrate a bit these rules, here is a representation of a B Tree of order 4.6.

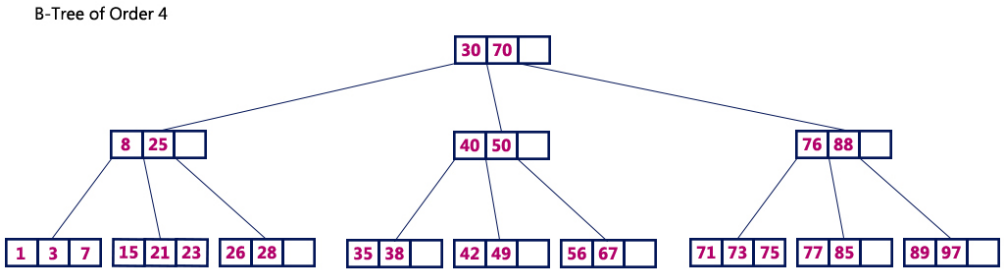


Figure 6: Example of a B tree[2]

Let's talk about the operations performed in logarithmic time that we mentioned previously : Search(), Insert(), Delete()

Search Algorithm : let's say we want the key k, we first start to look for this key at the root. If the key is not present in the root, we go down a level to the sub-tree with keys in range of k. If we can't go down a level (leaf node), the key value is considered not in the B-Tree.

Insert Algorithm : we always start an insertion at a leaf node, we found ourselves facing two cases : the node is full or not-full. If it is not full, we simply insert the value according to an ascending order. Else, we need to perform a **split node operation** : we find a median value among the keys, that we insert in the parent node. We then create two new nodes, values that are less than the median values goes into the new left leaf node and other values goes to the new right leaf node. This may cause an entire restructuring of the tree (if the parent node is already full, we need to go up a level, etc...)

Delete Algorithm : if the key is in a leaf node, we delete the key. If the node that holds the deleted key contains less than the minimum number of keys, we rebalance the tree recursively until all nodes satisfy the B-Tree conditions. If the key is an internal node, we swap the key with its inorder predecessor (largest key in the left subtree of the key) or inorder successor (smallest key in the right subtree of the key). Then, we delete the key which is now in a leaf node (in the implementation, we call the delete method on the key which is now a leaf node).

[2]This figure is done by`http://www.btechsmartclass.com/data_structures/b-trees.html`

3.3 B+ Tree

The B^+ tree[24, 17] is a balanced index structure based on B tree. The main difference between B tree and B^+ tree is on data pointer. B+ tree store data pointers only at the leaf nodes of the tree. So, the leaves must necessarily store all the key values along with their corresponding data pointers to the disk file block, in order to access them. This gives advantage to the B+ tree, as for the B-tree, it stores the data pointer along with their key value in each node of the tree, which reduces the number of entries that can be packed into a node. Moreover, the leaf nodes are linked to provide ordered access to the data. It can be useful for building a file system, for instance, BTRFS[15] and New Technology File System (NTFS) uses it for directory indexing.

To visualize the differences between the B tree and B+ tree; on fig.7, we used the same index on the same order as fig.6. We can notice that the tree is bigger with more node and more leaf.

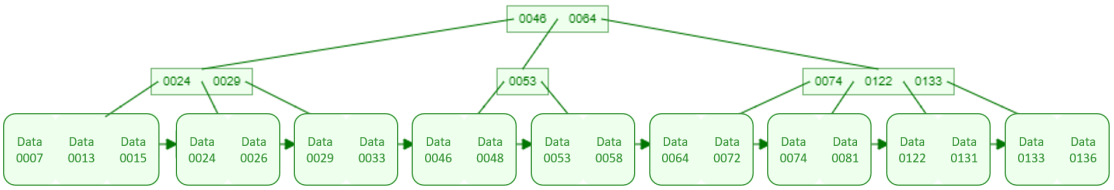


Figure 7: Example of a B+ tree [3]

Node size impacts the time complexity. As well explain in [24] "*It supports the exact match query as well as insertion/deletion operations in* $O(\log_p n)$ *I/Os, where* n *is the number of records in the tree and* p *is the page capacity in number of records. It also supports the range searches in* $O(\log_p n + \frac{t}{p})$ *I/Os, where* t *is the number of records in the query result*".

A lot of studies[4] suggest that the best size is a few cache lines. In modern computer cache lines size = 32, 64 or 128 bytes. It is possible to store int or string object as key.

They are 3 basics operation : Find(), Add(), Delete(). The 3 operation has been implemented on the java program correctly on single and multi-thread (cf.Experiment).

1. Find a value:
   (a) Starts at root node
   (b) Compare node's keys with the search value
   (c) Find the path to the children node and redo the process until finding a leafnode
   (d) If key value = search value return the record else return "not found"
2. Add a value:
   (a) Find the leaf where the value belong with the help of find()
   (b) Insert the key and the record pointer
   (c) If the leaf is more than full, split it.
3. Delete a value:
   (a) Find the leaf where the value belong with the help of find()
   (b) Delete the key and the record pointer
   (c) If the leaf is less than half empty, join it with other leaf node if it is possible, this operation can imply a split. Otherwise merge leaves and order a delete operation in the parent node.

[3] This figure is done by the help of `https://www.cs.usfca.edu/~galles/visualization/BPlusTree.html`

The B^+tree class implementation

As Shown in fig.8, we implement 3 Main classes; Data, Node, BPlusTree. The *"Data class"* is the consider as the schema of our database. In our case, we used a simple schema formed by an index (Integer) and an information (String). The *"Node class"* is the base for the construction of the B^+tree. It is an abstract class from which it extend 2 sub-classes *Leaf Node* which contained the pointers to the database, and *Inner Node* which contained the indexes and the pointers to others nodes. The *"BPlusTree class"* is the main class which contained the basics functions; Find(), Add(), Delete(), and other functions that back up them up, as well as, the Bulk loading function. It been extend to the *BPlusTree Multi* which change the function to fit better the multi-thread approach.

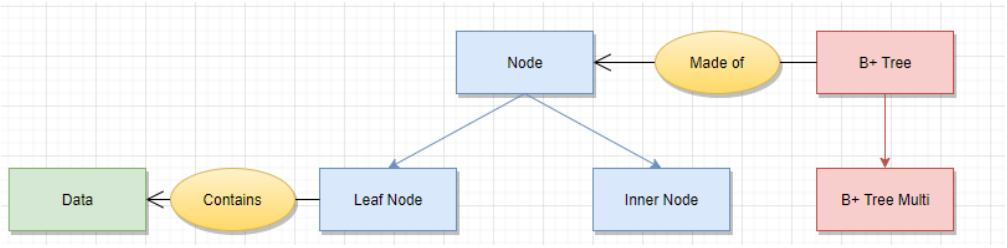


Figure 8: Class organization

Bulk loading

The first way to create B^+tree from scratch is to initialize an empty tree and insert tuple one by one (tuple-wise insert). There exist a better way, it is Bulk loading[5, 1]. This method needs a sorted dataset on key values. The tree is created bottom-up from left to right. Left to right means from small key values to high key values and bottom-up means from leaf node to root node. We also introduce an extra parameter *lambda* which refers to the filling rate of the leaf nodes, it should be between 0.5 and 1. The filling rate can be very useful. In fact, at the beginning of creating a tree, you often expect to add new information fairly quickly. When your leaves are all half-filled from the start, you have enough space to add new data without having to reshape the tree, thus avoiding latency.

3.4 R-Tree

A spatial index is more or less identical to a normal index: the role of any index is to allow very quick access to a selection of items from a large amount of data. Spatial data objects are not well represented by location points and represented in multi-dimensional spaces. R-trees are tree-like data structures used to index those multidimensional information (called spatial data) in an efficient way. A typical use case for R-trees is the storage of geographic information such as restaurant locations or the processing of game static data. A typical request might be "find all the restaurants within a radius of 2 kilometers".

The problem is that traditional one-dimensional structures database indexing are not suitable for multidimensional spatial searching. In particular, we can mention those that use an exact match with values (hash table) or those that use key values ordering (B-Trees, ISAM) [20].

Many structures have been proposed to handle multidimensional data: Cell methods, Quad trees, k-d trees, K-D-B Trees. But for different reasons, these methods have proven to be not appropriate (not good for dynamic structures, useful only for point data ...) [8]. Among the functional methods, we will deal in particular with the R-tree structure, which represents data objects by intervals in several dimensions.

The main idea of the R-tree is to represent nearby objects by their enclosing rectangle at the immediately upper level of the tree, the "R" in R-tree is for rectangle. There are 2 types of nodes: leaf node and non-leaf node. A leaf node is defined by a tuple (I, tuple-identifier) and a non-leaf node by a tuple (I, child pointer)

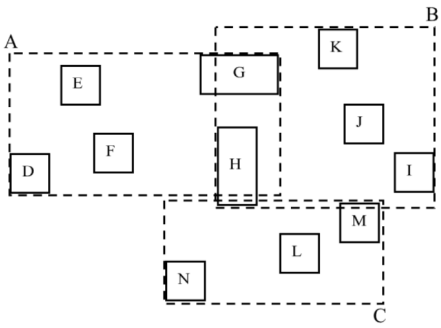


Figure 9: An example of 2D data and their minimum bounding rectangles [20].

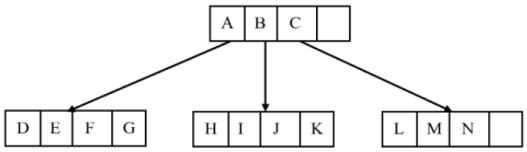


Figure 10: The corresponding R-tree [20].

where I = (I1, I2, ..., In) is an n-dimensional rectangle. Each node has a maximum capacity, noted M (a node has at most M entries), and a minimum capacity (inferior to M/2) guaranteed by the tree [8].

