Carl Otto Steen

The Prospect of Merging R-trees

Master's thesis in Computer Science
Supervisor: Svein Erik Bratsberg
June 2019

Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Norwegian University of
Science and Technology

Carl Otto Steen

The Prospect of Merging R-trees

Master's thesis in Computer Science
Supervisor: Svein Erik Bratsberg
June 2019

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Abstract

Spatial data are found in numerous applications and research areas. In recent times, there has been a rapid explosion in the amount of spatial data [7] generated by devices such as smart phones, satellites and geotagged sensors. This thesis mainly focus on the access method R-tree[19] as it is a general-purpose data structure, can handle a variety of spatial quires, and compatible with the massive volume of data. In this project methods and techniques for ingesting large amounts of rapidly produced spatial data into a R-tree are studied and elaborated. The process of merging separate data structures into a new index that cover, the whole data set, arise as an alternative to bulk loading and insertion. Bulk loading tend to rely upon pre-sorting the data to build the data structure, and bulk insertion may give worse query performance given the dataset and method implemented.

The variants of spatial data and R-trees are discussed in detail. This entails, How R-trees have adapted to tackle temporal dimensions, and what adjustments are necessary to extend R-trees to parallel or distributed environments. As we head into the next decade where more devices continue to produce spatial information with various associated information, the access methods for spatial data must be able to operate efficiently on clusters. The state of the art methods for parallelizing bulk loading R-trees on parallel systems are discussed. Also, the state of the art bulk insertion methods are examined.

A new merging method was devised, to bridge the advantages of bulk insertion. The thesis also include a comparative study of prominent methods from merging, bulk loading and insertion. The study outlines the performance characteristics of the various methods on real and synthetic datasets. Merging was found to promising under certain conditions. From the results, it is clear that many variables must be taken into consideration when constructing a R-tree for a specified application. Given the results, a learned cost model[50] may be the way forward.

Sammendrag

Romlig data benyttes i forskjellige applikasjoner og forskingsområder. I nyere tid har det vært en eksplosjon i mengden romlig data generert av smarttelefoner, satellitter og geomerketede sensorer. Denne oppgaven fokuserer først og fremst på R-trær siden de er indekser med et generelt bruksområde, kan håndtere en stor variasjon spørringer og er kompatible med enorme datasett. I dette prosjektet blir metoder for innsetting av store mengder data inn til et R-tre studert og utdypet. Flette sammen adskilte datastrukturer til en ny indeks, som dekker hele datasettet, oppstår som et alternativ til bulk lasting og innsetting. Bulk lasting pleier å benytte seg av sortering av data for å bygge datastrukturen, mens bulk innsetting kan gi dårligere resultater for spørringer, gitt datasett og metode implementert.

Forskjellige varianter av romdata og R-trær diskuteres i detalj. Dette inkluderer hvordan R-trær takler tidsdimensjoner, og hvilke justeringer som er nødvendig for å utvide R-trær til parallelle og distribuerte systemer. Nå som vi nærmer oss det neste tiåret, hvor flere maskiner fortsetter å produsere mer romdata med ulike typer tilleggsinformasjon, må aksess-metodene kunne operere effektivt over klustere. Dagens mest fremtredende løsninger, som innebærer å utføre bulk lasting parallelt over distribuerte(parallelle) systemer, diskuteres i tillegg til bulk innsettings metoder.

En ny flettemetode ble utarbeidet med fordelene fra bulk innsetting. Avhandlingen inneholder også en komparativ studie av fremtredende metoder fra fletting, bulk lasting og innseting. Undersøkelsen skisserer ytelsesegenskapene til de ulike metodene. Fletting viser seg å være lovende under visse forhold. Ut ifra resultatene er det klart at mange variabler må tas i betraktning når et R-tre skal bygges for en gitt oppgave. Dermed er en trent kostnadsmodel muligens veien fremover.

Preface

I would like to thank my supervisor Professor Svein Erik Bratsberg for providing code examples, inspiration and the support throughout this semester. A large portion of this paper is based on previous work done during a specialization project the preceding semester by the same author and with the same supervisor. A special thanks to my friends and fellow students at NTNU, whom I have discussed ideas with and learned from since the start of my degree. Among that group, I would like to thank Åsmund Brekke in particular for spell checking. I final thanks is necessary for the research community who specialize on spatial data and its applications.

Carl Otto Steen
Trondheim, June 11, 2019

Epitome

Chapter 1: Describes the rationale and the research goals of this project.
Chapter 2: Presents the background, context and future prospects of spatial data.
Chapter 3: Gives the reader the necessary background to understand R-trees and the following methods, in addition to surveying R-trees and how they can be adapted to meet new use-cases and challenges.
Chapter 4: Provides an examination of the state of the art solutions of bulk loading and bulk insertion.
Chapter 5: Describes Merging and the similarities between the methods.
Chapter 6,7: Includes a comparative study of different methods, their performance evaluation and discussions. In addition, the implementation and data sets are presented
Chapter 8: Conclusion and further work.

Contents

List of Figures

List of Algorithms

List of Tables

Chapter 1

Introduction

1.1 Problem Description

Spatial data is generated at an unprecedented rate. Imagine a data system with a continuous, high throughput stream of spatial data. Our goal is to minimize the latency of writing high-dimensional data to an index, while keeping sufficient query performance. Today the state of the art solution is to periodically rebuild the whole data structure by constructing sub-trees of the R-tree in parallel on a distributed environment. Building a new index from scratch given the distribution of the data, is known as bulk loading. To gain good query performance and space utilization, bulk loading usually relies on sorting the data according to a predefined ordering. Considering all the ways to ingest data into a R-tree, Bulk loading yields the best query performance. Many applications of big spatial data have no need for newly inserted data to be available right away for querying purposes. However, the downtime from bulk loading is incompatible with large scale, real time systems that require the ability to query the newest entries in the data system. Parallelizing construction operations significantly reduce the construction time, however the newest entries will not appear in the index until the next reconstruction, unless dynamic updates are supported. Another approach is bulk insertion, where the data is inserted into clusters, and in those clusters minor components are built and placed within the structure of an existing index. Bulk insertion goes a long way back and proposed methods from the two previous decades may not be the most optimal for the advancements and variety in the underlying hardware and growth of accumulated data. Another problem with bulk insertion is their professed inability to be effectively adapted to an R-tree spanning over several machines in a cluster. The task of solving this problem is rather complex, as we must have an acceptable query performance, in addition to having the newest entries available within a strict time frame. Merging separate R-trees into a new index may be an important tool for a possible solution.

1.2 Project Goal

Merging R-tees have received limited attention. The only notable work are from Vasilis Vasaitis [55],2004. The goal of this thesis is to evaluate if merging R-trees represents a valid part of the solution to the problem. Rebuilding an index from scratch is a costly operation and prevents queries on the access method. Merging R-trees can potentially reduce downtime of the system. Merging is similar to bulk insertion[44][45][52], insofar as both methods insert subtrees into a the covering R-tree index. Hence, many of the same heuristics used in bulk insertion can be

applied to merging, this will be demonstrated in the project by creating a new merging method. The purpose of creating a new method in this project is to illustrate the similarities in the design-space of merging, bulk loading and insertion, as well as taking advantage of known and exhaustively tested design solutions.

Research Questions:

- **Is merging R-trees a viable method?**
- **How does merged R-trees compare to R-trees created by bulk loading?**
- **Can we extend heuristics from bulk insertion over to merging?**

1.3 Scope

How merging can be extended to parallel system is beyond the scope of this thesis, but a natural next step. The reader will be presented with an up-to-date and holistic examination of R-trees, spatial data and decentralized approaches. Decentralized and parallel methods are unavoidable given the scale of the datasets. Since merging R-trees is an unexplored and perhaps promising alternative, competing equivalent methods will be thoroughly reviewed. In order to outline the prospects of merging, a comparative study of merging, bulk loading and bulk insertion was conducted. For the comparative study; two merging methods, the bulk insertion method with the best performance[52], one by one insertions and a prominent bulk loading method[47] was implemented. Performance is measured on datasets with various distributions and dimensionality.

Chapter 2

Background and Motivation

2.1 Spatial Data

Spatial data can be found in cartography, multimedia, computer aided design, targeted advertising, climate science, transportation, criminology, maritime control, augmented reality games, geo-social networking, online check-in services, location or route sharing applications, monitoring of movements, air quality inference and predication, evacuation route planning, monitoring of critical infrastructures, surveillance, computer vision and robotics. The purpose of spatial data is to represent concepts like; the location, size and shape of objects such as a buildings, lakes, mountains or cities. Spatial data is utilized in many of our everyday tasks and recreational activities. Google Maps is answering millions of queries per day. The "search this area"- functionality supports querying places like shops or restaurants. Games based on augmented reality has gained increased popularity in the recent years. For instance Pokemon GO has over 30 million active users[13]. The data is being generated in an ever-increasing rate. Sentinel 1 and Sentinel 2 from the Copernicus Programme, conducted by the European Space Agency, Generates about 2,6 Terabyte of images per day [29]

2.2 Spatio-Temporal Data

Spatio-temporal data extends the spatial data with a temporal dimension. For instance, in the case of neuro-imaging data, activity measured from the brain is stored along side the spatial location from which the activity was observed, in addition to the time at which the measurement was made. Thus, enabling us to discover temporal and spatial patterns of human brain activity from neuro-images [18]. Given a temporal dimension, the motion history of a moving object can be captured. Let us define a trajectory of an object O to be the function $T_O(t)$ which returns the position of O given any time t. However given a large set of objects and a fine-grained time axis, storing $T(t)$ for every object and time instance is rather infeasible. $T(t)$ is approximated by a finite set of observations at discrete timepoints t_1, t_2, t_3, ... t_n. Thus, spatio-temporal trajectory data-sets consist of temporally ordered lists of locations visited by the moving objects in a multidimensional space. These data-sets tend to get very large and grow at a rapid rate. With the recent proliferation of devices with GPS (Global Positioning System), sensor technologies to collect positional data and the integration of sensors with location information of devices in

upcoming "Internet of things"-Era, [30] requires new approaches to handle storage and query processing [18].

2.2.1 Methods for Building trajectories

Suppose we have vehicles equipped with GPS devices that periodically transmit their positions to a central server. At the server side, the data is processed, stored and ready to be utilized. In order to track the movement of a vehicle, the position must be reported continuously. Yet, GPS can only obtain the position at discrete time, in intervals of seconds. In order to represent the movements of objects we have to store the position samples and interpolate in-between the sampled positions. The simplest approach is to use linear interpolation, as opposed to other methods such as polynomial splines. These outlined positions can be represented as a sequence of line segments that make up the trajectory for a specific object. So, the trajectory segment for an object traversing in a n-dimensional spatial space becomes a line in a n+1-dimensional space, with the temporal aspect as an additional dimension. To detect if an object has entered an area is only available after one or more trajectory segments are examined. Trajectory segments may falsely suggest an entity has entered an area when it has moved around it. Another approach is to build trajectories based on updates regarding changes in position, velocity or direction. Resulting in that moving objects only send their information when data is changed. If we continually collect GPS sensor readings from 100 000 vehicles who transmit their location every minute, then our data set will increase by 144 000 000 new segments every day, assuming every vehicle change its position every minute[54].

2.3 Spatio-Temporal-Textual Data

Spatio-temporal data frequently occur with associated textual content. Usually a location in a trajectory is called a *point of interest(POI)*. Point of interests commonly consists of the following features; latitude, longitude, name, and other text description like activities [9]. Location based applications like Foursquare[6] enables users to search for points of interest in their surrounding area, like restaurants, nightlife spots and shops. When querying the area, personalized recommendations like a user's check-in history, their profile and their venue ratings, in addition to the time of day are integrated to retrieve the most relevant results. The impact of geo-location in social networks has drastically increased, enabling marketers to focus their products based on trending topics within a region of interest [42].

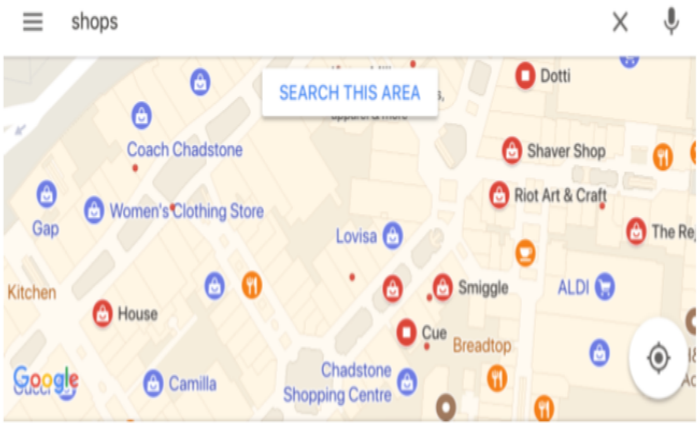


Figure 2.1: Points of interest; Google Maps [24].

2.4 Querying Spatial data

Spatial data are represented in a space that has more than one dimension. The data can be stored as coordinates, points, lines, polygons or regions. The Spatial relation between two objects adds new level complexity that a traditional relational database is unable store or query efficiently. Spatial data calls for a different access method to handle multi-dimensional data, introducing new types of queries such as:

- **Spatial selection:** Return all entries that satisfy a spatial predicate
- **Spatial Join:** Joins entries based on a given predicate on their spatial attribute values
- **Nearness queries:** Return entries that lie close to a specified location in the n-dimensional space
- **Nearest neighbor queries:** Find the k nearest neighboring entries of a given point or an entry
- **Region queries:** Return entries which belong partially or fully inside a given region, also known as window queries that search for entries within or intersect a given query window which is usually a rectangular region of interest.

These queries come in handy for locating hotspots(spatial clusters), spatial outliers, building predicative models and finding data points that tend to co-occur on a map.

2.5 Querying Spatio-Temporal Data

Spatio-temporal data need vast amount of storage due how the objects vary across the space and time domain. Spatio-temporal data are used in real-time systems like traffic control systems, location-based services and geographical information systems, therefore requiring fast processing time. Consequently, the cost of I/O and computation is high, making good indexing and storage schemes crucial.

The integration of the temporal aspects of data adds additional query types that can be put forward by a user. Queries may focus on a specific time instance(time-slice queries) or time interval(time interval queries). These two types can be combined with spatial predicates and the ability to query past, present or future positions of the objects. In other words, spatio-temporal queries cover trajectories, the movement of points or regions. Spatio temporal data mainly grows in the temporal dimension, and is regarded to be append-only along the temporal axis. The temporal axis can be unlimited. In other words, the confines of the spatial information tend to remain constant or expand slowly as the data set grows, while the temporal dimension is continually increasing. If moving objects update their locations too frequently, storing the historical data of moving objects, in it's entirety, might no be a feasible approach because of the massive generated volume[8].

2.6 Querying Spatio-Textual Data

In recent years there as been a flourish of location-based web applications such as online check-in services, route and location sharing. The common aspect for these applications is the way that locations are combined with semantic meanings. Spatio-temporal queries can be extended to handle activity trajectories[26], where each place can be associated with a set of activities performed by the users. Finding trending terms on social media platforms for a given spatio-temporal region, is also possible. New queries have been introduced, such as:

Activity Trajectory Similarity Query: is defined by an activity trajectory set A and a query Q. An Activity Trajectory Similarity Query will return a number of distinct trajectories from A that have the smallest minimum match distances with respect to Q. Note that activities must be evaluated in terms of both distance measures and the search process. Finding similar activity trajectories to a query is helpful for place recommendation and trip planning. In order to handle the huge information based on location, distance and activities, trajectory search are based on distributed index, usally on a MapReduce model [10][20].

Top-k Frequent Spatio-temporal Terms Query: For a spatio-temporal region, find the k most frequent terms posted by users in the region. Presume that if a user wants to know which terms have been popular around his current location over the past week, the query can specify a spatial circle with radius 10 kilometers and a temporal interval of a week[43].

Chapter 3

Theory

3.1 Access methods and Design Tradeoffs

An access method is categorized as a collection of algorithms and data structures for organizing and accessing data. For more than four decades, access methods have adapted to changing hardware and workload requirements. Even small changes in workload and hardware can cause complete redesign of access methods. The goal when designing an access method is to minimize read time(R), update cost (U) and memory overhead (M), commonly known as the **RUM overheads**[33]. However, optimizing in any two areas deteriorates the third. A tradeoff has to be made when we consider an access method for a data system. Access methods for high-dimension data have the same RUM tradeoffs. Also, they tend to have difficulties quantifying the read cost and rely on experimental estimations.

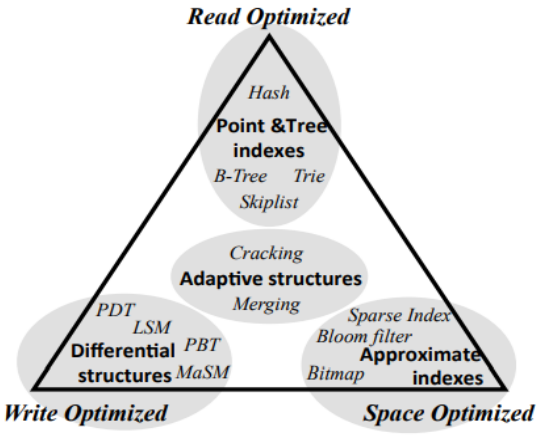


Figure 3.1: Where popular data structures fall within the RUM space. [33]

3.2 Limitations of the B+-tree

B+-trees have been considered the standard access method in all prototype and commercial relational systems for transaction processing applications [56]. However, the B+-tree is a key

value storage that is restricted to alphanumeric one-dimensional data. Consequently, B+-trees without a good mapping scheme are not very suitable for indexing spatial data. The desired index structure should be able to retrieve an object efficiently based on it's position in a n-dimensional space. The index should support n-dimensional range queries, for instance every point that lie within a particular radius of another. The local similarities of the n-dimensional objects should be preserved. In other words, objects that lie close together in the n-dimensional space should keep their similarity properties when stored on the index.

3.3 R-tree

R-trees were introduced in 1984 by Guttman [19].They are ideal for proximity queries like window queries, nearest-neighbor queries or spatial join queries[53], in addition to minimizing down-time of the structure when loaded with new data. This allows multiple processes/threads to simultaneously query the index while new data is inserted. R-trees are height-balanced just like it's one-dimensional counterpart(B+-tree). An height-balanced tree will automatically minimize its height, the number of connections from the root to leaves, regardless of what items are being inserted or deleted. All leaves are on the same level of the tree. Similarly to the B+-tree, the records are kept in the leaves of the tree. The records in the leaf-nodes each contain an n-dimensional rectangle(hyper-cube), called Minimum Bounding Rectangle(MBR), that has the minimum required size to contain the polygon or point of the spatial object and a pointer to the spatial object. Any arbitrarily shaped objects can be handled by operating with their minimum bounding rectangles. A R-tree node usually corresponds to a disk block. Every node in the tree has a designated MBR of the n-dimensional space. If the node is not a leaf, it will have pointers to other nodes that cover a smaller area of the MBR contained within this node. MBRs have to circumvent all the MBRs of their children. There are several RUM tradeoffs with regard to the R-tree, higher overlap between MBRs makes updates cheaper, but as a result, reads become more expensive. R-trees can be extended to handle spatio-textual and temporal data. For example, in order to execute Top-k Frequent Spatio-temporal Terms Queries, a sorted term list can be attached to each leaf node of the R-tree, where each list has pairs of terms and frequencies[43].

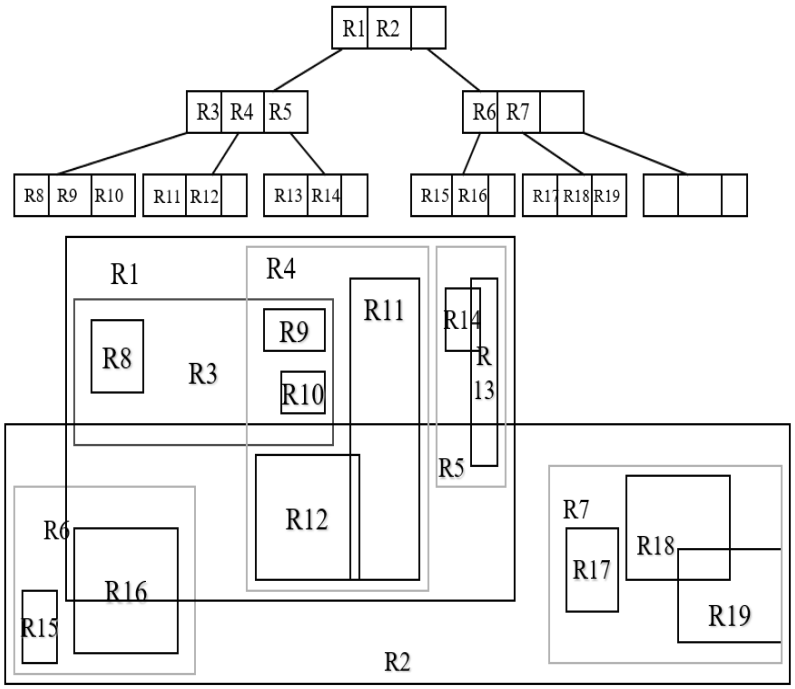


Figure 3.2: R-tree

3.3.1 Search

Searching starts from the root of the tree and descends in a manner similar to a B+-tree. However in every non-leaf node any number of sub-trees might be a valid candidate and needs to be checked. Thus a worst case performance can't be guaranteed, nevertheless irrelevant regions of the indexed space are not considered. To optimize the query performance we want the area spanned by the nodes in the n-dimensional space to be as small as possible and non-overlapping. Search performance is highly depended on the structure of the index. The arrival rate of data, skewness of the data distribution, if the tree needs to be static, what heuristic is used for deciding which MBR to place the entries, what algorithm is used when bi-partitioning set of MBRs in case of a split [63], and whenever or not some entries can be reinserted back into the tree, they all contributes to the structure of the tree. The R-tree shows good search performance if the search query causes traversals of only a few paths within the index.

3.3.2 Insertion & Packing

Simple Insertion

Inserting entries into an R-tree is analogous to insertions in a B+-tree. However, we are considering keys with more than one dimension. Nodes are traversed from root to leaf by selecting a path with an associated MBR that require the least enlargement. If two or a number up to M MBRs require the same enlargement, where M is the maximum number of elements allowed within a node, then the MBR with the smallest area is selected. The entry is inserted into a leaf and the

spans of the MBRs along the path from the root to that specific leaf are updated according to the enlargement of the MBRs. The entries of the node will be bi-partitioned into two new nodes if the selected leaf cannot contain the new entry because it would exceed M entries. If the new nodes cause overflow in the parent node, then the process is repeated and may propagate all the way upwards to the root. Splitting nodes in R-trees requires more complex heuristics than that of a B+- tree. The structure of the tree is non-deterministic as the correct path to place a new entry is dependent on the current MBRs that are frequently updated with new entries. This property makes the task of quantifying the read cost with accuracy rather difficult. In most R-tree variations, linear worst-case query time complexity cannot be avoided and we must rely on experimental estimations to test the performance.

3.3.3 Bulk loading R-Trees (packing)

Suppose that the spatial data set is known beforehand and that insertions and deletes rarely occurs. In that event building a static R-tree might be more favorable than having a particular dynamic R-tree which mainly relies on inserting one item at the time [57]. Given that every entry in data set is known, the entries are used to built a new index, the process is referred to as bulk loading. Trees that are built by bulk loading are usually static. Static trees require the whole tree to be rebuilt every time an insertion of a spatial object is needed, which makes queries impossible during the time the index is being rebuilt. Inserting items one at the time can cause excessive space overlap between the MBRs and make them cover spatial areas with no or few entries. Experimental results from [57] shows that the average space utilization of dynamically maintained R-trees is between 50% and 70%, while most bulk-loading algorithms are capable of obtaining over 95% space utilization [28]. Bulk loading R-trees usually relies on space-filling curves which is a locality preserving transformation of N-dimensional data down to a one-dimensional space. The ordering of the data depends on the position on a continuous line sweeping through the whole N-dimensional space exactly once. There are two ways to create an index by bulk loading.

- **Bottom-up:** The data is sorted by a criterion that preserves spatial proximity, generally the position an entry has on a space filling curve is used. The leaves are first constructed by grouping records with close spatial proximity to construct optimal MBRs. the upper tree levels are constructed by the same procedure, creating larger MBRs, until every entry is covered by the root's MBR.
- **Top-down:** Given the distribution of the data set, partition the space according to a splitting decision policy. For each partition continue to repartition until the node capacity requirements are satisfied. These types of methods tend to greedily construct sub-trees. The rationale for this approach is the fact that overlap in the top levels of the tree is more critical than overlap between leaves[62].

Sort-Tile-Recursive (STR): [47]

STR is a bottom-up bulk loading method that is not based on space-filling curves, and yet outperforms space-filling curves for mildly skewed or uniform data [56][47]. The idea is to tile the data space into slices, so that each slice contain the same number of entries. Suppose we have a k-dimensional dataset with r spatial objects, resulting in $P = \lceil \frac{r}{M} \rceil$, where P is the number of leaf nodes. The idea is to first sort the objects based on their coordinate on of the k axes, and divide the objects into S slices, where $S = \lceil P^{\frac{1}{k}} \rceil$. Each of the S slices will consist of runs with $\lceil MP^{\frac{k-1}{k}} \rceil$ consecutive objects. The last slice can potentially have less than$\lceil MP^{\frac{k-1}{k}} \rceil$ objects.

Each run is processed recursively using the remaining k - 1 coordinates. Suppose P=27 and k = 3. The entries will first be sorted on one of the k dimensions, then divided into 9M size runs. The runs will be sorted on one of the two remaining dimensions, then group into consecutive runs of size 3M and sorted on the last dimension.

3.3.4 Deletes

Deletes are handled similarly to insertions. A search is needed for finding the element that is to be deleted. It's worth noting that several branches of the R-tree need to be checked in order to find the appropriate element. Again we see that a better structure improves the performance of this operation. Deletes from a leaf might lead to a cascade of updates to the parent MBRs, because the entry might lie on the border and it's removal changes the span of the MBR. The nodes in the tree have a minimum fill-degree. A node cannot have less than m elements, unless the node is the root. If a delete cause a node to have fewer than m elements, then a node might redistribute elements from a sibling, merge with a sibling or reinserts all remaining elements. Due to the multi-dimensionality of R-trees, there is no clear concept of adjacency as seen in B+-trees. So the remaining nodes are not merged with the same simple heuristics as B+-trees, which makes reinsertion of the elements the most frequent implementation.