Concerning search operations, the R-tree allows you to do this in a simple way: you use the enclosing rectangle of a branch to determine whether to continue exploring it, thus eliminating most of the irrelevant nodes until you reach the desired element. R-trees do not guarantee good worst-case performance, but generally perform well with real-world data.

The other operators, insertion and deletion, which modify the structure of the tree, are more difficult because the tree has to be balanced (leaf nodes are at the same height) and the rectangles must fill the space optimally (less empty space, less overlap)[8].

Insertion can be summarized in 3 main phases: locating the insertion position, splitting the node if necessary, and if the split takes place, propagate changes downwards and upward. Splitting is the act of, when a page is full, dividing its elements by trying to minimise the area of the rectangles corresponding to the two subsets. Deletion can also be summarized in 3 main phases: Find the node concerned by the deletion, delete the element, spread the change downwards and upwards.

Let's focus on three implementations of the essential algorithm for R-tree: the split. As explained above, the main idea is to find the minimum area for the two child nodes.

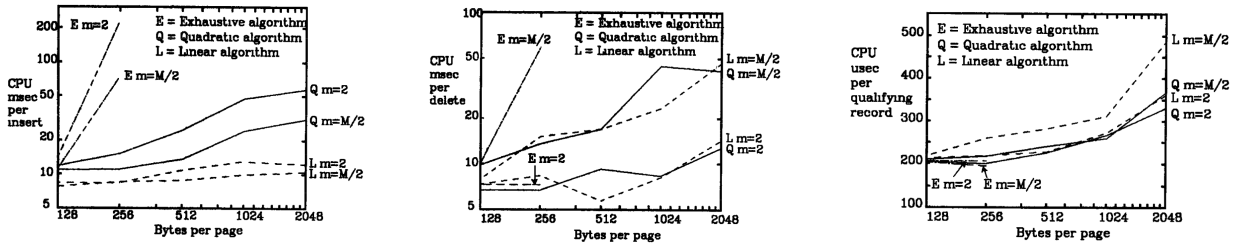
- The **simple algorithm** would therefore be to generate all possible grouping partitions. The problem is that the number of possibilities is quickly very high.
- A **quadratic cost in M algorithm** is also able to find a small-area split. The algorithm chooses 2 entries as the first elements of the groups, which corresponds to the worst possible combination. Then the remaining objects are assigned to groups one at time until all objects are assigned.
- Finally, a **linear cost in M algorithm** exists and its implementation is close to the quadratic algorithm. It first finds extreme rectangles along all dimensions, before adjusting their shapes and selecting the most extreme pair.

Figures below show some performance results between the different algorithms that can be found in the literature. Details of the test algorithm used here are available in [8].

Now let's expose the approach we have taken for the R-tree in-memory indexing structure. The index type is clustered, similarly to the B-tree and B+tree structures, and we construct the tree recursively with a Node$< T >$ object (T being any possible shape that is constrained by an interface we created), where each Node is either an internal node (children are a list of nodes) or leaf nodes (children are records). We chose to have our own Record$< T >$ abstraction in order to allow to have any type of data if the use-case is permitting it, and for now we only have an "id" field.

Here are the operations we implemented for single and multi-thread versions:

1. **Insert()**: We insert records just as described by [8]. We find a leaf node by comparing spatial proximity and node enlargement, add the record to the node and perform splitting if necessary. We implemented the 2 original splitting strategies, linear and quadratic splitting, as they defined the basis for the R-tree indexing structure.
2. **Delete()**: We follow a similar procedure here, finding the leaf node and record, and if the record exist, we delete it and re-balance the tree. Here again, we might split nodes and assign a new root.
3. **search()**: This is the traditional index/search algorithm. We traverse down the tree comparing spatial proximity, and return the record if present, and null if it is not found.
4. **rangeSearch()**: This is a more real-world query that can be translated to: "Find all the restaurants in the given range distance". It selects branches of the tree to expand upon based on area overlap and returns all the given shape records in the input area.
5. **knnSearch()**: This is the implementation of the k-nearest-neighbors for an index structure and can be translated to "Find the 3 closest restaurants to my current position". It proceeds with the incremental knnSearch() implementation cited by [12]



(a) CPU cost of inserting records. (b) CPU cost of deleting records. (c) Search performance CPU cost.

Figure 11: Performance results [8].

Keeping in mind that minimization of coverage and overlap is crucial to the performance of R-trees, there are several other dynamic versions of R-tree that we can find in the related literature [20].

- **The R+ tree** is like a mix between the classic R-tree and the k-d-tree. The main difference is that the algorithm is based on the absence of overlapping: the nodes are not optimally filled and an identifier can be found in several leaves simultaneously.
- **The R*tree** focuses on both overlapping and coverage minimization. It therefore uses a hybrid strategy: a revised node split algorithm based on perimeter, then minimizes overlap and a insertion algorithm, in which overlap is minimized for leaf nodes, while for inner nodes, enlargement and area are minimized.

- **The Hilbert R-tree** is the structure that out- performs all the older ones. It sorts rectangles according to the Hilbert value of the center of the rectangles. More precisely it uses the Hilbert curve, to impose a linear ordering on the data rectangles.

4 Experiment

4.1 B Tree Results

Experiment setting

The Database we used for our B-Tree test is the same as the one used in the B+-Tree approach, in order to facilitate comparison in the results, but also the coding. As we divide our work into two separate groups for B-Tree and B+-Tree, we found it more convenient to work with the same dataset.

	Dataset Information
Size	10 Millions Entries
Data Keys	IDs
Data Values	Nickname, Location, Level of Trust
Example	636973 ; Jake ; Uptown ; 5;
Type of generation	Random - Python file

Table 1: Dataset Information

To test our single and multi-threaded implementations, we chose to study the time taken to complete *search*, *insert* and *delete* queries, ranging from 10 000 keys to 1 000 000 keys with a step of 20 000 keys between queries. Keys are generated randomly. The multi-threaded implementation adapts itself according

to the resources of the machine which runs the code. If the machine has a maximum of 8 threads, queries will be split in 8 parts and feed to each thread. But if the machine can handle 16 threads, queries will be split and threads will be created accordingly. We can also override this parameter to choose a specific number of threads. We've use the Fine-Grained Approach for the multi-threaded implementation of search queries and Coarse-Grained Approach for insert and delete queries. The queries were built randomly to avoid testing

only a specific search query. However, one of the downsides of this random query approach is that results were found to be inconsistent depending on the generated keys. To overcome this downside, we implemented an epoch system.
For every query size t, we execute the query multiple times (depending on the epoch number) with different random queries, but with the same size t. This approach tend to balance the keys that are not in the tree with keys actually in the tree, keys that are in internal nodes and keys that are in leaves, etc... Finally, we take the average time taken by the query of size t over the epochs and save it for our output csv file.

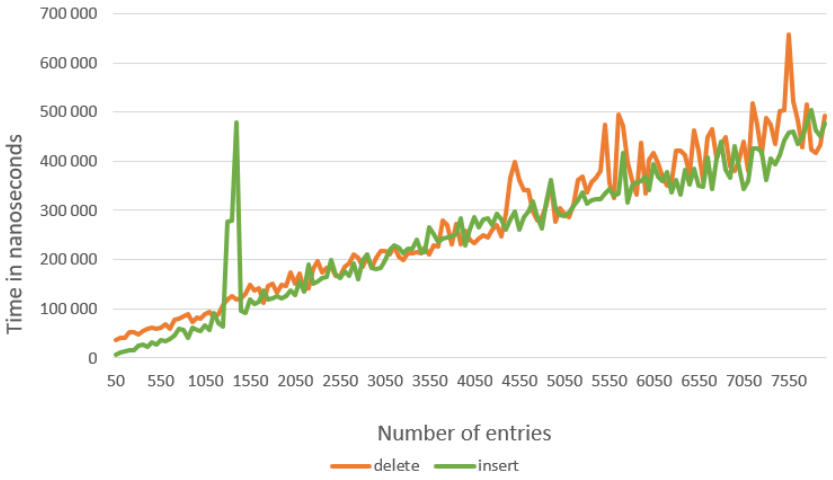
Finally, here is the environment that we used for our tests:

- Processor: AMD Ryzen™ 7 3700X CPU with 8 cores (16 threads) @ 3.60GHz - 4.40GHz (x64-based processor)
- RAM: 16 GB and 1TB HDD
- Java SDK 11, Spark 2.4.5, Intellij IDEA 2019-2020

Single thread and Multi-thread

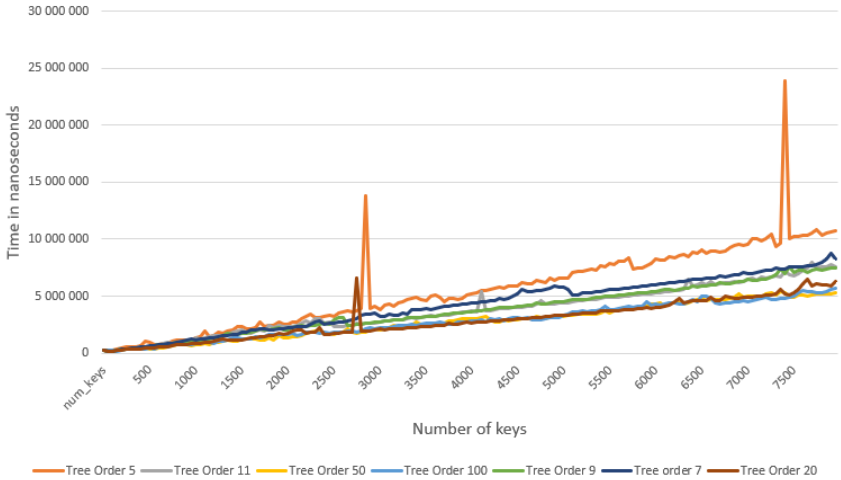
Single thread

As we can see on the graph below, in a single threaded context, insert and delete operations have nearly identical run time complexity. For an equivalent number of entries to add or remove, the run time is very close.