3.3.5 Splitting

Given that a node is full, we want to find the best way to divide M+1 elements among two newly generated nodes so that they are not needlessly examined during searches. More formally the problem is; given any query, create a bi-partition that minimizes the probability of invoking both nodes.
Guttman proposed three algorithms in his original paper[19] with respectively linear, quadratic and exponential complexity. It should be noted that, in Guttman's original paper the results did not suggest any significant performance gap between linear and quadratic split. On the other hand, in the results from the original R*tree paper from Norbert Beckmann [40], the authors found that quadratic split outperformed linear split. The three most common splitting methods are described bellow. [56]

Linear Split

Begin by choosing two spatial objects that are furthest apart from each-other as seeds for the two nodes. Then consider each remaining object in a random order and assign it to the node requiring the smallest enlargement of its respective MBR, thus minimizing the sum of area for the two nodes.

Quadratic Split

Begin by choosing two spatial objects as seeds for the two nodes, where these objects if put together create as much dead space as possible. Dead space is the space that remains from the MBR if the areas of the objects contained are subtracted. Until, there are no remaining objects unassigned, select and insert the object for which the difference of dead space if assigned to each of the two nodes is maximized in the node that requires less enlargement of its respective MBR.

Exponential Split

All possible groupings are exhaustively checked and the best is chosen with respect to the minimization of the MBR enlargement. This method is guaranteed to find the optimal partition, while the linear and quadratic are approximations. However, this method is CPU-intensive and less used in practice.

3.4 R*Tree

In 1990 R*-tree was introduced [40]. R*-trees cost somewhat more to construct than standard R-trees, as the elements that are furthest away are getting reinserted during overflows, this process is described as *forced re-insertion*. Forced reinsertion moves objects between neighboring nodes and hence decreases overlap. As a side effect, storage utilization is improved. Due to more restructuring, less spits occurs. The R*tree takes additional metrics when splitting a node. Making it ideal to illustrate the trade-off between choosing different combinations of heuristics that can be optimized when splitting a node. The task of evaluating to what degree a heuristic should be followed is very complex way, due to they all influence each other.

The area covered by a MBR should be minimized
This lowers the number of paths check during queries but, gives a less degree of freedom when inserting new MBRs which leads to worse storage utilization

The overlap between MBRs should be minimized
Trees that contain MBRs that have a high degree of overlap expect a larger number path traversed during queries. Thus overlap should be minimized. Similarly, as the previous heuristic, this constrains the shape of the MBRs, thus lowering storage utilization. Another problem is that smaller overlap between MBRs makes the tree less update-friendly

The margin of a MBRs should be minimized In this context the margin is defined as the sum of the length of the edge in the MBR. This results in the rectangles becoming in a sense more quadratic. As a consequence, the MBRs are more easily packet together and the area of upper levels, near the root, are minimized. Unfortunately, more quadratic shape of the MBRs causes more overlaps between them.

Storage utilization should be optimized
The goal of storage utilization is keep the height of the tree low. By having a compact and short tree the path from root to leaf is minimized, thus increasing performance during queries.

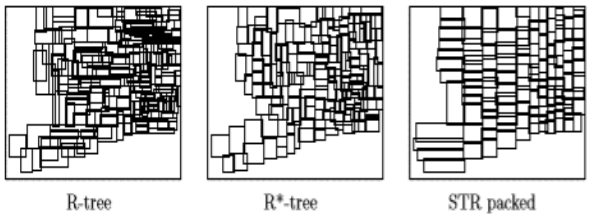


Figure 3.3: Resulting MBRs [56]

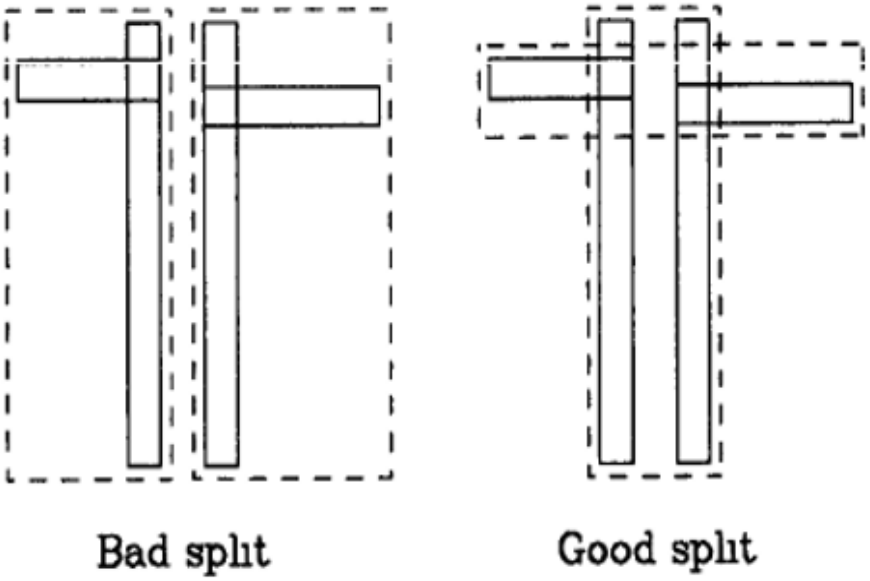


Figure 3.4: Good split and Bad split, as a result of different heuristics [40]

3.4.1 R-trees for Spatio-Temporal data

It is possible to treat the temporal aspect of the data-objects as an extra dimension and continue to use any type of spatial R-tree as an access method. However, the naive method, assuming that time is an additional dimension rarely generates simple and efficient solutions. A lot of empty volume can be created since objects that have a long lifetime correspond to very large MBRs, which in turn lead excessive overlapping between the MBRs and decreased query performance. Storing entries together according to their spatial properties(proximity), may not be optimal if some of the entries correspond to the same object, but on different time-steps. Queries might be interested in the whole trajectory of the moving object and be able to make predictions on further movements. In addition to the spatial properties, entries should be stored in association with the trajectories they belong to(trajectory preservation) as seen in STR-tree[17]. Indices for trajectory data are usually separated into three groups[48][35]:

- **Multi-Version Structure Methods**, where timestamps map to a spatial index structure and unchanged nodes are shared between each version. This strategy mainly focus on the temporal dimension and secondly spatial dimensions. A new R-tree or a logical representation is generated each time an update occurs, taking it's toll on storage cost. Notable examples are the HR-tree[34] and the multi-version 3-dimensional R-tree(MV3R-tree)[61], which addresses some of the shortcomings of the HR-tree and the 3D-Rtree considering space and time requirements. These indices are efficient with time slice queries, retrieving all objects that are within a window on the temporal dimension.

- **Spatial Partition Methods**, where trajectories are first grouped into respective spatial partitions and then a temporal index is created for trajectories in each partition. This strategy mainly focus on the spatial dimensions and secondly the temporal dimension. Partitioning the space frequently causes high query efficiency. The spatial extent of the data set needs to be known beforehand to make these methods a viable approach, nonetheless

problems with a skewed data distribution may arise. Notable examples are Scalable and Efficient Trajectory Index(SETI)[54], TB-tree, Polar-tree, Chebyshev Polynomial Indexing

- **R-trees with augmentation in temporary dimensions**, this approach manages both the spatial and temporal aspects as separate equal dimensions. The index structure can adaptively adjusted based on the data distribution, However the structure will deteriorate over time, in addition to drawbacks previously mentioned. Notable examples are the STR-tree, RT-tree, 3D R-tree.

3.5 Concurrency And Parallel Systems

This section describes how to adjust the R-tree to parallel architectures. Latches are often referred to as lock in the literature. In the following section, latches describes locks placed on data structures to limit access to shared data for threads, while locks are meant to limit access to records. The purpose is to create dynamic structure that support fast insertions and deletions of data without hindering existing processes that are concurrently using the index structure for querying purposes.

3.5.1 Concurrency control for R-trees

There are several concurrency control algorithms(CCAs) that are used for B+-trees that are not applicable for R-trees because of the increased complexity involved in the update operations. When an object is inserted into or deleted from a leaf node, the corresponding MBR must be correctly adjusted to updated boundary that complicates the concurrency control algorithms. Concurrency control algorithms for R-trees are based on latch coupling, breadth-first search, latch-mode conversion, and scope techniques[23]. Latch coupling and breadth-first are used to prevent deadlock. latch-mode converting and scoping are applied in order to lock the fewest exclusive nodes. Scope marks the nodes that may split or be deleted. latch coupling serialize concurrent operations that share a path in an R-tree. There are three types of latches with regard to latch coupling: read-latch (r-latch), write-latch (w-latch), and exclusive-latch (e-latch). An r-latch are compatible with another r-latch, while w-latches and e-latches are not compatible with latches of the same type. r-latches and w-latches are compatible with each other, while e-latches are not compatible with any other latch. In other words the e-latch gives a thread exclusive control over a node. Latch-mode conversion are used in order to give a thread a more restrictive latch on a node, i.e from a simple write-latch to an exclusive one, the reverse operation is also possible.

Concurrency Measures for Insertion

The first step is to search down the tree until the appropriate leaf is found. If any node along the path may overflow, a w-latch is placed on that particular node and the parent as the MBR of the parent may need to be updated if a new node is generated. If a node contains M elements then the node will overflow if the nodes along the path down to the leaf and the leaf are full. When an entry is inserted into a leaf the w-latch on the leaf is converted into an e-latch. If the leaf is not full it safe to release the w-latches on the parental nodes held the thread. However, in case of overflow the w-latches of the parents are converted into e-latches as nodes are updated in a bottom up manner. This phase is also applied in B+-trees. nevertheless, a reconstruction phase of the MBRs are required as the total volume/area of the MBRs may expand all the way up to the root.

Encountering Deadlocks

Suppose we have two nodes A and B, where A is the parent of B. If we assume that thread t_1 has a r-latch on A and wants to gain a r-latch on B. While t_2 on the other hand has an e-latch on B and wants to convert the w-latch on A over to an e-latch. Given this scenario, a deadlock has occurred as t_1 cannot have a r-latch on a node that has an e-latch, while t_2 cannot converts it's w-latch to an e-latch if there are other latches on the node. A possible solution is for t_2 to convert it's e-latch on B over to a w-latch, so that t_1 can place it a r-latch on that node. After t_1 has released all it's r-latches, t_2 converts the w-latch on A over to an e-latch.

3.5.2 On the scalability of Concurrency

Exploiting parallelism has several difficulties because of the complexity of coordinating competing accesses to shared data for threads. Often low overhead concurrency control and other latch-free techniques are used for increasing the scalability of the system. Latch-free algorithms tend to outperform their latch-based counterparts when the number of OS contexts exceeds the number of cores in the system. Latch-free algorithms can for instance consist of compare-and-swap(CAS) operations. CAS compares the contents of a given memory location with a given value and, only if they are the same, modifies the contents of that memory location to a new given value, if not the thread might be restarted. The trend in modern database systems, main-memory database systems in particular, move towards to a process model that has a one to one mapping between OS contexts and processing cores, limiting the performance gain of latch-free algorithms. Instead they are subject to the same types of synchronization overheads as the latch-based counterparts.[25] One can make the argument, that scalability of concurrency are based on the ability to avoid contention on global memory locations. If several threads or processes read and write to the same location, then severe overhead might occur, especially if the location is on disk.

3.5.3 Parallel Database Management Systems

Considering that a modern application demands more resources and may have multiple users operating the system simultaneously, the system hosting the application must be able to scale up to meet the required performance. Parallel or distributed database systems are spread out across multiple resources in order to improve parallelism and increase availability. Through parallelism performance is increased considering throughput, storage capacity and latency. The data can be scattered between multiple processor, disks and even data centers but, still appear as one logical database. There are three main variants of parallel systems, although it is worth to mention hybrid schemes like shared virtual memory

Shared everything:
every processor shares the same memory and disks. Communication is done by reading from and writing to a global memory.

Shared disk:
every processor shares a common storage area but each has its own exclusive memory.

Shared nothing:
every processor has it own partition of disk memory. Communication is done by message-passing. Best approach given cost efficiency.

The **shared nothing** parallel architecture looks somewhat identical to loosely interconnected distributed systems. still, distributed and parallel systems are not interchangeable. on distributed systems, nodes can be far from each other, connected using public network and problems of unreliable communication may arise. Whereas parallel systems have nodes that are physically close to each other, connected with a high speed local area network and communication cost is assumed to be small and reliable. There are three forms of parallelism used by parallel systems [32], they are defined as:
Inter-query parallelism which enables the parallel execution of multiple queries generated by concurrent transactions.
Intra-query parallelism makes the parallel execution of multiple, independent operations within the same query.
Intra-operation parallelism, the same operation can be executed as many sub-operations using function partitioning in addition to data partitioning.

3.5.4 R-tree Variants On Parallel Systems

Given large enough data set a single sequential R-tree on a uni-processor with one disk may provide insufficient response time. The R-tree access method most be modified in order to support data intensive applications on parallel systems. These variants try to exploit multiple resources like, disks and processors, to gain more efficient processing and support multiple users.

Multiplexed R-tree, one processor & multiple disks

The multiplexed R-tree[22] consists of one processor operating over several disks. It's considered a useful strategy for I/O bound applications by taking advantage of the I/O parallelism, and also enabling storage of huge amounts of data that do not fit in a single disk. In addition to requiring good heuristics for splitting nodes, the index must also have a policy for finding the optimal assignment of nodes to disks. How the data are distributed reflects in the read and write-performance of the index. Given a set of queries, it is desirable to have the workload spread evenly among all the disks. For smaller queries, in terms of span of the spatial data, the most promising solution is to activate as few disks as possible in order to best support concurrent users. Whereas for queries that have a large span, parallelism should be maximized by using a large number of disks. The metrics for the two scenarios is defined as minLoad and uniSpread[22].