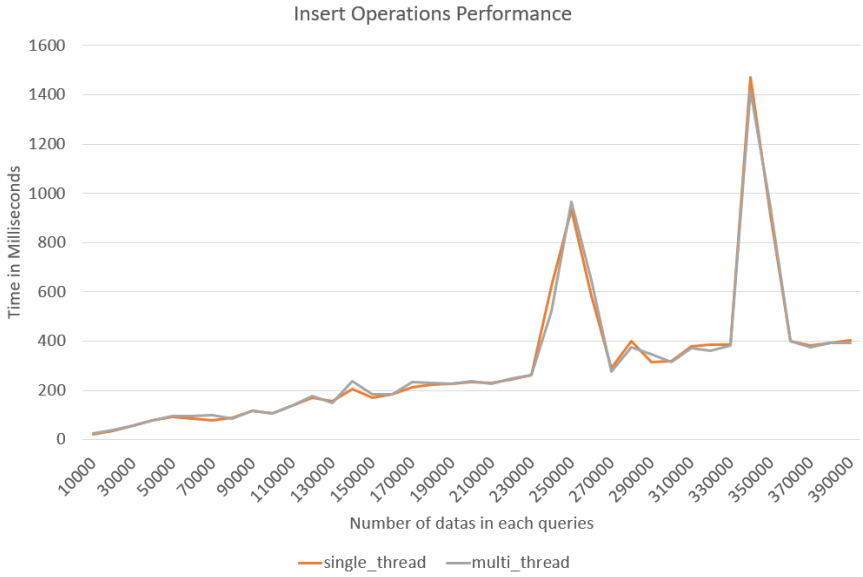
The random spike at 1400 is due to the fact that the computer isn't idle when we run our tests, so sometimes random spikes will occur due to other processes borrowing the thread or delaying the operation. What matters is the trend of our test.

We tested different tree orders on the search algorithm to see how the order of the tree could impact the search time. From our experiments, there was no gain in terms of performance going above a tree order of 20.

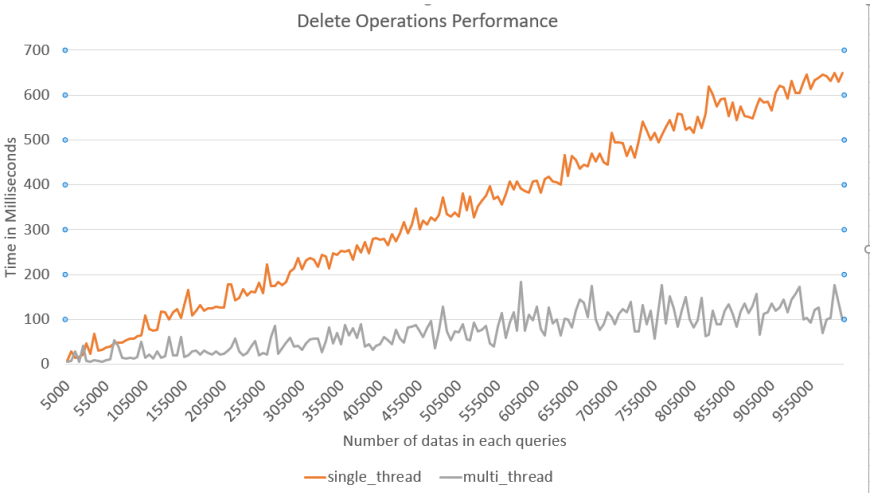


Theoretically, the tree order should be set to the size of a page on a disk but we didn't find a programmatic way in Java to retrieve this information. Also, Java doesn't allow to choose how we want to allocate the objects on the disk so we could not replicate the real life scenario of 1 node = 1 page.

Multi thread



For the insert operations, as we can see on the graph, there's a lot of noises in the time data that we retrieved. There is little difference between multi-threaded and single-threaded due to the fact that our multi-threaded approach for this insert operation has been the Coarsed-Grained approach.The spikes on the graph can be interpreted as random datas being inserted in leaves rather than internal nodes, which can take more time. We've taken less steps and less data than delete due to the high amount of time. We see less noise because insert operations are always successful while delete can either be a successfull and unsuccessful.



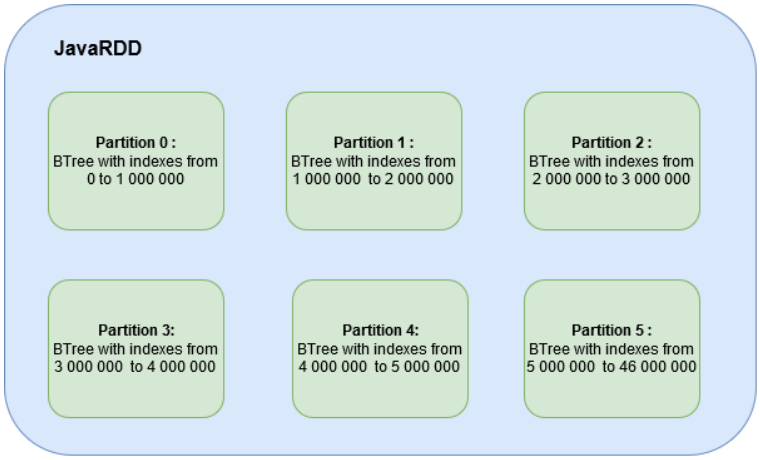
The particularity of the Delete Testing has been the creation of copies of the tree, in order for our delete epochs to not delete the same tree (and found ourselves with an empty tree at the end of our tests). At each query size step, we performed 5 epochs on a tree that is rebuilt evertime. Multi-threading has taken less time than single threading, and seem to be scaling quite well with the amount of data to delete.

Spark

Spark provides RDDs which can be partitioned to perform parallel operations on each partition. Indeed, when run a query on a Spark RDD, each partition will translate into one task, and one task is runned by a single thread.

As a result, our approach to use our BTree implementation in Spark was:

1. import the dataset in Spark
2. partition it with a custom partitioner which assigns the correct partition to each sample
3. create a new JavaRDD which preserves the partitions, but now each partition only contains a single BTree with all the data of the partition loaded inside.



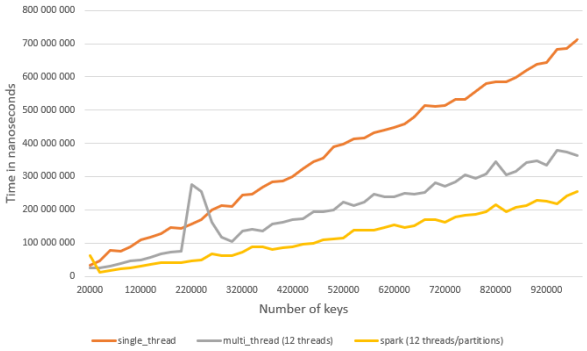
(a) Spark implementation of BTree

When a search job is actually called, each thread will do the following on one of the partitions:

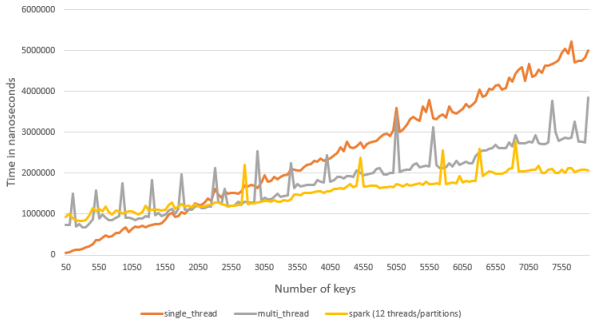
1. among the searched keys, only keep the ones in the range of the partition
2. search the keys in the BTree

Search algorithms comparison: single-thread, multi-thread and Spark

Intuitively, the Spark search implementation should be faster than the multi-threaded search implementation, which is faster than the single-threaded search. Indeed, in our Spark implementation, not only we have as much threads as in the multi threaded implementation, but also each spark thread works on a smaller tree (as explained above). This intuition shows on the following tests.



(a) Search algorithms comparison for large queries



(b) Search algorithms comparison for small queries

The first test covers queries from which search from 20 000 keys to 1 000 000 keys. On this graph, we clearly see the benefits of the Spark implementation which scales well for large queries.

The smaller graph on the left shows the breakpoint where the multi-threaded implementation and the Spark implementation become better than the single threaded; this breakpoint is around 2000 keys to search. Before that breakpoint, the overhead of the multi-threading makes the multi-threaded and Spark implementation slower.

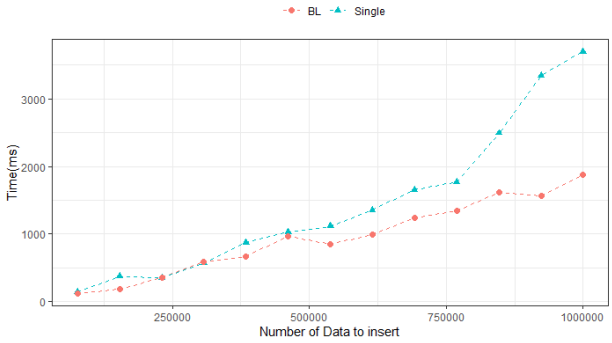
4.2 B^+ Tree Results

Single and Mulithread:

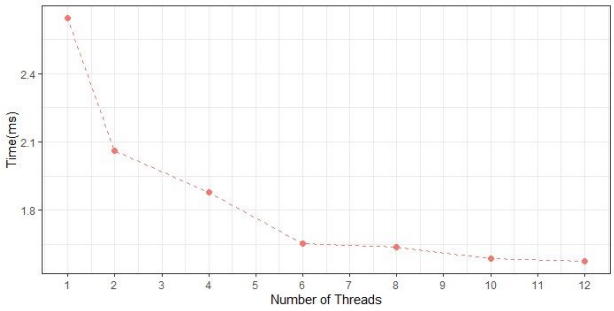
In order to test and verify the effectiveness of the implemented approach, we created a simple database same as B-tree using a python script. We only used two columns from the database; Id and Nickname, it is the easiest way to test the complexity of B^+ tree and compare it with others algorithms. For each test, the size of database is different, this will be detailed below. Here is a sample of the database in table.2.

ID	Nickname
1	Johnny
54	Paul
10	Matt
3	Bully
29	Pedro
15	Tim

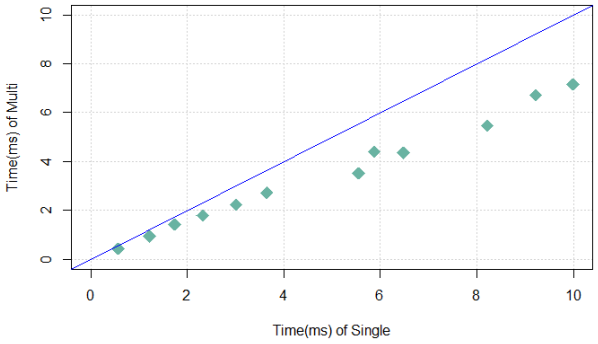
Table 2: Data Sample



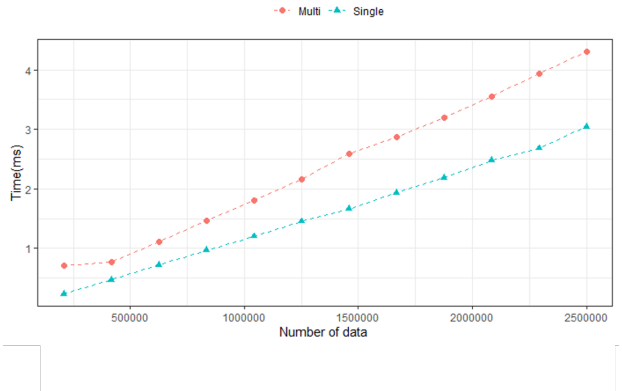
(a) Sequential vs Bulk Loading: Insert()



(b) Thread parameter impact: Search()



(c) Single vs Multi thread: Search()



(d) Single vs Multi thread: Mixed()

Figure 18: B+Tree Results

The Figure (fig.18a) represents the results of the first test which is a comparison between the sequential insertion and the Bulk Loading methods which is also an insertion method. As we can see the latter is faster than the classical method. This result is in agreement with the theory explained above. There is a breaking point around 600 000 datas to insert. Before this point the two methods are quite equal but after that the

Bulk loading takes the lead on the sequential. This is due to the fact that we already can create all the **leaf nodes** at the beginning. Then, we just insert in the tree leaves after leaves whereas we insert keys after keys in the sequential approach. For this test we used 1 million of lines.

The purpose of the next Figure (fig.18b) is to understand how the choice of the number of threads affects the multi-thread performance. This test was performed on a computer with 12 threads. It makes sense because if we use just 6 threads to do the work of 12 threads the processing time will be approximately 2 timers slower. For this test we used 25 million of lines. For 1 to 12 threads we plot the time for searching 10 million of keys.

On the Figure (fig.18c), each point represents the time needed for each methods to search a given number of keys. Like the previous test we used a dataset with 25M lines and we give to both methods a growing number of keys to search going from 0 to 25 million divided in 12 steps. As we can see the multi-threaded approach is slightly faster than the single-thread, it saves 30% of time in average for the last 4 points. This is due to the read-lock implementation. For example if we want to search 3 keys values with the single thread implementation we will have to carry out the search operations one after the other. On the contrary, in the multi-thread implementation we are able to start the second and the third search operations before the first one has finished, hence we save a lot of time.

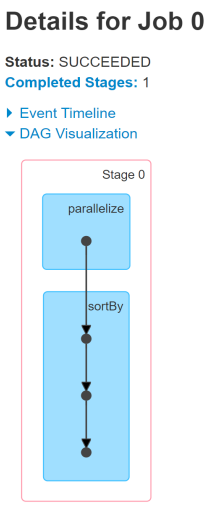
For the last Figure (fig.18d) we used a dataset of 5 million rows. This test consists in comparing the single thread and the multi thread implementation for a mix of operations. We give to the three methods: Insert(), Delete() and Search() from 0 to 2.5 million of random keys divided in 12 steps. Paradoxically, we notice that the single thread is faster than the multi thread approach. In fact, the Delete() and Insert() operations are not that much improved with the fine-grained locking approach used for the multi threading implementation. Moreover, they are even slower because of the task assignment between the threads. The multi-thread implementation could be faster by adding a lot more Search() operations to our test.

Finally, here is the environment that we used for our tests:

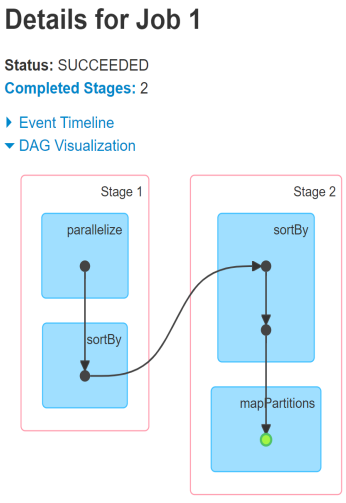
- Processor: AMD Ryzen™ 5 1600 CPU with 6 cores (12 threads) @ 3.2GHz (x64-based processor)
- RAM: 16 GB and 1TB HDD

Spark:

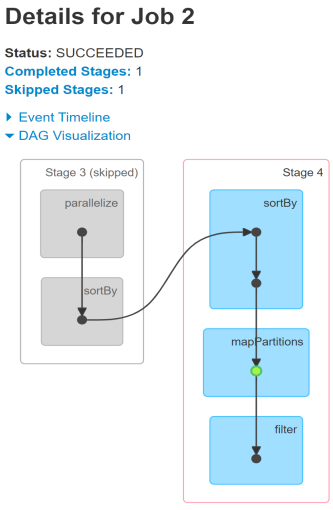
In this part, we do the Spark implementation with JavaRDD object, which is use to generate parallel tasking. Here we used another PC with different configuration of : Intel(R) Core(TM) i5-2520M CPU @ 2.50GHz with 4.4 RAM and 2 CPU cores (maximum thread of 4).



(a) DAG -Loading and sorting Data



(b) DAG - Count Operation



(c) Search using Foreach operation

Figure 19: B+Tree Spark Jobs

1. **The inputRdd**: This represents is the initial data records, loaded into memory from a .csv file. This RDD is a "Data" type, and it is partitioned based on maximum number of thread of the computer (here, 4).

2. **The inputRddSorted** : This is a transformation over the inputRdd, where the id of the data will be sorted, to help us make B+tree that don't overlap between them.

3. **The BTreeRdd**: This is the last transformation made on inputRddSorted, to get B+tree. This RDD is partition based on the same partition of the inputRDD.

In fig.19, the spark jobs began well by loading the data of 1.5M records, then sorting them in around 11 seconds. Here, we take the approach of loading the big amount of data and fixing the JavaRDD to apply the other operations on it, like search. Next, we test a simple operation of count() after applying the cache() to the RDD, that help us to load once for all the *BTreeRdd* in the memory. At final jobs, we launch a search task that loop on each BPlusTree partition to get the one with the searched key. Each search task take around 15-50 ms , which is good for search only one key but it became wore if we raise the numbers of searched keys as seen in fig.20.

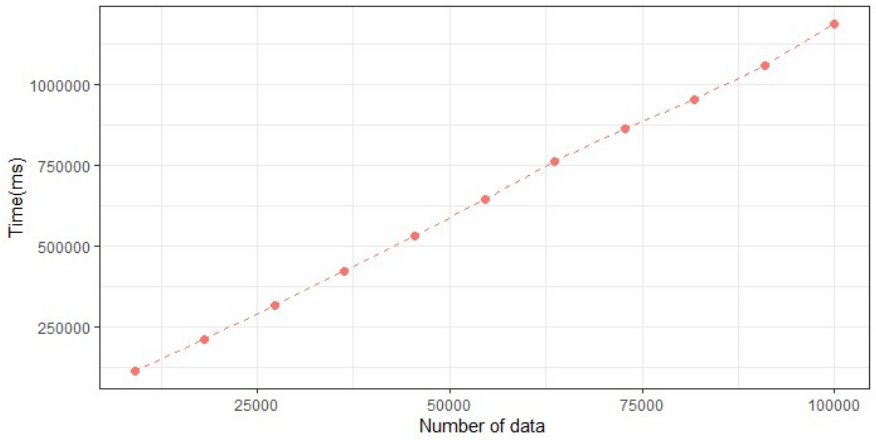


Figure 20: Spark result to search

Here fig.20, we can see that the Spark search takes a lot more time than excepted, which makes it slower than the single thread implementation. Although, few reasons for these results could be that the computation and test weren't made on the same machine, and that using the local cluster settings, searching through a high volume of data would still take some time. Ideally, we would want to test and assess the search query performance on a dedicated cluster with separate machines running each executor on the distributed data in parallel.

4.3 R Tree Results

Single and Multi-thread

In this part we will compare and assess the performance of our algorithms for insert(), search(), delete() and the 2 other search algorithms we implemented.

Part 1: Choice of the Dataset

To do our experiments on a small scale, we created a small database that we made ourselves. But for a larger scale, we created a class that randomly generates a set of points or rectangles based on the number we gave it.