- **minLoad** Queries should impact fewest nodes as possible, imposing a light load on the I/O sub-system. in other words, queries with small search regions should activate as few disks as possible.
- **uniSpread** Nodes that qualify under the same query, should be distributed over the disks in the most uniform manner. in other words, queries that retrieve a lot of data should access as many disks as possible.

the multiplexed R-tree achieve minimum load by having a sufficient number of disks. Uniform spread requires a good placement policy when assigning nodes to disks. Ibrahim Kamel [22] found that assigning a new node to a disk with the most dissimilarity with other R-tree nodes was the most promising. This placement heuristic is based on proximity indexing.

Master-Client R-trees, shared nothing architecture with one disk per machine

Master-Client R-tree(MC-Rtree)[14] is an improvement from the prior Master R-Tree(MR-Tree) index [36]. The idea behind the MR-tree is to have one dedicated master machine containing

all the internal nodes of the parallel R-tree, while the leaf nodes are distributed across several machines. The lower levels of the parallel R-tree store a reference with (MBR,site-id,page-id) for each of the leaf nodes. The site-id is used to locate a machine and page-id is for referencing a specific page. For every query a list of site-id and page-id pairs is generated. For each pair a request is sent to the corresponding machine.
For MC-Rtrees on the other hand both master and client machines have an R-tree as index structure. The master only need to store the client ids and the MBR of the data nodes, whereas the data nodes are kept in a corresponding R-tree on the client side. A major advantage is that only one request per client is required. The building of the R-trees for the master and the clients is done by packing and done in parallel. The choice of placement policy for the data was not significant determining factor in response time for the queries[14] in contrast to one processor & multiple disks systems [22].

3.6 R-trees incorporated in Log Structured Merge Trees

3.6.1 LSM-Tree

The Log Structured Merge Tree (LSM-tree) is an ordered, persistent index structure supporting multiple operations like insert, delete, and search. The LSM-tree has become the default storage structure in NoSQL databases such as Apace Cassandra[2] and Google Bigtable[1]. Embedded data stores like LevelDB[3] and RocksDB[5] are LSM based. The index structure is designed for write-intensive workloads, and thus performs exceptionally well for frequent updates of high volume data. The general idea is to first batch updates in memory before writing them to disk, thus amortizing the cost by having fewer and larger sequential writes rather than many and smaller random writes. Entries are first placed into an in-memory residing component where updates are performed in-place. When the size of the in-memory component reaches a certain threshold, the entries are sequentially flushed to a disk residing component by a rolling merge process. Updates to the on-disk components are performed out-of-place to ensure sequential writes. As the number of disk components increases, they form a hierarchy where the newest updates occupy the smallest components on the upper levels of the tree and updates propagate down to the lowest levels by merging together components. Disk components are periodically merged based on the specified merging policy, that decides the number of components of each level and when they are subject to merging. For more details see [41][46][37][38][59]

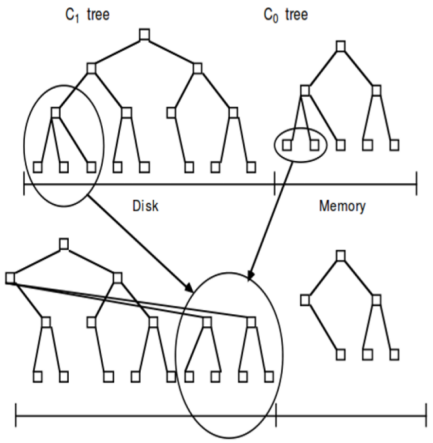


Figure 3.5: Merging process in the LSM-tree [41]

3.6.2 R-trees as a components

Since R-trees are spatial index data structures they tend to appear as a secondary index in LSM-based systems. Entries are on the form e = <sk, pk>, representing a secondary key and the associated primary key. A possible implementation is described in [60] and [59]. The in-memory components consists of a R-tree with and additional B-tree for storing deleted keys. The deleted-key B-tree is necessary in the merge based reconciliation step since the entries are not maintained in a totally ordered manner due to the n-dimensionality of the key. Insertion-operations insert entries into the in-memory R-tree in the same way as described in the theory chapter. Delete-operations on the other hand, deletes the key if it exists in the in-memory R-tree, and inserts a deleted-key entry into the B-tree to filter out entries returned from older components on disk during search operations. When an R-tree is flushed to disk the entries in the in-memory R-tree and Deleted-key B-tree are sorted by their value on the same Hilbert space-filling curve. The new on-disk component is created by merging and bulk loading the two set of entries. The rationale for merging by bulk loading the two sets is the out-of-place updates. The entries from the deleted-key B-tree become ”anti-matter” entries in the newly created disk component, if no identical entries exists in the sorted R-tree set.

Chapter 4

Related Work

4.1 Methods For Parallel Bulk Loading Spatial Data

Since parallel bulk loading are designed to handle large amounts data, how parallel or distributed environments can incorporate these methods is the focus of this section, as they are the state of the art solution. The goal is to decrease the bulk-loading time by adopting a parallel architecture. A R-tree forest can be built in parallel, where each machine builds an R-tree for an equal sized data partition. The overall R-tree is constructed by the roots of individual built R-tree [16]. Bottom-up and top-down methods are presented in this section.

4.1.1 Parallel Algorithm for Bulk loading R-tree using Z-ordering

Jianzhong Qi [24] describes a packing method for R-tree bulk-loading based on space filling curves which can be parallelized. The paper compares the query and insertion-performance with R-trees that guarantees a bound of performance in the worst case, such as the Priority R-tree. the proposed parallel R-tree bulk-loading algorithm outperforms the PR-tree bulk-loading algorithm in running time by 86% on large data sets with 20 million data points. The result illustrates the performance gain of parallelizing bulk-loading algorithms. In addition, the resulting query performance from the resulting structure outperform STR by 20%.

Packing method used

Suppose N number of points spread across a N-dimensional space. The first step in the packing method is to transform each coordinate in the over to a point in a N-dimensional rank space. If two points share the same value on one axis, then their value on the other axis is used to break ties. In figure 4.1 P_2 and P_3 share the same value on the X-axis, but since P_2 has a smaller value on the Y-Axis, P_2 will be placed before P_3 on the X-axis in the rank space. Similarly, P_6 and P_7 share the same value on the Y-axis, but since P_6 has a smaller value on the X-Axis, P_6 will be placed before P_7 on the Y-axis in the rank space. As a consequence, every point has an unique value on every axis of the N-dimensional space.

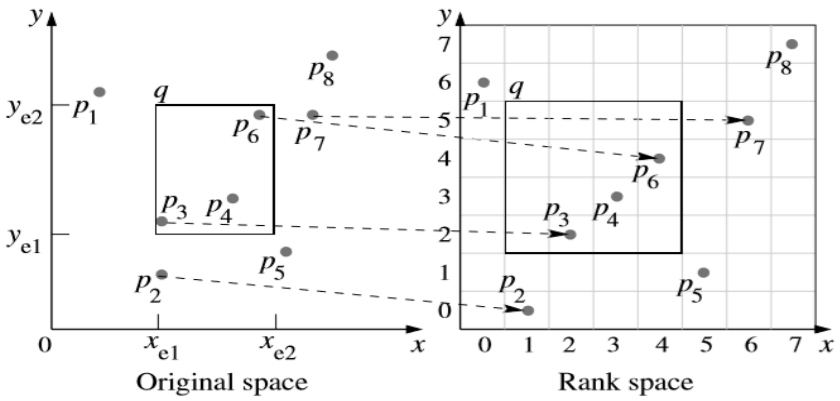


Figure 4.1: Mapping to Rank space [24]

After the mapping over to the rank space, the leaf nodes of the R-tree are generated by packing points in ascending order of their Z-order values. As previously mentioned Z-ordering is a space filling curve. It is recursively construed and the goal is to maintain locality-preserving properties of the N-dimensional curve. Since Z-order curve sometimes makes large jumps between two areas locality-preserving is worse than seen in Hilbert curves. However Z-order curves makes up for this in the simplicity of it's design. For every axis in the N-dimensional space, the integer value in each axis is converted into a binary representation. Z-ordering divides the space into $\prod_{i=1}^{N} 2^{B_i}$ quadrants, where B_i is the number of of bits that are used to represent a axis "i" in the N-dimensional space. Each quadrant gets it's unique identity by interleaving the binary values of every axis. The line sweeps over every quadrant from 0 to $\prod_{i=1}^{N} 2^{B_i}$ in an incremental fashion. The axis need to be ordered from most significant to least significant in order to give the sweeping line a direction. Then every point is sorted according to the z-order of the associated quadrant. The leaf nodes of the R-tree is created by inserting M, maximum capacity of entries within a node, consecutive points into each leaf. The total computational complexity of the algorithm is O(N·log N),which is the same complexity as sorting. Bulk-loading a R-tree takes O((N/M)$log_{W/M}$(N/M)) I/Os, where W is the number of data points that can fit inside the working memory.

Parallelizing The Packing Method

The goal of parallelizing the packing scheme, is to modify it so that it can be utilized on a massively parallel computation model(MPC) such as Spark or a MapReduce framework. Computations on data are organized in synchronous rounds. In each round of parallel computation, every machine receives some data from other machines and performs computations, while also sending processed data to other machines. Local computation costs per round plays a minor part in the total computational cost. Memory per processor is the most often regarded as the constraining factor of the model, also limiting the number of bits a processor can send or receive in a single round.

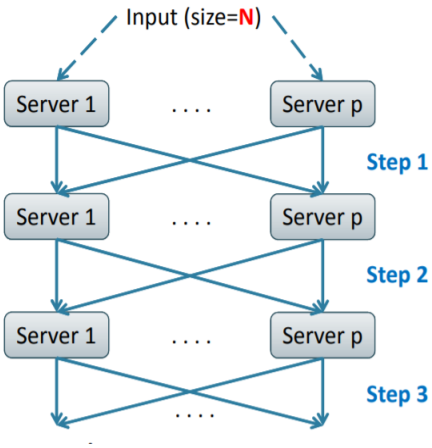


Figure 4.2: Rounds in MPC,[4]

Let N be the total size of the input, p be the number of processors(servers), then each of processor p must have a local memory to store the load $L \geq N/p$. The algorithm takes $O(log_s N)$ rounds. s is the amount of data points that can be stored in each load L. MPC algorithms are evaluated by:

- **the number of computation rounds R**
- **the parallel running time τ**
- **the total amount of computation ω**

Let Theta $\theta_{i,r}$ be the computational cost of processor **i** in round **r**. Then $\tau = \Sigma_{r=1}^{R} Max(\theta_r \in (1..p))$, since all the work is done in parallel, the machine with the most computational cost defines the upper bound for the round **r**. $\omega = \Sigma_{r=1}^{R}\Sigma_{i=1}^{p}\theta_{i,r}$ defines the total amount of computation, and should not exceed O(N·log N) if p = 1. The parallel running time τ should be O((N·log N)/p) in order for the parallelizing to be beneficial. The parallel algorithm fist require sorting. Sorting n data points given the MPC model, where the the load is evenly distributed on the p machines takes O(log_s n) rounds, O((n log n)/p) running time, and O(n log n) total amount of computation. Similarly Mapping n data points to the rank space and sorting by Z-order values takes O(log_s n) rounds, O((n log n)/p) running time, and O(n log n) total amount of computation. After these steps it is worth noting that each data point can be assigned to the processors in sorted order for packing independently of each-other. For each round of the packing, a new level of the R-tree is created. Assume there are n data points in total and s is the number of data points that can be stored inside the working memory of each processor. The first round n/s sub-trees will be created, assuming $p \geq n/s$. The next round takes n/s roots into the working memory of the processors, if we assume that a processor can store s roots, then n/s^2 sub-trees will be generated for the second round. Hence, the m'th round will result in n/s^m. Each round of the packing requires fewer processors then the previous. In total, the packing takes $\lceil log_s n \rceil$ rounds.

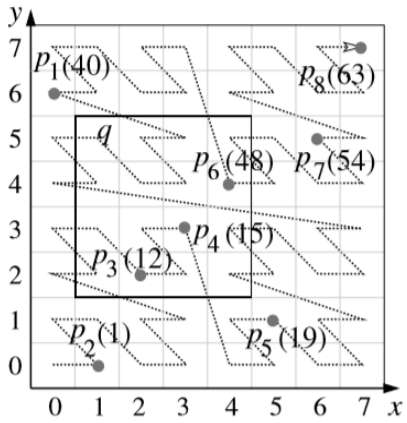


Figure 4.3: Z-ordering [24]

4.1.2 Parallel Bulk Loading Into R-trees in Parallel Spatial Databases

Another method for bulk loading for large amount of spatial data into a parallel R-tree operating across processors based on a shared-nothing architecture, is addressed in [11]. As mentioned earlier, we have to assume the data set is available in advance, in order for bulk loading to work. This can be achieved by archiving in a data warehouse or similar storage systems. This bulk-loading approach does not rely on sorting the initial data set based on spatial attributes. The challenge is to exploit parallelism in order to efficiently create an index that can operate over every machine in the system, not at the expense of worsening query performance. The method assume that there are a power of two processors in shared-nothing architecture, every processor has its storage and main memory, and that spatial attributes need to be horizontally fragmented across all or a subset of the processors. It's implicitly assumed that every spatial record is already divided among the machines in the system before the index is created. There are numerous steps in the process for generating a parallel index structure.