Part 2: Experiment settings

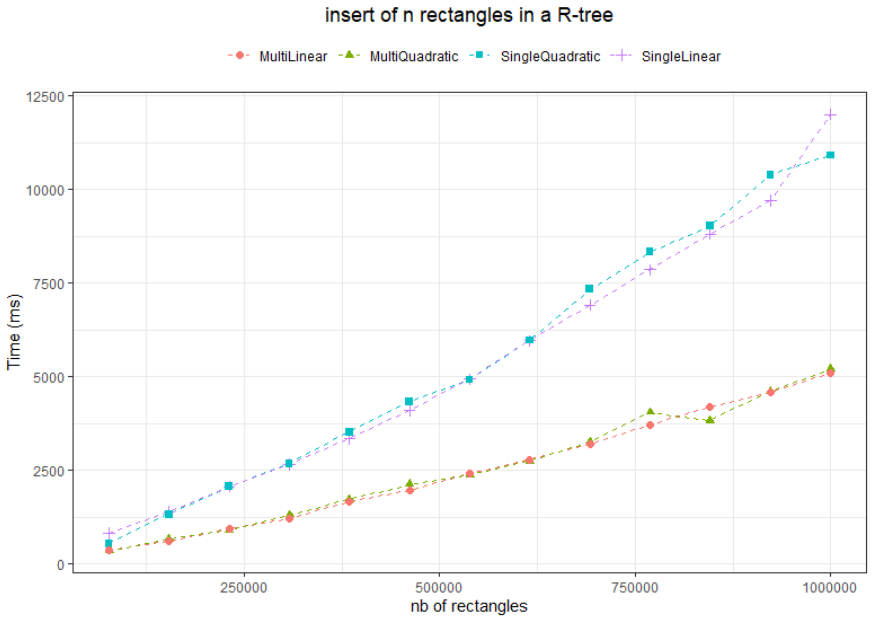
As for the experiments we did below, we decided to compare the different approaches using the metric of time (in ms). We did this because it was the most quick and easy way to model the difference between two approaches.

For the thread setting in multi-thread, we used the newCachedThreadPool() most of the time because the thread allocation was more balanced that way, and fixing manually the number of threads was not scalable enough.

As for the computer we use to make the experiments, we had the following settings :

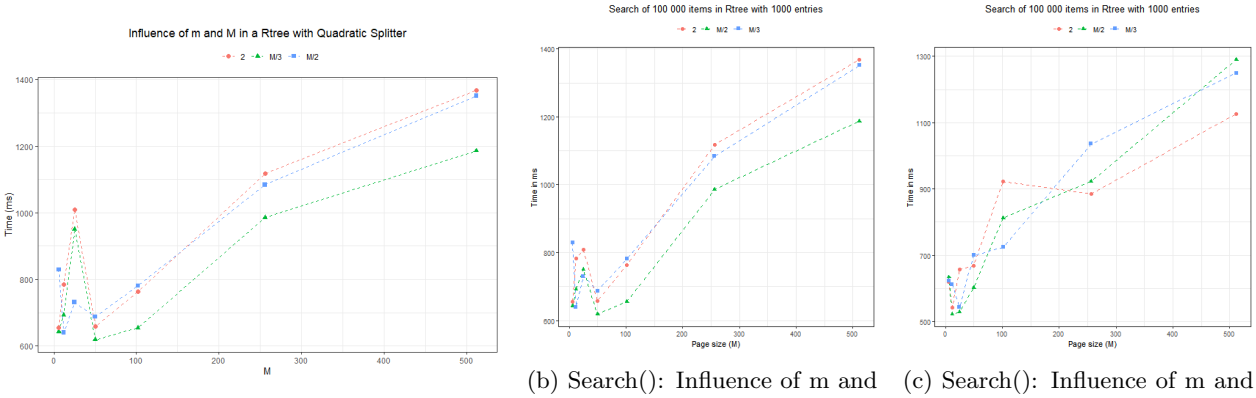
- Processor: Intel(R) Core(TM) i7-7700HQ CPU with 8 cores @ 2.80 GHz (x64-based processor)
- RAM: 8.00 GB and 1TB HDD
- Java SE 15 and Eclipse IDE 2020-09.

Part 3: Experiments
Linear and Quadratics splits approaches:



(a) Insert(): Comparison between the use of Linear and Quadratic split on single and multi-thread

In 25 we compared the two different implementation of a split we made, that is the linear and quadratics splits, using the insert method. According to these graph, we noticed that linear and quadratic splits strategies have approximately the same processing time. This is why we decided to use only the quadratic splitting method for the rest of our experiments. Let us now see the influence of the min and max page size in 22.



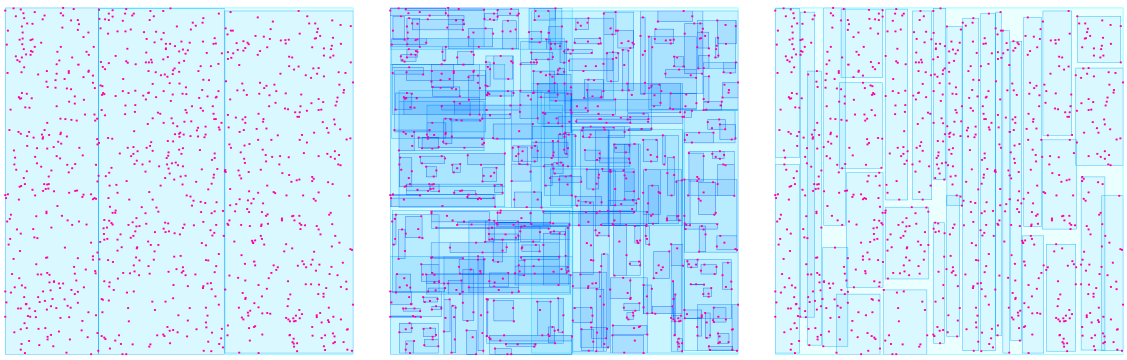
(a) Insert(): Influence of m and M

(b) Search(): Influence of m and M with Quadratic splitter

(c) Search(): Influence of m and M with Linear splitter

Figure 22: These 2 plots show us the influence of m (minPageSize) and M (maxPageSize) over the efficiency of our insert and search methods.

Given we have chosen the quadratic approach for node-splitting, we can see that $m = M/2$ seems to be the best tradeoff. We followed the test with a value proportional to the size of the input data, $M \approx N/20$ and $m = M/2$. Let's visualize also edge cases with min and max pages sizes:



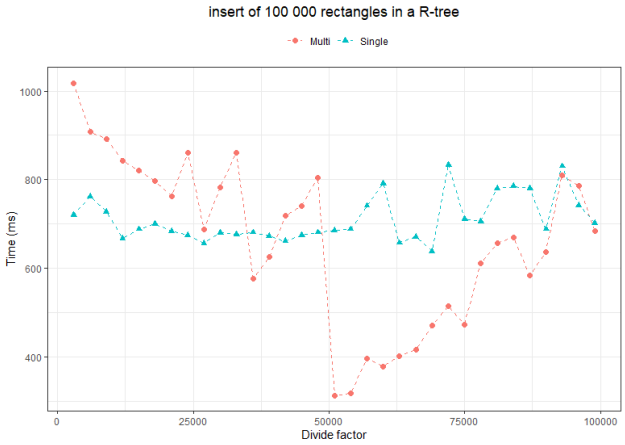
(a) An example of R-tree with too few page nodes

(b) An example of R-tree with too small min page size

(c) An example of R-tree with well balanced page sizes

We can draw the same conclusions with the visualization above, the first case show almost no overlap between the records, but the nodes contain too much data to be efficient in spatial queries, the lack of more nodes will introduce latency in the queries. For the second graph, the page size settings introduce way too much overlap, and the queries will certainly go into several tree branches even if they don't need to. The last graph is a good trade-off between number of records in node and minimal overlap, and confirms our intuition on the setting of m and M.

Divide Factor



(a) Insert(): Divide Factor comparison

(b) Search(): Divide Factor comparison

The divide factor is the number with which we divide our set of N items to pass to any method. For example, if we want to insert 1000000 items and have a divide factor of 1000, each worker thread will process $\frac{1000000}{1000}$ items.

While the variations in the single thread parts are largely due to method calls overhead and randomness of records, the variations in the multi-thread case can be explained by looking at the insert graph. Having 1 item per worker thread introduces a lot of overhead in creating the tasks and is slower than the single-thread case, especially given the insert method is quite heavy in nature. But, there is a threshold at a point where the number of threads becomes distributed enough to accommodate the tasks without creating too much overhead. In the search case, since the operation is easily parallelizable and not heavy in nature (no node-splitting), the divide factor is efficient even when very small, and becomes not efficient when the queries are not parallelized enough across the workers threads.

It's why we chose to use a divide factor of $\frac{n}{2} + 1$ for every insert test and a divide factor of $\frac{n}{10}$ (approximately) for every search test.

Insert method

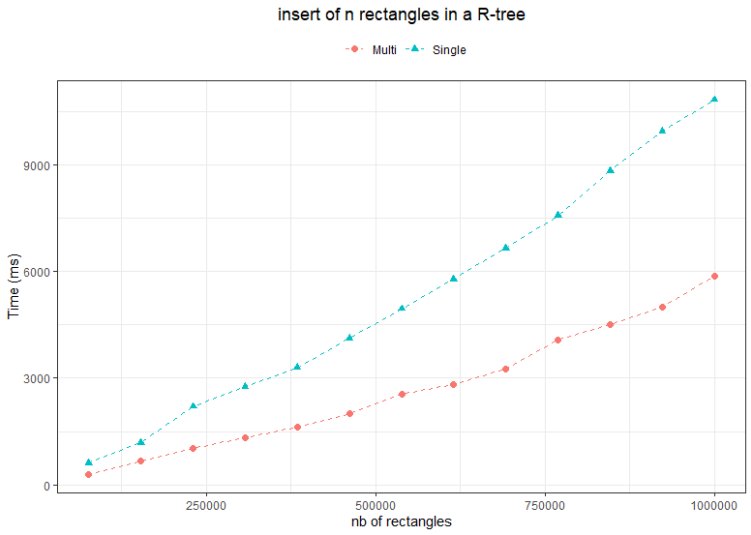


Figure 25: Insert(): Comparison between the use of single-thread and multi-thread

So as we use the fine grained locking for the search methods, it's expected to have the search queries

performing slightly better in parallel. Note here that the reader threads will not block the others readers, and that it will only block the writers in a find-grained way (if we wanted to insert at the same time). This is why spawning multiple threads for search is highly efficient.