Random sampling

The first step is to select one processor as a coordinator. The rest of the processors sample $S_i = N_i \cdot f$. S_i is the sample size from processor **i**, N_i the total number of spatial records in the local storage of processor **i** and f $\in \langle 0, 1]$. We adjust f in a way that the sample size S_i is not too small, such that S_i is a representative selection of the data distribution. Too large sample size will decrease performance due to I/O overhead. The sample size may vary between the processors as a consequence of skewed data distribution. The samples are used in order to create partitions of the N-dimensional space derived from the spatial relations of the stored objects.

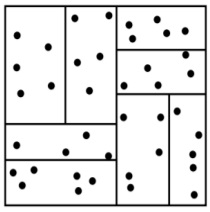


Figure 4.4: Partitions of a 2-D Space [11]

Space partitioning and region assignment

The spatial space needs to be partitioned in a way that each partition mapped to a single processor. The space decomposed by the spatial information retrieved from the samples. After splitting the space into $\|P\|$ regions, where $\|P\|$ is the number of processors. The reason $\|P\|$ is a power of two is to simplify the partition algorithm that is used. We can hope that each partition contain approximately N $/\|P\|$ elements, where N is the total number of entries in the parallel system. However the number of elements may deviate for sufficiently skewed distributions and a higher sampling rate f is required. Partitioning the spatial space resembles The Top-down Greedy Splitting Algorithm[62].
The next procedure in this phase is to find a suitable mapping between the regions and the processors. The goal is to assign a region to a process that contain the maximum of entries within that region. Each edge connecting the spatial region R_i to a processor P_j is weighted by the number of objects from P_j that are enclosed by region R_i. The problem can be reduced to weighted bipartite graph matching problem and can be solved by the Hungarian algorithm[27] with a complexity of $O(R^2E)$. If $\|R\|$ is small even exhaustive search might be a viable approach. The problem can be solved by a greedy algorithm that for each region assigns the region to the processor with the most samples within that region. There is a trade-off between the complexity for creating a suitable matching and communication costs and load-balancing between the processors. If P_j is assigned R_i, then every processor P_k,(P_j excluded), that has objects in R_i must to transmit the spatial information of the objects to the processor P_j in order for P_j to construct a local index. A local index consists of entries with (recordID, processorID, span of the spatial attributes). The observant reader can spot that the spatial object itself is not transmitted, only the spatial information.

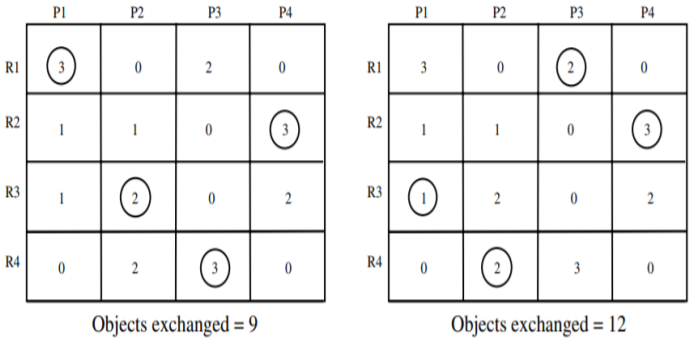


Figure 4.5: Matching between regions and processors [11]

Load balancing

Each processor should be in control of the approximately the same number of spatial objects in order to have the same amount of workload. So every processor has to send the spatial attributes of the objects that have been assigned to other processors, in addition to receive the spatial attributes from the other processors.

Generating local indices and a global index

Since the local indices know the data set in advance after the load balancing step, they are build by packing in order to have better space utilization, higher query performance and faster construction time than inserting each entry one by one. The upper levels of the local indices is sent to the coordinator. The coordinator can easily pack the numerous number of sub-tress from each local index, as each associated MBR do not intersect. Issues may arise in the global index created by the coordinator if the heights of the local R-trees vary. Underflow nodes or a limit on the height of each tree can be set by the coordinator in order to achieve a balanced parallel R-tree.

4.2 On Parallel Bulk loading of spatio-temporal data on a Parallel System

There are vast variety of spatio-temporal index structures, where each are designed for a special purpose use-case. Text data can be associated with spatial and temporal attributes, some applications require online access to the current locations of moving objects, others are specilized for indexing historical spatio-temporal data, predicting the furture based on historical data, predicting moving objects based on the underlying road network. However, the magnitude of spatio-temporal data that is being generated makes centralized indexes less suitable for the requirements of spatio-temporal applications [8]. parallel or distributed spatio-temporal access methods are either extensions to general-purpose scalable systems(in most cases built on a massively parallel computation model) or build as a standalone index

4.2.1 Spatio-Temporal Hadoop

Spatio-Temporal Hadoop (ST-Hadoop)[31] extends the Hadoop MapReduce system and is built for storing, indexing, and querying spatio-temporal data on a distributed environment. In order to bulk load the system a sampling process is required. A MapReduce job samples the spatio-temporal data and keeps it in main memory of the master node. The sample is used to direct the bulk loading of the spatio-temporal data by estimating the spatial distribution and how the distribution changes along the temporal dimension. The sample is read and than sorted in a chronological order, which is used to find size and the number of records for every time step. The sample is used to direct the bulk loading of the index structure. The data is first partitioned into temporal slices, then a spatial index is built for every temporal slice. The boundaries of the temporal slices are estimated from the sample. The indexing of temporal and spatial attributes are inserted into a temporal hierarchy index. The layers of the hierarchical index are bulk loaded iteratively. Each iteration creates temporal and spatial indexing with different time resolution, such as weeks, months, and years. Upon inspection, this bulk insertion scheme are in many ways analogous to the parallel bulk insertion into R-trees in parallel spatial system. Another access methods which uses similar methods as the spatial counterpart is the DTR-tree[21], which

is built on a share-noting Spark cluster. Each machine maintains a small R-tree for the data located on it and each machine can be used for creating large R-tree.

4.3 Methods For Bulk insertions Spatial Data

If we consider a data system that continuously receives spatial data into an existing index structure, we will eventually reach a point where the amount of data accumulated causes an unacceptable down-time if the index structure is to be rebuilt by bulk loading, given that the constructed index cannot handle dynamic data updates efficiently and without deteriorating the structure. Although horizontally scaling the system and parallelizing operations of the index construction process make periodically rebuilding the index less troublesome, it is worthwhile to address methods without this limitation as bulk loading methods do not make use of information stored in the already existing data structures. Bulk loading gives far better query performance than bulk insertion. Bulk insertions resembles dynamic insertion algorithms, where the structure may have excessive space overlap and dead-space, which in turn results to bad performance[57]. However, Bulk insertions may significantly outperform inserting entries one by one, both in terms of insertion throughput and query performance from the resulting index. There is a tradeoff between the time to load the new data into the R-tree and the quality of the resulting data structure[44]. Bulk insertion operations must be effective enough to surpass the rate of the data generation while maintaining good query performance.

4.3.1 Small Tree Large Tree (STLT) & Generalized Bulk Insertion Strategy (GBI)

The Small Tree Large Tree bulk insertion method is specialized with regard to inserting bulks of skewed data, as many non-synthetic datasets tend to be[44]. The assumption behind the algorithm is that data arrive in localized portions of the n-dimensional space. If the assumption holds true, then the data can be bulk loaded into smaller R-trees, with non-overlapping MBRs[56]. The idea behind the algorithm is to consider the root of the smaller trees, with it's associated MBR, as an individual entry of the R-tree insertion algorithm, where the entry is placed at the level corresponding to the difference in height between the trees. As seen in R-tree insertion algorithm, the most suitable node to insert the entry may contain more than M entries. Thus, STLT applies techniques in order of priority to handle overflow.

- **Merging sibling nodes:** Tries to create free space by combining the two closest nodes with the combined number of entries less than M. If no suitable pair of nodes are found, then the next technique for handling overflow is to reinsert a node.
- **Reinsert:** The goal is to delete and reinsert the node, whose MBR has the largest distance from the center of the enclosing MBR.
- **Splitting:** As a last resort, the overflowed node is split. The splitting may cascade upwards all the way to the root.

Generalized Bulk Insertion Strategy (GBI) [45] extends the STLT in such a manner that non-skewed datasets can be used with STLT. A clustering phase is applied before the packing algorithm in order to identify clusters and outliers. Then for each clusters a small R-tree is constructed and STLT can employed. Entries that does not belong to a cluster are classified as outliers. Outliers are inserted one by one into the large tree. In terms of I/O cost for insertion, GBI scales far better than inserting each entry one at the time into the tree. However, the rate

of I/O query cost from the resulting index, as the number of entries in the index increases, is slightly worse for GBI than individual insertions. The query performance of GBI is explained by the overall overlap within the R-tree. If we decrease the space covered by the small R-trees, the structure of the overall tree will contain less overlap and give better query performance. Consider that the root of the small tree covers a smaller area, the node of the large tree where the entry is inserted into is less likely to be enlarged, thus resulting in less overlap. There are no guarantee that the roots of the inserted small trees are not overlapping with existing nodes. It should also be noted that during certain conditions the constructed large tree may break a property of having nodes with less than m entries. Since there are no requirements for a minimum number of entries in the root, a small tree can be constructed consisting of a root with fewer than m children or entries. If we are to place that specific small tree into the large tree as an internal node, then the node would be in an illegal state and make the R-tree not legitimate by the minimum fill requirement.

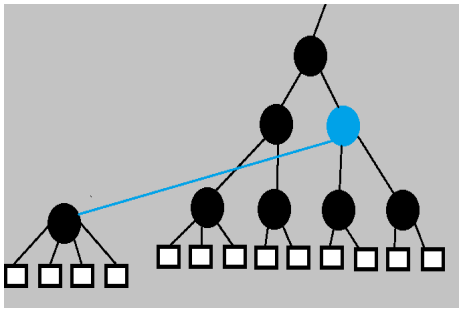


Figure 4.6: Inserting a smaller tree into a large one

4.3.2 Bulk insertion by seeded clustering (SCB)

The previously mentioned shortcomings of GBI with regard to node underflow and overlap are solved by Bulk Insertion by Seeded Clustering [52]. To guide the insertion of objects, the method creates a seeded tree of top k levels of the R-tree index to classify the input data items into clusters in a linear time. The purpose of the seeded tree is to cluster the entries by taking the existing structure into consideration. Each cluster is created by grouping the entries that share the same path from the root of the seeded tree to a leaf of the seeded tree. The entries that are covered by the MBR of a leaf node are bulk loaded to create a small R-tree. Note that the MBR of the small tree does not exceed the MBR of the leaf node in any dimension, and thus no expansion of the MBR is necessary. The figure 4.7 below illustrates this point. A and B are two MBRs at the k level of the target R-tree. If we disregard the structure of the tree, then B would have to be expanded to include the point in the bottom right corner, as it would be part of a cluster along with the two right entries. However, in the seeded tree method, the point would be grouped with the three left entries and hence avoid the need to expand the MBR. Data points that does not find path in the seeded tree are classified as outliers and are inserted one by one into the R-tree. The roots of the small trees must be inserted at the appropriate level to keep the tree balanced. if the root of a small tree contain less than m entries, the entries of the root is distributed among the sibling nodes that overlaps. SCB achieves reduction of overlap by a post-processing step. Assume that V is a node in the R-tree, that is most suitable to place the root S of a small R-tree. The entries of V are partitioned by those who overlap with S and those that do not overlap. The overlapping entries are repacked by creating nodes that enclose the objects using a bulk loading

method. According to the results of the paper, SCB outperforms GBI in terms of both insertion and query performance. [52].

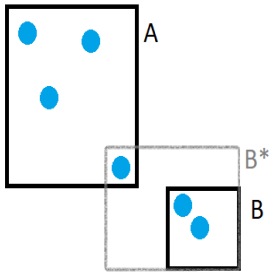


Figure 4.7: MBRs when the existing R-tree structure is taken into consideration

Chapter 5

Merging

There is third alternative way of dealing with insertion of large amounts of data, where the task is to unify datasets indexed by individual data structures into a single indexed dataset. This process is often referred to as merging. The goal of merging can be solved by both bulk insertion and loading as they both produce a single indexed dataset. One can even argue that merging is a special case of the former. However, merging differs as there are no restrictions on the size of the tree that are merged into the receiving tree. Lets us define the methods of merging R-trees down to first principles in order to make a clearer distinction from bulk loading and bulk insertion.

5.1 On the design space of merging algorithms

Given the constraints of the R-tree, what are the design space of possible methods for merging R-trees. In algorithmic design and more recently outlined in data structure design as seen in [50], there are design principles that shape the design in order to solve a problem or optimize how data is accessed, stored or updated. Merging differs from bulk insertion and loading, in the sense that the individual components that are to be merged, are data structures themselves that can access, store and update data. If we decompose one of the trees down to its individual components, then we have entered into the domain of bulk insertion. Likewise, if we fully decompose both trees, the problem resembles bulk loading. What heuristics applied for inserting sub-trees, splits and what criterion, used for deciding if subtree is going to be left intact or decomposed, are the most defining factors of a merging method. In data structures like the LSM-tree, merging plays an additional role, the process reconciles updates through the lower levels of the tree. The number of components in a level and the merging frequency shape the performance of the data structure. Merging standard R-tree components are in-place updates, compared to merging LSM-based R-tree components that are out-of-place. Update in-place systems tend to have unrivaled query latency but, worse write-throughput. LSM-based R-trees also require a specified shared ordering(i.e Hilbert curve, Z-ordering) in order to be effective, as queries have search regions that rely upon scan operations.