(a) Search(): Small tree and varying records

(b) Search() : Varying tree and fixed number of records

Figure 26: faire la caption rhoya

We have tested our search method in two different ways:

- First we took a fixed R-tree of 1000 rectangles and increased linearly the number of search in the tree
- Then we took variants sizes of R-tree and made on each exactly 1000 search of a rectangle

Here again the results are expected, and it is worth noting that usually, R-trees are used with a relatively small number of items ($< 10M$), and the absolute results of 2000 ms for searching $1M$ of items in a R-tree of 1000 items are quite acceptable in terms of real-world use cases latency. We can also see that the search item count greatly influence the query time, as seen in the first graph, where searching only 1000 rectangles in a 50000 records takes under 200 ms.

Delete method

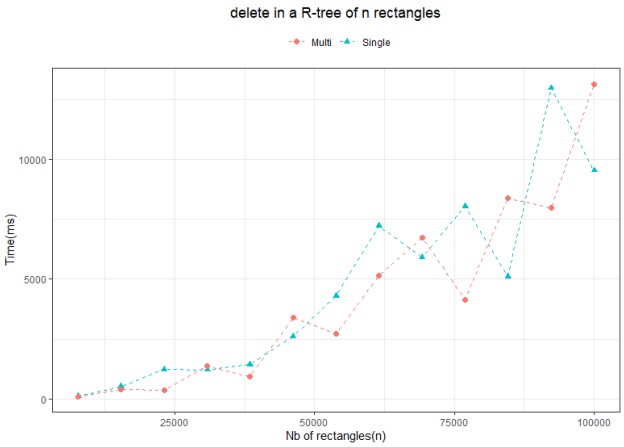
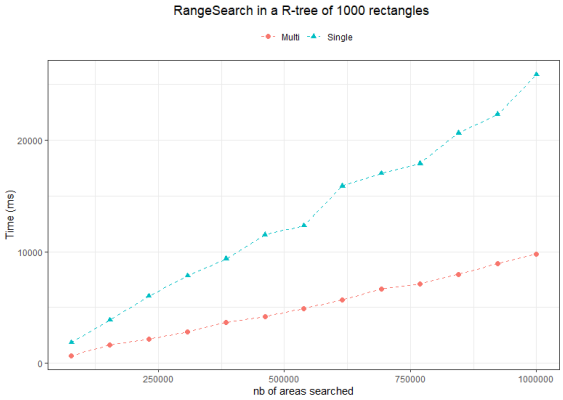


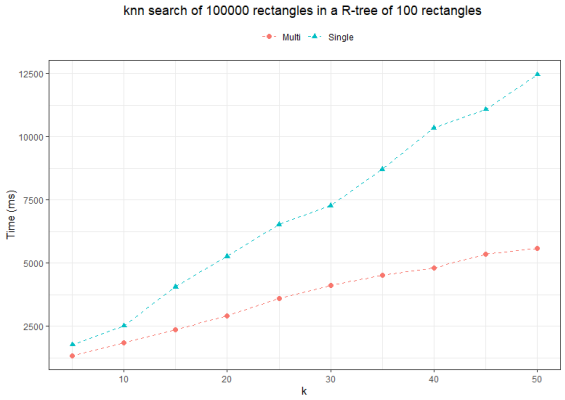
Figure 27: Delete(): multi-thread and single thread

In this graph we tested our delete method in single and multi-thread. Not surprisingly, we have no noticeable differences between single and multi-thread. This is due to the fact that the delete operation is heavy in nature, where node splitting and node re-insertion could occur. Thus, having a multi-threaded version with fine grained locking doesn't help us much here.

RangeSearch and KnnSearch methods



(a) RangeSearch(): multi-thread with lock and single thread



(b) KnnSearch(): multi-thread with lock and single thread

In these two graphs we decided to test our two last methods we implemented, that is the range search and the knn-search. It's important to show that for the knn search method, the search time is following a similar trend than the standard search, which means the real-world use cases are in sync with the standard search-indexing method. The latency is higher for rangeSearch probably due to spatial distance computation and to the number of items to search, compared to the KNN parameters and to the selectivity of the range records searched (searching on the whole range of the dataset). However, it still remains usable for a decent amount of queries.

Overall, we are satisfied with our results, which are in line with our forecasts, both in terms of speed, between single and multi-thread (search queries, insert/delete methods), and difference between linear and quadratic split and the performance as a function of page min and max sizes.

Spark

Let us now compare and evaluate the performance of using Apache Spark as a distributed environment with the R-tree-indexing structure.

As previously discussed, the paradigm in which we will place ourselves is the one that the indexing system will be used with a lot more reads than writes, as it is even more accurate here with the R-tree indexing structure, because it's used to index spatial-objects that are not subject to change every minute.

For example, R-tree would be best suited to index locations or shapes of restaurants or shops, and an user would be then able to query for these entities in a range perimeter of where he is with the rangeSearch() operation, or to query the 3 nearest entities via the knnSearch() operation.

Part 1: Choice of the Dataset

We chose to work on a randomly-generated dataset because it was lass skewed to assess the performance of the R-tree. Indeed, we tried first to take a real-world dataset like the NYCtaxi yellow taxi coordinates points for the year of 2016, but because of the nature of the data, objects were too skew towards a specific spatial cluster and interpolating them wasn't the best option to correctly evaluate the performance of the search queries. This is also due to the fact that R-tree indexing wouldn't be the best option for taxi trips locations.

	Dataset Information
Size	Variable size of records (10K to 10M)
Data Keys	Spatial shapes (Rectangles or points)
Data Values	Simpled IDs to spot the records and debug
Example	Rectangle($min_x, max_x, min_y, max_y$)
Type of generation	Random - Java program

Table 3: Dataset Information

Part 2: Experiment settings

For performing the experiments, we used the local Spark mode instead of the cluster one. We tested a local cluster configuration mode with Docker, but since it was operating on the same machine only with simulated cluster nodes through Docker, we preferred to assess the performance in local mode, with multiple threads acting as the driver workers, in the same JVM.

Here are the settings of the computer and the experiment:

- Processor: Intel(R) Core(TM) i5-6300HQ CPU with 4 cores @ 2.30 GHz (x64-based processor)
- RAM: 8.00 GB and 1TB HDD
- Spark version 3.0.1 and Hadoop version 2.7 and we assigned the mode to "local[*]" meaning Spark will use as many threads as possible for the given tasks payload.

Part 3: Experiments

For the following, let us fix the tenets and structure on which the tests were run:

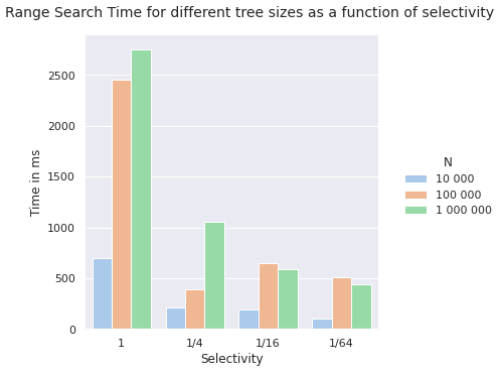
- Form the previous results on single/multithread, We chose to create an Rtree composed of Points, with $M = \frac{N}{20}$, $m = \frac{M}{2}$ (m minPageSize, M maxPageSize and N the number of records to insert). The QuadraticSplitter strategy was used as it does not influence that much the performance when compared to the LinearSplitting strategy
- The points were generated in a random range $[0, N]$ to better mimic the possible spread of a huge number of records, and the parameter will become fixed at some points in the tests.
- We used a number of partitions $p = logical\ cores * 3$ as recommended in the Spark documentation and we use time as the main performance metric for search operations, by performing each test time 10 times and computing an average of the total time. For each test, we will show the DAG of operations.

It is here necessary to introduce, at a high level, the 4 types of JavaRDD that will serve as data abstractions over the search tests:

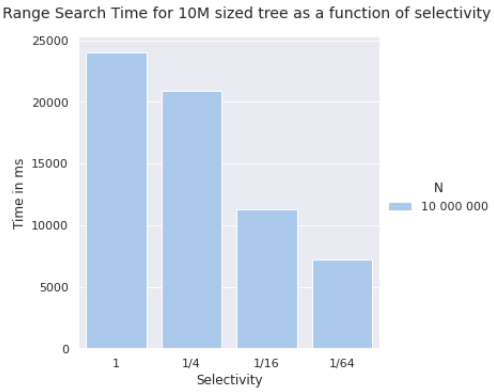
1. **The initialDataRDD**: This only represents the initial data records loaded into memory from a .txt file. Note that these are partitioned in a random way.
2. **The indexedInitialRDD**: This is a transformation over the initialDataRDD, and is essentially composed of an R-tree indexing each partition. Here also, the R-tree are given records that are randomly distributed across the cluster and we will see why this approach is not the best suited for spartial indexing.
3. **The spatialRDD**: This represents the initialDataRDD re-partitioned spatially across the cluster, meaning the data records will not be randomly distributed anymore, but grouped into partitions that are spatially related. We did this by sampling data from the initialDataRDD, constructing grids of the data-space, and re-assigning each record to its corresponding spatial grid, that will serve as re-partition structure for the spark partitions.
4. **The spatialIndexedRDD**: This is a transformation over the spatialRDD, and is essentially composed of an R-tree indexing each partition. However here the R-trees will be covering a subest of the whole data-space, and will provide a more efficient query speed since there will be less overlap due to the data spatial re-partitioning.

We also define the selectivity factor for range search queries, as $selectivity = \frac{queryMbrArea}{datasetMbrArea}$, with mbr being the minimum bounding box that covers all the records inside it. These approaches and concepts for spatial partitioning are taken from [21].

First, we compared the range search time for different tree size and selectivities to be able to select of number of records N that we will use for next tests.



(a) RangeSearch() as a function of query selectivity



(b) RangeSearch() for 1M of items

Since Spark operated a filter() operation to find records in a given range, it makes sense that the more selective (small) the range rectangle is, the faster it can determine appurtenance of records or not to a range. We can also witness that the more selective the query is, the more the size of the tree doesn't matter much (except for 10M of items which is a large factor slower than the others): we choose to follow the test with $N = 100000$, as it was still quite fast to get results and simulates quite accurately the real use cases of a R-tree.

Here is the DAG of operations for a simple record RDD range search:

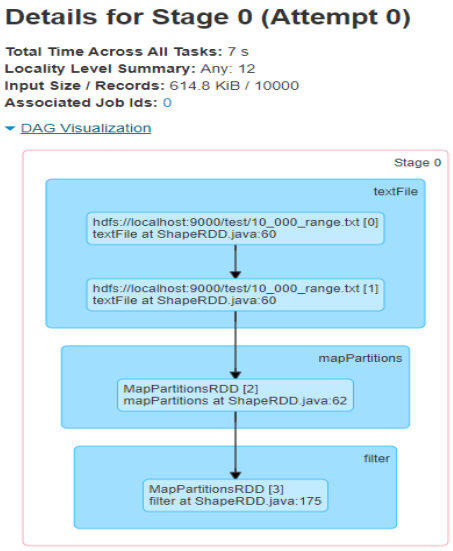
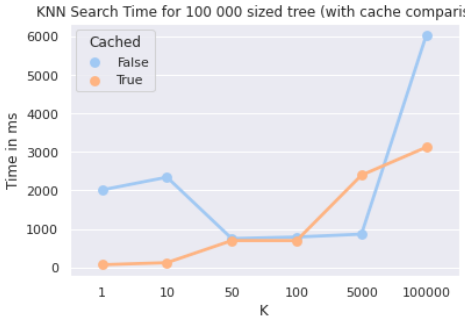
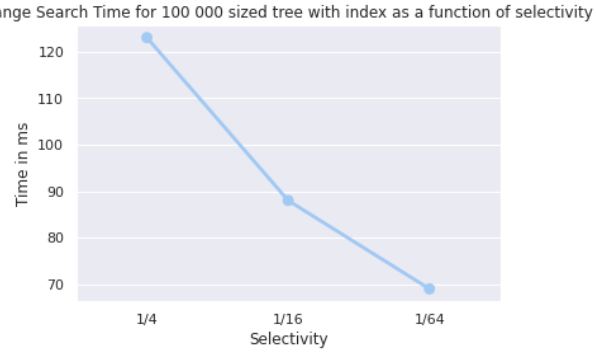


Figure 30: DAG of operations for the simple RDD rangeSearch()

We can also leverage another power of Spark to speed up the computations, especially given that we use it in local mode, where memory is shared across the JVM, which is **caching**. We can cache to different locations like the Disk or main memory, which allowed us to have a flexibility of changing the persistence location when testing with bigger datasets.



(a) Cache effect comparison on knnSearch() for initial RDD



(b) Range query search time for index initial RDD

We can witness the greatly improved query time for the query coming after the initial RDD load, and from now on, We will be caching each RDD transformations that we will use since the size of the datasets can allow us to do so. Now we compare with the same query on here the data records are indexed by an R-tree in each partition. Here is a range query, that we will use as comparison with future tests.

Here is the same KNN query with randomly distributed records, indexed by an R-tree.

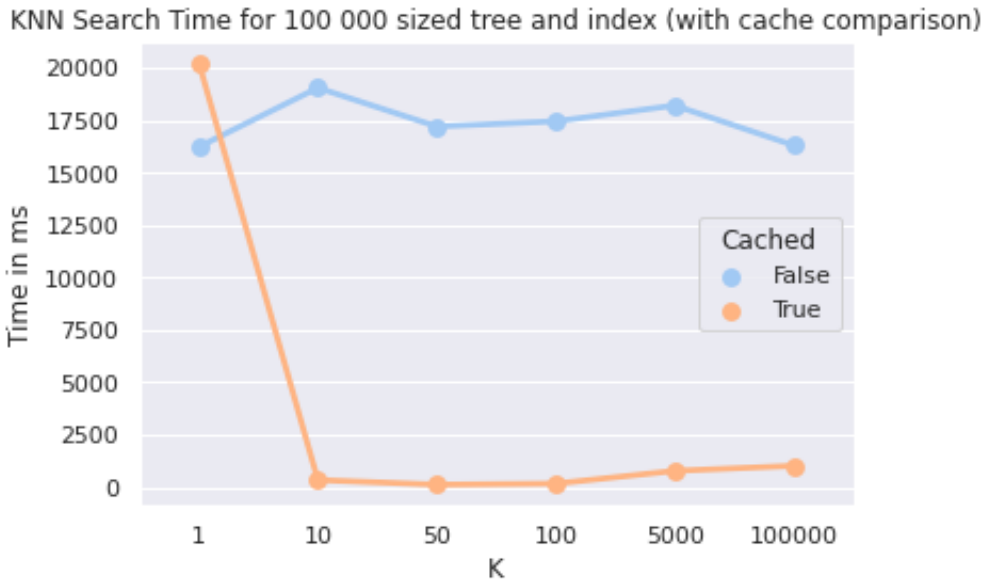
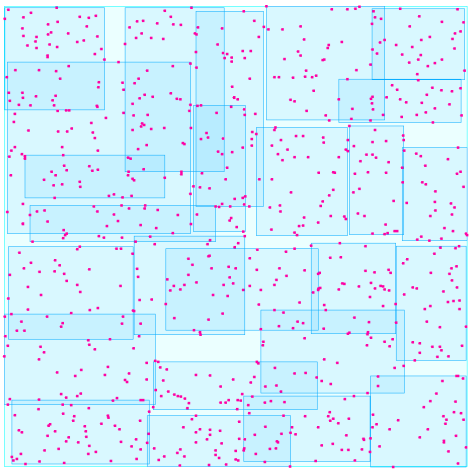


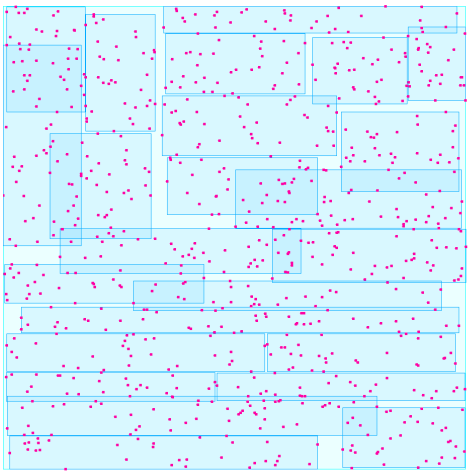
Figure 32: Cache effect comparison on knnSearch() for initial indexed RDD

With no indexing, the query time is increasing with the values of k, because the algorithm is a basic filtering, and traverse randomly distributed records with no spatial proximity. With indexed data, the records are also randomly distributed but are R-tree indexed, and the time for retrieving nearest neighbors as a function of k is more or less constant with K increasing, and this is because R-tree KNN filtering happens in a local way in the leaves that are spatially interesting and close to each other, and computing a distance is quite fast, compared to a range check in the previous case with not indexed RDD. We can notice that the overall time for not cached indexed methods is a lot slower than not cached not indexed methods, because indexing each partition with an R-tree is time consuming.

However, the times for cached methods are quite similar, and this is an issue since R-trees should allow the queries to execute faster on spatially-close items. This is because even if the records are spatially close in an R-tree, there are still randomly distributed across the cluster, so there is a huge overlap on the area that the R-trees are covering, let's take a look:



(a) Overlap example of R-tree with randomly partitioned records



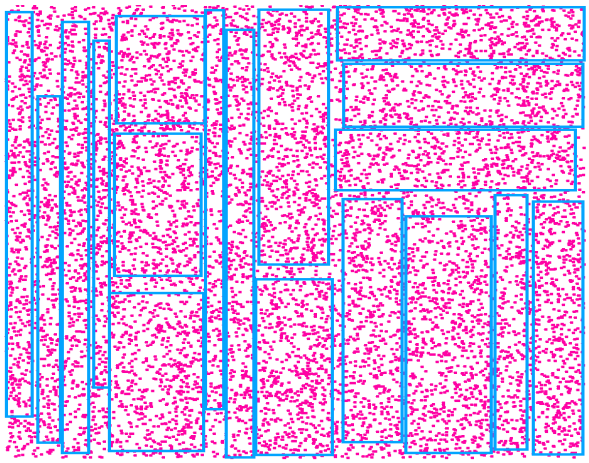
(b) Other overlap example of R-tree with randomly partitioned records

These R-trees are both on a different partition, but there are composed of randomly distributed records, so the search queries will still not be as much efficient as it could be because of this partition-dependant overlap. This is why we need to have a **spatial re-partitioning**.

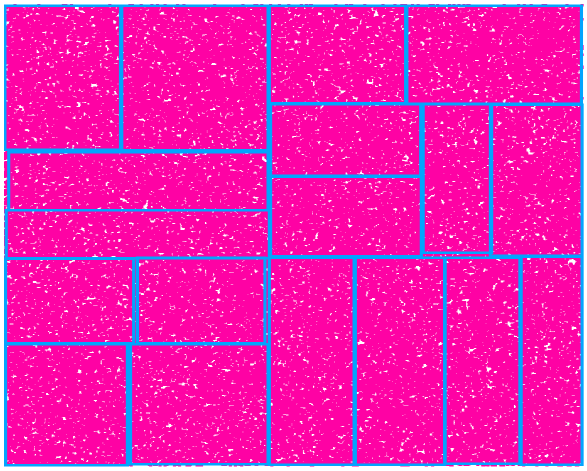
The approach is the following:

- We take a 5 % sample of our original data, and constructs spatial grids of our datasets, in a proportional way to our number of partitions.
- These grids are constructed by insert the sampled data into an R-tree, and selecting the leaf mbr's of this tree.
- We then re-assign each record to its corresponding grid, and keep each grid in a separate partition, to achieve both inter-partition spatial proximity, given that grids are composing the dataset space (spatialRDD), and intra-partition spatial proximity with the R-tree indexing structure in each new grid-based partition (indexedSpatialRDD).