5.2 Merging R-trees, Vasaitis, Nanopoulos and Bozanis

The only work to explicitly solve problem of merging R-trees was done by Vasaitis, Nanopoulos and Bozanis[55][56]. The paper describes a method where one tree is inserted into another by a linear top-down traversal of one of the trees. The method does not make any assumptions of

the underlying data distribution like STLT. The paper found no noteworthy difference in quality between the trees produced by merging and bulk-loading. Nonetheless, the construction time for merging was an order of magnitude smaller than for bulk-loading.[55] Given two trees, the tree that is being inserted into is defined as the *receiving tree* while the other is named the *giving tree*. The merging process relies on two variable-size buffers, created and reused as needed, attached to every node of the receiving tree. The first buffer is referred to as the insertion queue. The purpose of this buffer is to store entries that are destined to the node itself and the child nodes enclosed by the associated MBR. The other buffer is known as local insertion queue, which sole purpose is to store entries that are destined to the node itself. The buffers are expanded and contracted by the changing number of objects stored within. The use of buffers are somewhat similar to existing design solutions, like the buffer tree proposed by Arge[12]. In order for the queues to be accessed in an optimal manner, M + 3 extra pages must be available in the main memory. 2 pages for the node's insertion queue, M for the child nodes' insertion queues, and 1 for the node's local insertion queue. One page is required for reading, storing the page that contains the current entry, and another page for writing, storing the tail of each queue. The proposed merging algorithm is more memory intensive compared to the R-tree insertion algorithm, where only one page is needed to store the current node.

5.2.1 Generalized split and Multiple splits

The splitting algorithms of the R-tree and other augmented variants, deal with M+1 entries of an overfull node into two partitions where each one has between m up to M+1-m entries. However, during merging there are no upper limit on the number of elements inserted into one node at the same time. Therefore, the partitioning problem needs to be generalized in order to handle nodes with an arbitrary number of entries. M and m are redefined as L and l, where $l = \lfloor \frac{Lm}{M+1} \rfloor$. In the case of $L = M + 1$, l would be equal to m. Since each node can contain an arbitrary large number of entries after the split, the generalized split is recursively invoked until all the produced nodes contain less than M entries. The node generated from the splits are inserted into the local insertion queue of the parent of the overflowed node.

5.2.2 Tree insertion and decomposition

The insertion begins at the root of the receiving tree and recursively descends down to the leaf level. The root of the giving tree tree is inserted as an entry into the insertion queue of the root. In the best-case scenario, the entirety of the giving tree can be inserted as a single entry into the receiving tree. A tree is decomposed into smaller subtrees or single objects when the height is not suitable, or the decomposed entries give better performance with regard to overlap or area expansion. If the subtree needs to be placed at a lower level, the subtree is routed to the insertion queue of child node that require least area enlargement. If the children of the subtree would cause less area enlargement in total than their parent node, then the subtree is decomposed. a similar comparison and decomposition is applied with regard to overlap, if the subtree is to be placed at the current level of the receiving tree.

5.3 Merging method that bridges advantages from bulk insertion

Consider the similarities between bulk insertion and merging, both methods inserts subtrees into a target tree. The only difference is that the sub-trees are attained from clusters instead of

decomposed from larger trees. By trial and error, a new merging method with the heuristics from Seeded Clustering and General Bulk Insertion was devised in this project. The implementation chapter provides additional details of the method.

5.4 Merging patterns

In the case of several R-trees with low height from root to leaves, merging the R-trees can be done in the same way as packing nodes in a bottom-up bulk loading methods. Given a variety of R-trees with different heights, finding the most optimal order of merging the R-trees can be a non-trivial task. The decomposition heuristics and restructuring invoked by the inserted subtrees, affect the optimal ordering. One possible pattern is to choose the largest R-tree among the set of trees as the target tree, and insert the smallest tree into the target tree. As a result the number of decompositions are minimized. Another way is to choose to merge the R-trees with the smallest height difference, resulting in more restructuring. It is also possible to choose the ordering of merged components by their degree of overlap and area enlargement. If the merging method deteriorate the resulting structure, regardless of the ordering, then it is beneficial to have the fewest number of components to be merged. The pattern chosen in this thesis is the minimization of the number of decompositions.

Chapter 6

Implementation

6.1 Choice of Language

The R-tree and the following methods was implemented in Java. Java is often bench-marked as one of the fastest language platforms available, comparable to C and C++.

6.2 Comparative Study of Merging and Equivalent Methods

To study and compare different methods for writing spatial data to the R-tree, the following data ingestion methods was implemented: **one by on insertion** with the *R* quadratic split*[40], The **Seeded Clustering(SC)** Bulk insertion method[52], **Merging** method with the use of *buffers*(almost identical to[55]), **Merging** method without *buffers* that employ the *post-pruning* from **SC**[52] and *reinsertion* from **GBI**[44][45], and the **Sort Tile Recursive (STR)** bulk loading method [47] was implemented as well. The two implemented merging methods use two fundamental different approaches when inserting subtrees into the target tree. *The without buffers* approach mainly restructures the target tree in the lower levels to best suit the newly inserted sub-tree, while *the with buffers* is more likely to decompose a sub-tree and expand the upper levels of the target tree after inserting the sub-tree. For the purpose of evaluating the performance of proposed merging methods and equivalent methods. A test was conducted to compare the performance with regard to query performance, evaluated in terms of IO. The time needed to create, update and merge the tree was also measured.

6.2.1 Sort Tile Recursive

This bulk loading method is described in theory chapter. Enhancements to the method was applied in order to prevent nodes from containing less than m entries. A node can potentially steal [1,m-1] entries from the M entries of the left sibling node, justified by the fact that they lie closest to each other in the data space. The method is fairly simple to implement and is the packing and bulk loading method used in the following Seeded Clustering method. For the sake of consistency, the method is also used to create the two input R-trees for the merging methods subsequently described.

6.2.2 Seeded Clustering

This bulk insertion method is described in *Related Work*. The post-process method is included the in this section as it is reused in the subsequent merging method. A problem not fully addressed in the original paper by Taewon Lee [52], is the scenario where over and under-flowed nodes generated by the insertion-procedure. An overflowed node is a likely scenario, especially if a R-tree with good space utilization is inserted into a parental node. For simplicity reasons, I chose to reinsert the children with the largest distance from the center of covering MBR. If we inspect the Repacking method invoked by the post-processing, we find that potentially more than one entry of **Ni** can overlap with the same entry of the set *ent*. See figure 6.1. There are several possible ways for solving this issue. One can simply remove entries of *ent* after they have been repacked with the first overlapping entry of **Ni**. However, they may still overlap with the next **e** $\in$ entries of **Ni**. Another way is to put the repacked entries into *ent* but, if the degree of overlap between the entries of **Ni** and *ent* is sufficiently large enough, packing the whole sub-tree with bulk-loading is more favorable as the entries will not be redundantly packed.

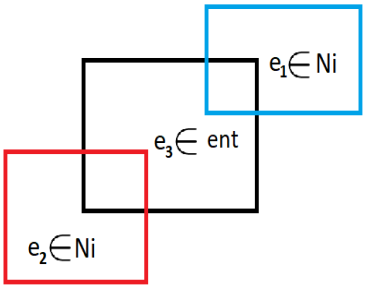


Figure 6.1: Overlapping MBRs in the Repack method of post-processing

```
Post-Process(Nt, Ni);
input  : Nt,Ni
output: A set of packed nodes
Nt: the node where Ni is inserted;
Ni: the root of an input R-tree;
NoOvlp ← Entries of Nt that does not overlap with Ni;
Ovlp ←Entries of Nt that does not overlap with Ni;
Repacked ← Repack(Ovlp, Ni);
return Pack(Repacked ∪ NoOvlp)
```

Algorithm 1: Post-Processing the Overlaps of MBRs and insertion of a sub-tree into the parental nodes

```
Repack(Nset, Ni);
input  : Nset,Ni
output: A set of packed nodes
Nset: a set of nodes whose parent nodes overlap with Ni;
Ni: a node from the input R-tree;
ent ← the set of entries from every node in N_set ;
Ovlp ←Entries of Nt that does not overlap with Ni;
NoOvlp ← subset of ent that does not overlap any entry of Ni;
Repacked ← ∅;
if Ni is a leaf then
 | return Pack(ent ∪ entries of Ni)
else
 | for e ∈ entries of Ni do
 |  | entOvlp ← find elements of ent that overlap with e;
 |  | if entOvlp ≠ ∅ then
 |  |  | Repacked ← Repacked ∪ Repack(entOvlp,e)
 |  | else
 |  |  | Repacked ← Repacked ∪ e
 |  | end
 | end
end
return Pack(Repacked ∪ NoOvlp)
```

Algorithm 2: a recursive repacking method

6.2.3 Merging Method Without Buffers

This merging method was devised with the advantages from the previously described methods. Decomposing subtrees based on the area criterion[55] proved to be a simple and effective heuristic. It is implicitly assumed in the merging algorithms, that the function *Area Criterion* descends each level of the tree down to the parental node and checks if the criterion is satisfied. An overlap criterion was not applied, since the post-processing step removes most of the incurred overlap. When inserting a subtree into an eventual overflowed parental node, the merging sibling heuristic from **STLT** and **GBI** is not applied, as we assume both the sub-tree and target tree have good initial space utilization. Finding two close siblings with the combined number of entries less than M+1 would be unlikely. Reinserting was a good way to minimize the size of the covering MBR during overflowed nodes. If one entry has an MBR that lies on the corner of the covering MBR, than the area of covering MBR can be reduced if the bordering entry is replaced into another branch. Finding a parental node to insert an entry was done in similar way to the original R-tree

insertion algorithm, only differing in the levels of the tree to consider. Reinsertions was also applied to subtrees that had less than m entries. Note that with the augmented STR previously described, this is an impossible scenario. Nevertheless, this is an too important functionality to omit. A major disadvantage of the the method is that the reinsertion and post processing steps make concurrency control extremely difficult.

```
Merge(Input R-tree, Target R-tree);
input : Input R-tree, Target R-tree
output: Merged R-tree
Entry describes both sub-trees and records/spatial objects;

if Input R-tree height > Target R-tree height then
 | merge(Target R-tree, Input R-tree)
end
InsertionStack ← Input R-Tree Root;
while InsertionStack ≠ ∅ do
 | Entry ← Remove Head(InsertionStack);
 | if children of Entry < m then
 | | reinsert(children of Entry)
 | end
 | if not satisfied Area Criterion(Entry, children of Entry) then
 | | InsertionStack ← Decompose(Entry)
 | else
 | | Parent Node ← FindNodeTo Insert(Entry) ;
 | | Post-process(Parent Node, Entry);
 | | if #Children of Parent Node > M then
 | | | if CanReinsert(Children of Parent Node) then
 | | | | reinsert(Children of Parent Node)
 | | | else
 | | | | Split(Parent Node)
 | | | end
 | | end
 | end
end
```

Algorithm 3: Merging algorithm with Post Processing and Reinsertion

6.2.4 Merging Method With use of Buffers

The method implemented is very similar to the merging method by Vasilis Vasaitis [55] described in a chapter 5. Both the Area and overlap criterion are implemented, in addition to the use of buffers. The merging method in the previous chapter only reads a node of the receiving tree once in order to insert all the sub-trees or entries. However, as previously mentioned this comes at the price of having two buffers for each node, the local insertion queue and insertion queue. The local insertion queue of each node needs to be retained until each entry is finally loaded. The buffer is freed up after each element is inserted into the sub-tree and potentially multiple sibling nodes are generated from generalized split procedure. The insertion queues on the other hand can be relocated to other nodes after each sub-tree or entry is guided down to the next insertion queue of the nodes on the lower levels of the tree. We can notice that there are two phases of a merging method that rely on buffers. The first phase is to load all the sub-trees and entries of the deconstructed input-tree(giving tree) into the appropriate set of local insertion queues. The second phase is to build the tree bottom-up with multiple generalized splits. Both phases are

completed in a recursive tree insertion algorithm in the original paper by Vasaitis, Nanopoulos and Bozanis[55].