Here is a visualization of the grids for 5 percent of the sampled data:



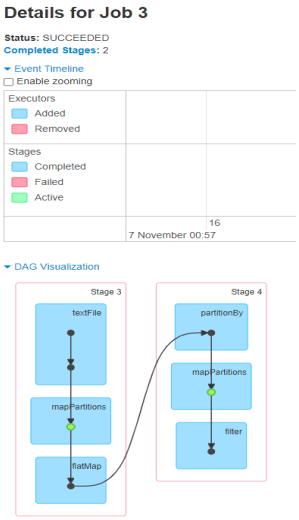
(a) Grid records coverage after spatial re-partitioning



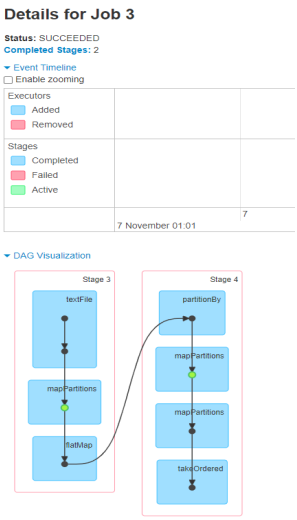
(b) Grid records coverage after spatial re-partitioning for 100K records

We can see that 5 % of the data can represent quite uniformly the data spatial re-partition. However, for all the records that are not in a bounding grid, we must keep an overflow grids to not exclude items from the partitions.

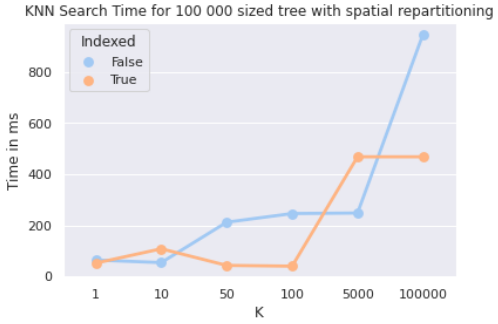
Once the items are spatially re-partitioned, we can build an R-tree index for each partition as we did before and compare the performance the between spatialRDDs and indexedSpatialRDDs, for both search methods.



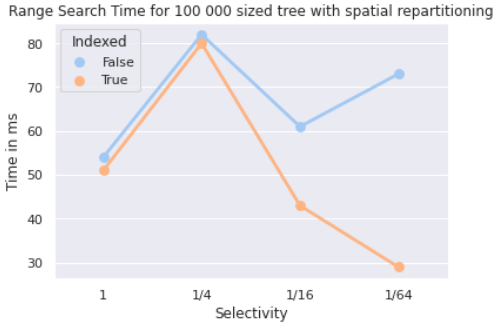
(a) DAG for spatial RDD actions



(b) DAG for indexed spatial RDD actions



(c) R-tree index effect for KNN search on spatial re-partitioned RDDs



(d) R-tree index effect for range search on spatial re-partitioned RDDs

For both methods, the spatialRDD version is faster on average than the randomly partitioned records. For range queries, the higher the selectivity, the faster the search is because of spatial proximity of the items inside the R-tree, and it also holds true for the not indexed data, even if there is a small factor of randomness. For KNN queries, it is the same case, which proves that spatially re-partitioned items are way faster to query in the case of spatially indexable data.

Let's now visualize 2 sample R-trees indexing the spatialRDDs, as well as as the overflow grid:

(a) R-tree index coverage for spatially re-partitioned RDD

(b) Another R-tree index coverage for spatially re-partitioned RDD

(c) Overflow grid records that were not assigned during re-partitioning

There is little to no overlap for the standard R-tree mbr's, and the overflow grid contains data records that are extending up to all the space of the data, but their quantity is quite low for 100 000 items, and having an overflow grid like this is a trade-off worth taking to re-partition spatially the records.

Here is a last comparison on the range queries with indexed re-partitioned data:

(a) Range Search latency for indexed re-partitioned records

We can completely see the difference in the search query times with no spatial partitioning and with: form of items, it takes about 2.6 seconds to search for items in a range with standard data, while it takes under 100 ms for spatialRDD. Of course these are extreme cases of our comparison set, but it shows how much the R-tree parameters and spatial re-partitioning can affect the query speed for our records.

5 Discussion

We've chosen to use fine-grained locking and coarse grained locking techniques for the multi-threaded implementations, one very interesting improvement we can implement for this project is the lock-free approach. For B-Tree results (in terms of time), we've observed a longer time of execution for queries in multi-threading that might be due to overhead during thread creations. Also, the fine-grained approach could give better results as well for the delete and insert operations, due to the hand-by-hand locking scheme.
It is the same case for the two others tree-structures where the multi-threaded approaches could be even more efficient enhanced by implementing a lock-free based approach. Being able to configure and spin up an entire dedicated cluster for Spark performance tests could have been an interesting option too.
Using random generated datasets was also a way to abstract the data generation and put a focus on the actual implementation rather than evaluating indexing for real-world uses cases.

As tree-based indexing algorithms were new for most of the students of our group, we have been able to acquire new skills and learn new things about data structures, the problematic of data storage and enhance our coding skills thanks to this project. Some students had already implemented Binary Search Trees and studied B-Trees theory over the previous years, so putting into practice these complex structures was really interesting.
The three types of implementations of B-Trees, B+-Trees and R-Tree were very challenging. Even after seen the theory and the advantages of each implementation at the start of the project, we've acquired a deeper understanding of why and when should we use single-threaded, multi-threaded and spark implementations, by coding them and testing the results.

References

[1] J Bercken and Bernhard Seeger. An evaluation of generic bulk loading techniques. VLDB, 2001.

[2] Anastasia Braginsky and Erez Petrank. A lock-free b+ tree. In Proceedings of the twenty-fourth annual ACM symposium on Parallelism in algorithms and architectures, pages 58–67, 2012.

[3] Axel Bruns. Faster than the speed of print: Reconciling 'big data'social media analysis and academic scholarship. First Monday, 18(10):1–5, 2013.

[4] Ping Chi, Wang-Chien Lee, and Yuan Xie. Making b+-tree efficient in pcm-based main memory. In Proceedings of the 2014 international symposium on Low power electronics and design, pages 69–74, 2014.

[5] Paolo Ciaccia and Marco Patella. Bulk loading the m-tree. In Proceedings of the 9th Australasian Database Conference (ADC'98), pages 15–26. Citeseer, 1998.

[6] Douglas Comer. Ubiquitous b-tree. ACM Computing Surveys (CSUR), 11(2):121–137, 1979.

[7] John L Furlani. B tree structure and method, September 22 1998. US Patent 5,813,000.

[8] Antonin Guttman. R-trees, a dynamic index structure for spatial searching. 1984.

[9] Edward R. Jorgensen II. Coarse-gr coarse-grained, fine-gr ained, fine-grained, and lock-f ained, and lock-free concurr ee concurrency approaches for self-balancing b-tree. UNLV THESES, 82019.

[10] Hosagrahar V Jagadish, Beng Chin Ooi, Kian-Lee Tan, Quang Hieu Vu, and Rong Zhang. Speeding up search in peer-to-peer networks with a multi-way tree structure. In Proceedings of the 2006 ACM SIGMOD international conference on Management of data, pages 1–12, 2006.

[11] Edward R Jorgensen II et al. Coarse-grained, fine-grained, and lock-free concurrency approaches for self-balancing b-tree. 2019.

[12] Dong-Ho Lee, Hyung-Dong Lee, Il-Hwan Choi, and Hyoung-Joo Kim. An algorithm for incremental nearest neighbor search in high-dimensional data spaces. In International Conference Human Society@ Internet, pages 436–453. Springer, 2001.

[13] LING LIU and M. TAMER ÖZSU, editors. B-Tree, pages 273–273. Springer US, Boston, MA, 2009.

[14] Mingyuan Ma, Sen Na, Hongyu Wang, and Jin Xu. Aegcn: An autoencoder-constrained graph convolutional network. arXiv preprint arXiv:2007.03424, 2020.

[15] Ohad Rodeh, Josef Bacik, and Chris Mason. Btrfs: The linux b-tree filesystem. ACM Transactions on Storage (TOS), 9(3):1–32, 2013.

[16] Hari Singh and Seema Bawa. A mapreduce-based scalable discovery and indexing of structured big data. Future generation computer systems, 73:32–43, 2017.

[17] V Srinivasan and Michael J Carey. Performance of b+ tree concurrency control algorithms. The VLDB Journal, 2(4):361–406, 1993.

[18] Xindong Wu, Xingquan Zhu, Gong-Qing Wu, and Wei Ding. Data mining with big data. IEEE transactions on knowledge and data engineering, 26(1):97–107, 2013.

[19] Lei Xu, Chunxiao Jiang, Jian Wang, Jian Yuan, and Yong Ren. Information security in big data: privacy and data mining. Ieee Access, 2:1149–1176, 2014.

[20] A. Nanopoulos Y. Manolopoulos and Y. Theodoridis. R-Trees: Theory and Applications. Springer UK, 2006.

[21] Jia Yu, Zongsi Zhang, and Mohamed Sarwat. Spatial data management in apache spark: The geospark perspective and beyond. Geoinformatica, 23(1):37–78, 2019.

[22] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In Presented as part of the 9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12), pages 15–28, 2012.

[23] Fahad Zaqout, Merza Abbas, and Al-Samarraie Hosam. Indexed sequential access method (isam): A review of the processing files. In 2011 UKSim 5th European Symposium on Computer Modeling and Simulation, pages 356–359. IEEE, 2011.

[24] Donghui Zhang, Kenneth Paul Baclawski, and Vassilis J. Tsotras. B+-Tree, pages 197–200. Springer US, Boston, MA, 2009.