In order to potentially incorporate this merging method on a parallel or distributed environment, I found it necessary to explicitly divide the method into two phases. The insertion queue for each node was replaced with a shared **insertion stack**. The idea is to apply multiple processes or threads in a load balanced manner to find the correct local insertion queue of a parental node to store the entry. The idea is to invoke a thread for each entry in the insertion stack. Note that the deconstruction and loading phase does not change the structure of the tree, requiring only r-latches on the nodes. A semaphore or a similar access control mechanisms is needed for the local insertion queues and the insertion stack. The only downside is that the nodes of target tree (receiving tree) needs to be read more than once. Multiple threads was not applied in the experimental evaluation, as it would be an unfair comparison of the execution time.

```
Merge(Input R-tree, Target R-tree);
input  : Input R-tree, Target R-tree
output: Merged R-tree
Entry describes both sub-trees and records/spatial objects;

if Input R-tree height > Target R-tree height then
 | merge(Target R-tree, Input R-tree)
end

Deconstruction and loading phase:;

InsertionStack ← Input R-Tree Root;
LocalInsertionQueues ← initialize ;
while InsertionStack ≠ ∅ do
    Entry ← Remove Head(InsertionStack);
    if Entry is a record then
        Parent Node ← FindNodeToInsert(Entry);
        LocalInsertionQueue of Parent Node ← Entry;
    else
        if not satisfied Area Criterion(Entry, children of Entry) then
            InsertionStack ← Decompose(Entry)
        else
            Parent Node ← FindNodeToInsert(Entry);
            if not satisfied Overlap Criterion(Parent Node, Entry, children of Entry)
             then
                InsertionStack ← Decompose(Entry)
            else
                LocalInsertionQueue of Parent Node ← Entry
            end
            ;
        end
    end
end

Construction and splitting phase:;

while  LocalInsertionQueues ≠ ∅ do
    for each Node in level do
        a set of Generated Nodes ← Generalised split(Node, LocalInsertionQueue of
          Node);
        if Generated nodes ≠ ∅ then
            For the last level, check if root;
            if Node is root then
                Parental Node ← New Node;
                New Node children ← Node;
            end
            LocalInsertionQueue of the Parental Node ← Generated Nodes;
        end
    end
end
```

Algorithm 4: Merging algorithm with Buffers

6.2.5 One by one insertion:

The *choosesubtree* method inserts entries along the path that least expands the MBR. In case of a tie, the node with the smallest MBR, in terms of volume/area, is chosen. The split applied is the *R* quadratic split* from Norbert Beckmann [40], every method implemented, except STR, adopt this split or a generalized version. The split relies on sorting. First the axis that generate the smallest margin is chosen. Nodes are sorted on their values in this dimension. For both minimum and maximum value, partition the nodes on a split point k, where (m<=k<=M+1-m). Choose the partition with the minimum overlap. In case of tie, choose the partition that has the smallest total volume/area. The original R*-tree applies reinsertions, the generated R-tree is not by definition an R*-tree. The revised version of the R-tree does not rely in reinsertions, but has a more advanced set of heuristics and weight functions. They were not implement here. Forced-reinsertions hinders concurrent users as efficient algorithms for concurrency control become increasingly difficult [39].

6.3 Data-sets

In the performance test of the merging and equivalent methods, three data-sets were used. One from `https://www.kaggle.com/max-mind/world-cities-database` that represents real data, and the other two containing synthetic data-points in respectively 2 and 5 dimensional space. The purpose is to illustrate the performance for the methods with regard to data points from skewed and uniform distributions. A R-tree created by STR, will have no overlapping MBRs in the leaf-level when dealing with data-points. Usually, query and insertion performance will degenerate with increased dimensionality. The 5-dimensional dataset was used to figure out to extent that claim holds true. In the case of a 20-dimensional, access methods like the X-tree [49] clearly outperforms any variant of the R-tree. Limiting to five dimensions shows to be a good realistic upper bound.

6.4 The Cities of The World

The data-set consists of 3 173 958 cities. There are duplicate entries in the data-set, as "Trondheim" and "Trondhjem", that are two names for the same city, both exists in the data-set as individual entries. Even some neighborhoods like "Moholt" are included as a city. Each entry accommodate both latitude and longitude, with up to 6 decimal precision, resulting in an accuracy down to 11 cm. Latitude ranges from -90 to +90, from the southern to northern hemisphere. Longitude on the other hand, ranges from -180 to +180 specifying coordinates west and east of the Prime Meridian. The data-set is skewed in the sense that each entry is sorted by the lexicographical order of country. This results in that the placement of data points in the data-space reflects the order they are inserted into the index. coincidentally, the data distribution is skewed as well. The Sahara desert contains far less entries per square kilometer than central Europe. A heat map is included to visualize density of cities within an area. Note that the map does not illustrate population density, rather specifies the placement of entries from the data set.

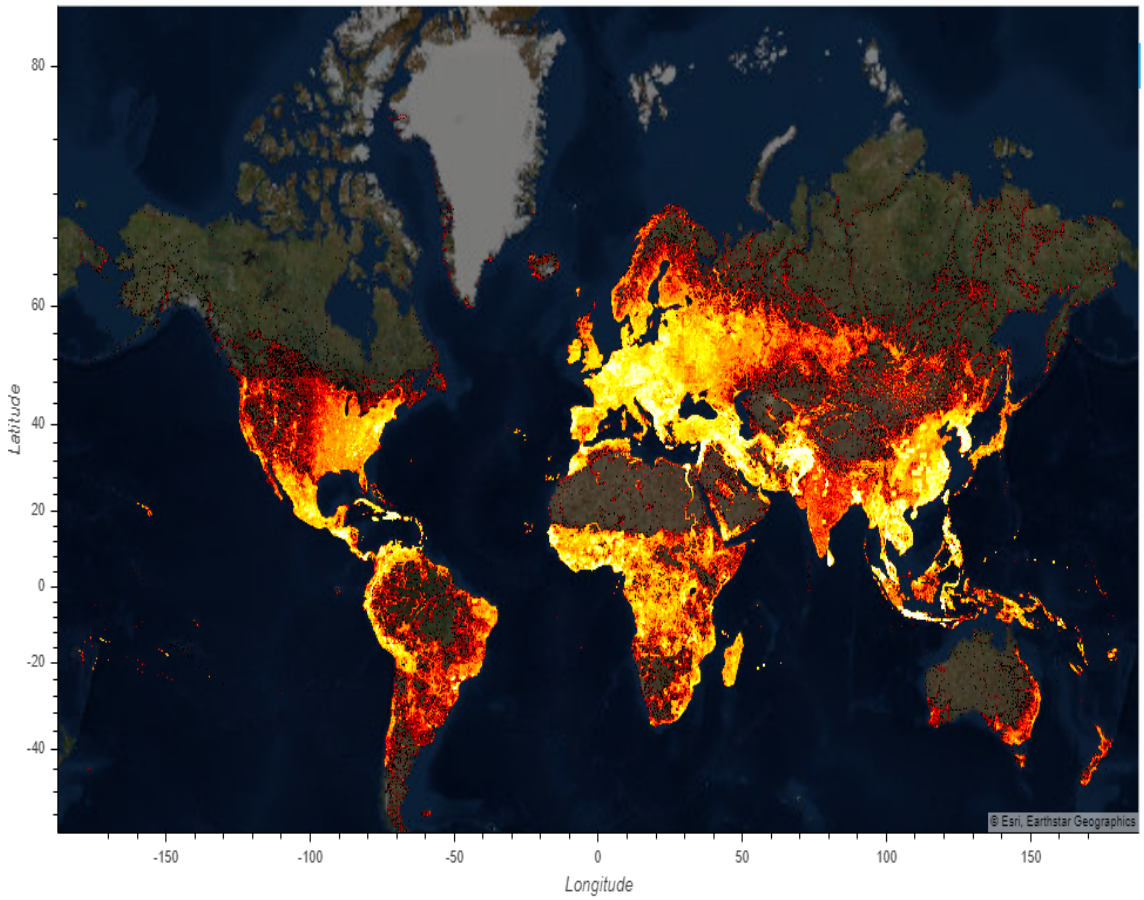


Figure 6.2: Heat map of cities: generated by datashader, geoviews and holoviews

6.5 Synthetic data

9 000 000 million entries was generated for this dataset. Each axis has length from 0 to 100. The coordinates of the data-point was represented by a random 32-bit floating point in the IEEE 754 Standard, within the same bounds as the axes. By the law of large numbers, the data points are distributed equally across the data space, making the distribution uniform.

Chapter 7

Experiments, Results and Discussions

7.0.1 System Configuration

The table below specifies the system configuration used when running the test. The main memory is sufficiently large enough to store every spatial data point of the datasets.

Operating System:	Windows 10 64bit
Processor:	Intel Core i7-4720 HQ CPU @ 2.60 GHZ
Logical Cores:	8
RAM:	16 GB

Table 7.1: System Configuration

7.1 Experimental evaluation

7.1.1 Metrics

To test the performance of the different methods the resulting query performance from resulting R-tree and the construction time was measured. The total number of IOs(Writes and reads) required to construct the index was also possible to evaluate. Since the bulk loading method, post-processing step, area and overlap-criterion makes most of the methods CPU-bounded, the total number of IOs for constructing the index was omitted as a metric.

7.1.2 Explicit Cost Model for calculating the total number of reads from queries:

To evaluate query performance, a cost model was implemented. The model takes a search region as an input and returns the number of IOs that required to perform the query. Let us assume that leaves contain entries of the form: entry = (Oid, R), where Oid is an object identifier and R is the approximation of the MBR containing ((l_0,u_0), (l_1,u_1) ... (l_{n-1},u_{n-1})), where l_i and u_i are the lower and upper limit of dimension **i**. Similarly, the non-leaf nodes contain entries on the form: entry = (Pid, R), where Pid is the pointer to a child node. If R was not stored within an entry, but rather in the child node or object, then the query operation would have to read all

the [m,M] entries to retrieve the MBR, in order to check the spatial condition. In this explicit cost model, a node is only read if it's MBR intersects the search region **S** of the query. Reading records is also regarded in this cost model as an IO operation. In this cost model we assume that every memory address Pid can fit into main memory. Note that there exists models for the prediction of R-tree performance without explicitly counting the number of IO[58].

7.1.3 Construction Time:

The construction time was measured at the time when a method had the specified input. In the case of the merging algorithms, the clock started when correct amount of R-trees was available, meaning that the bulk loading phase was not recorded. In the case of inserting one by one and bulk loading(STR), time started when every entry to be inserted was available. The initial bulk loading phase of Seeded clustering was not recorded. Time was recorded just before the seeded tree was built.

7.1.4 Methods to be tested

STR:

The whole dataset is bulk loaded with the Sort-tile recursive method.

Inserting one by one:

Every entry in the dataset is inserted one by one into the R-tree. Quadratic split, from R*star-tree[40], is invoked in case of overflow.

Merging without buffers:

Divide the dataset into two equal portions and bulk load each subset with STR, then merge the two components together.

Merging with buffers:

Divide the dataset into two equal portions and bulk load each subset with STR, then merge the two components together.

Seeded clustering:

Divide the dataset into two equal portions, bulk load the first component with STR, then apply Seeded clustering with the rest of the entries.

Merging without buffers inc:

Divide the dataset into nine equal portions and bulk load the each subset with STR, then incrementally merge the one component with the rest. This will illustrate how repeated merge operations affect the construction time and query performance.

Merging with buffers inc:

Divide the dataset into nine equal portions and bulk load each subset with STR, then incrementally merge the one component with the rest. This will illustrate how repeated merge operations affect the construction time and query performance.

Seeded clustering inc:

Divide the dataset into nine equal portions, bulk load the first component with STR, then incrementally apply Seeded clustering to the rest of sub-sets. With this approach 8 Seeded trees in total are built. This will illustrate how repeated seeded clustering affect the construction time and query performance.

7.1.5 Default parameters

The default parameters of each R-tree is shown in table 7.2. These parameters are used in every experiment conducted, except for the construction time of the real data set. It is useful to show the construction time vary for a fixed number of entries in a data set where M is the variable. The value of 40 for M, was selected to give a fairer comparison to *insertion one by one*. Any value below that degraded the query performance of *insertion one by one* to the extend that the results might give an unrealistic image.

M:	40
m:	16, 40% of M
top k levels of seeded tree:	3

Table 7.2: Default parameters of the R-tree

7.2 Test results for real data

A set of realistic queries was generated to test the performance of the methods. purpose is to illustrate and compare the query performance with a set of queries that can be put forward by a user.

Q1

Q1 is a point-query to find the city of Trondheim. For some point-queries inserting one by one can outperform STR. This query do not reflect the overall performance of the tree, but is intended to illustrate the number of reads required to fetch a single entry(or duplicate entries). As previously mentioned, Trondheim exists as a duplicate entry in the dataset, thus increasing the number of IO's.

Q2

Q2 is an area-query by a minimum bounding rectangle with Paris and Berlin in each respective corner. This query returns every entry contained within the MBR, traversing several paths of the tree, due the density in this area is relatively high.

Q3

Q3 is an area-query to retrieve every city in world, traversing every path in the tree. Note that this query indirectly evaluates the space-utilization of the tree.

Qset1

Qset1 is a set of 25 queries covering the whole world, where each query covers the lengths of 72 units of longitude and 36 units of latitude. Due to earth having a spherical shape, the area of each query is not the same. The average number of reads is denoted in the tables. This query set aggregates the query performance of large search areas where the distribution of data points is skewed.

Qset2

Qset2 is a set of 9 queries covering China. The average number of reads is denoted in the tables. This query set aggregates the query performance of medium size search areas where the distribution of data points is increasingly more dense in the direction of the coast.

Method	Q1	Q2	Q3	Qset1	Qset2
STR:	8	70 124	3 175 995	127 097.24	15 637.22
Inserting one by one:	15	70 207	3 178 538	127 205.80	15 663.00
Merging without buffers:	10	70 129	3 175 995	127 111.52	15 641.44
Merging with buffers:	9	70 127	3 176 172	127 106.84	15 614.00
Seeded Clustering:	9	70 134	3 176 606	127 134.16	15 638.44
Merging without buffers inc:	11	70 135	3 175 999	127 118.44	15 665.55
Merging with buffers inc:	22	70 155	3 176 151	127 120.20	15 658.44
Seeded Clustering inc:	16	70 145	3 176 713	127 142.92	15 661.44

Table 7.3: Number of IOs by querying the resulting R-tree, default ordering

Method	Q1	Q2	Q3	Qset1	Qset2
STR:	8	70 124	3 175 995	127 097.28	15 637.44
Inserting one by one:	8	70 183	3 178 283	127 188.72	15 656.89
Merging without buffers:	10	70 131	3 175 995	127 108.32	15 644.67
Merging with buffers:	10	70 146	3 176 125	127 112.68	15 643.23
Seeded Clustering:	8	70 145	3 176 551	127 130.88	15 646.89
Merging without buffers inc:	11	70 146	3 175 999	127 121.84	15 664.67
Merging with buffers inc:	24	70 218	3 176 045	127 156.12	15 681.78
Seeded Clustering inc:	12	70 167	3 176 236	127 149.68	15 683.44

Table 7.4: Number of IOs by querying the resulting R-tree, random permutation ordering

Observations:

When constructing the R-tree with the default ordering from the dataset, we can observe that **STR** and **insertion one by one** respectively represents the bounds on the query performance. The query performance of **insertion one by one** seems to be dependant on the order of which the objects are inserted, while STR is not affected. The query performance of the the merging methods and Seeded clustering deteriorate when the merged R-trees are built from entries that span a larger area. Note that in the random permutation ordering, portions of the dataset contain entries spanning all over the world.

7.2.1 Construction time: real data

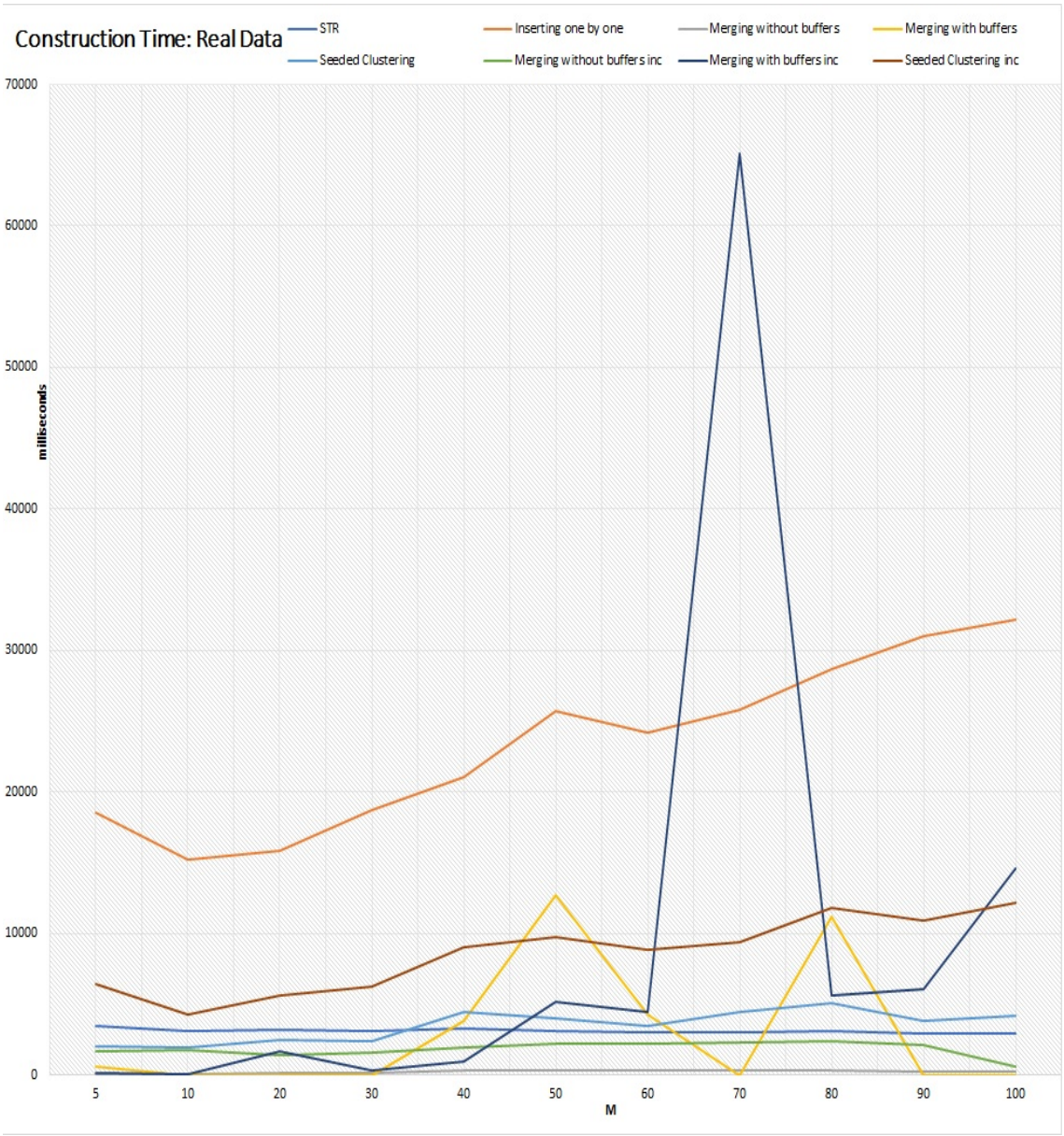


Figure 7.1: milliseconds for loading 3173958 cities, with increasing maximum fanout

Method	M:5	M:10	M:20	M:30	M:40
STR:	3466,2	3067	3161,2	3135,2	3299,2
Inserting one by one:	18567,6	15264,2	15846	18751	21038,2
Merging without buffers:	87,4	32,6	150	115	369,4
Merging with buffers:	618	0,2	1,8	0,6	3802,2
Seeded Clustering:	2040,4	1930,4	2514	2434	4445,4
Merging without buffers inc:	1660,8	1803,8	1375,6	1610	1904,8
Merging with buffers inc:	130,8	73,8	1675	294,6	936,6
Seeded Clustering inc:	6435,4	4304,2	5597,2	6274	9026,2

Table 7.5: milliseconds for constructing the R-tree with increasing maximum fanout M: part 1

Method	M:50	M:60	M:70	M:80	M:90	M:100
STR:	3067,8	3001,2	3007	3141,6	2948,8	2916
Inserting one by one:	25701,2	24159	25774,2	28718,8	30987	32206,4
Merging without buffers:	350,2	315,2	353	369,4	239	214,2
Merging with buffers:	12750,6	4275,6	1,4	11192,2	0,8	0,4
Seeded Clustering:	4047,2	3472,2	4468,6	5065,6	3822,6	4220,6
Merging without buffers inc:	2254,8	2201,4	2274,4	2406,2	2115,4	587,8
Merging with buffers inc:	5162	4476,6	65121,6	5629,8	6045,6	14571,6
Seeded Clustering inc:	9731,2	8860,6	9418,8	11772,6	10912,6	12213,4

Table 7.6: milliseconds for constructing the R-tree with increasing maximum fanout M: part 2

Observations:

Since the number of entries in the dataset is fixed, it is interesting to observe how the size of M impact the construction time. STR has more or less the same value throughout the graph, due to the fact that sorting dominates the construction time. The construction time of inserting one by one increases linearly, as a consequence of partitioning the entries during a split require more computations with increased maximum fanout. The most interesting observation is the sudden spike of construction time seen in the **Merging with Buffers methods**. In the worst case scenario, every sub-tree of the input tree is decomposed down to individual entries. We observed in the case of inserting entries one by one, that partitioning a larger set of entries require more computations. However, with an arbitrary large number of entries in the local insertion queue buffer and multiple recursive splits, the construction time explodes. Splits can also cascade to the upper levels as generated node are inserted into the local insertion queue buffer of the parental node, causing a new wave of potential multiple splits.

7.3 Test results for synthetic data

Since the distribution data points is uniform across the data space. The query performance was tested by dividing the the data space into **n** partitions and taking the average number of reads from the n search areas. Step denotes the length of each side in the search areas. When dividing the data space into different portions to build R-tree components, the entries in the portions span all over the data space.

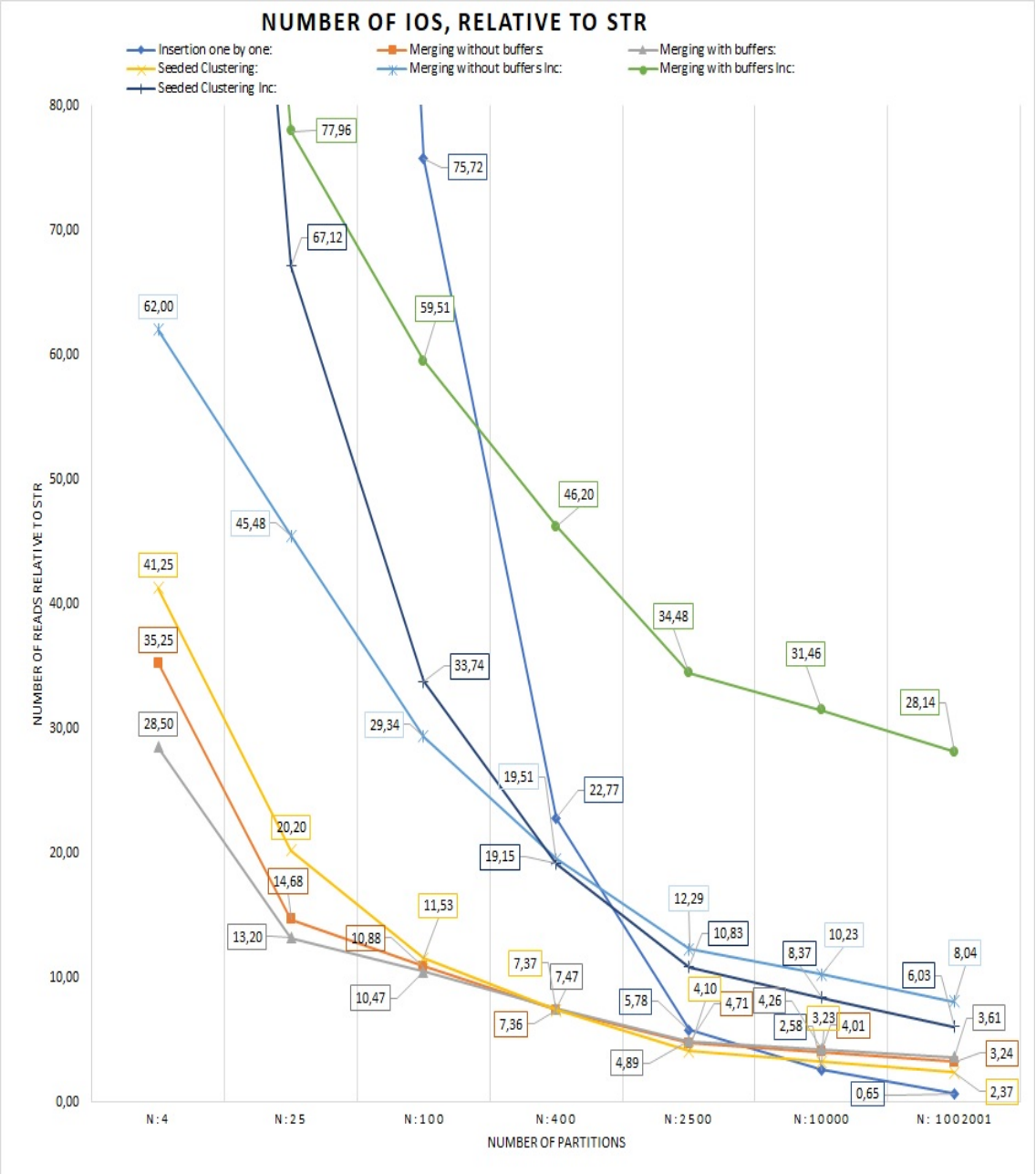


Figure 7.2: Number of IOs by querying the R-tree with 2D points, Number Reads Relative to STR:

Method	n:4 step 50	n:25 step 20	n:100 step 10	n:400 step 5
STR:	2 251 514.0	360 265.08	90 079.66	22 528.33
Inserting one by one:	2 253 105.75	360 540.88	90 155.38	22 551.10
Merging without buffers:	2 251 549.25	360 279.76	90 090.54	22 535.69
Merging with buffers:	2 251 542.50	360 278.28	90 090.13	22 535.80
Seeded Clustering:	2 251 555.25	360 285.28	90 091.19	22 535.70
Merging without buffers inc:	2 251 576.00	360 310.56	90 109.00	22 547.84
Merging with buffers inc:	2 251 716.50	360 343.04	90 139.17	22 574.53
Seeded Clustering inc:	2 251 707.75	360 332.20	90 113.40	22 547.48

Table 7.7: Number of IOs by querying the R-tree with 2D data points, uniform distribution, Part 1.

Method	n:2 500 step 2	n:10 000, step 1	n: 1 002 001 step 0.1
STR:	3 611.07	907.402	14.084
Inserting one by one:	3 616.85	909.985	14.733
Merging without buffers:	3 615.78	911.407	17.324
Merging with buffers:	3 615.96	911.657	17.692
Seeded Clustering:	3 615.17	910.637	16.455
Merging without buffers inc:	3 623.36	917.633	22.126
Merging with buffers inc:	3 645.55	938.864	42.226
Seeded Clustering inc:	3 621.90	915.776	20.114

Table 7.8: Number of IOs by querying the R-tree with 2D data points, uniform distribution, Part 2.

Observations:

The most apparent fact from looking at the results, is the degree of how much the merging with buffers approach deteriorates with increased number of merged components compared to the post processing counterpart. R-trees consisting of two merged R-trees perform at the same level, expect from the largest search areas where merging without buffers performs worse. Note that the performance gap between inserting one by one and STR drastically decreases with smaller search areas, this is due to the overlap observed in the directory nodes of the resulting R-tree. Since STR is the packing method used in Seeded clustering and the merging methods, they are affected in the same way.

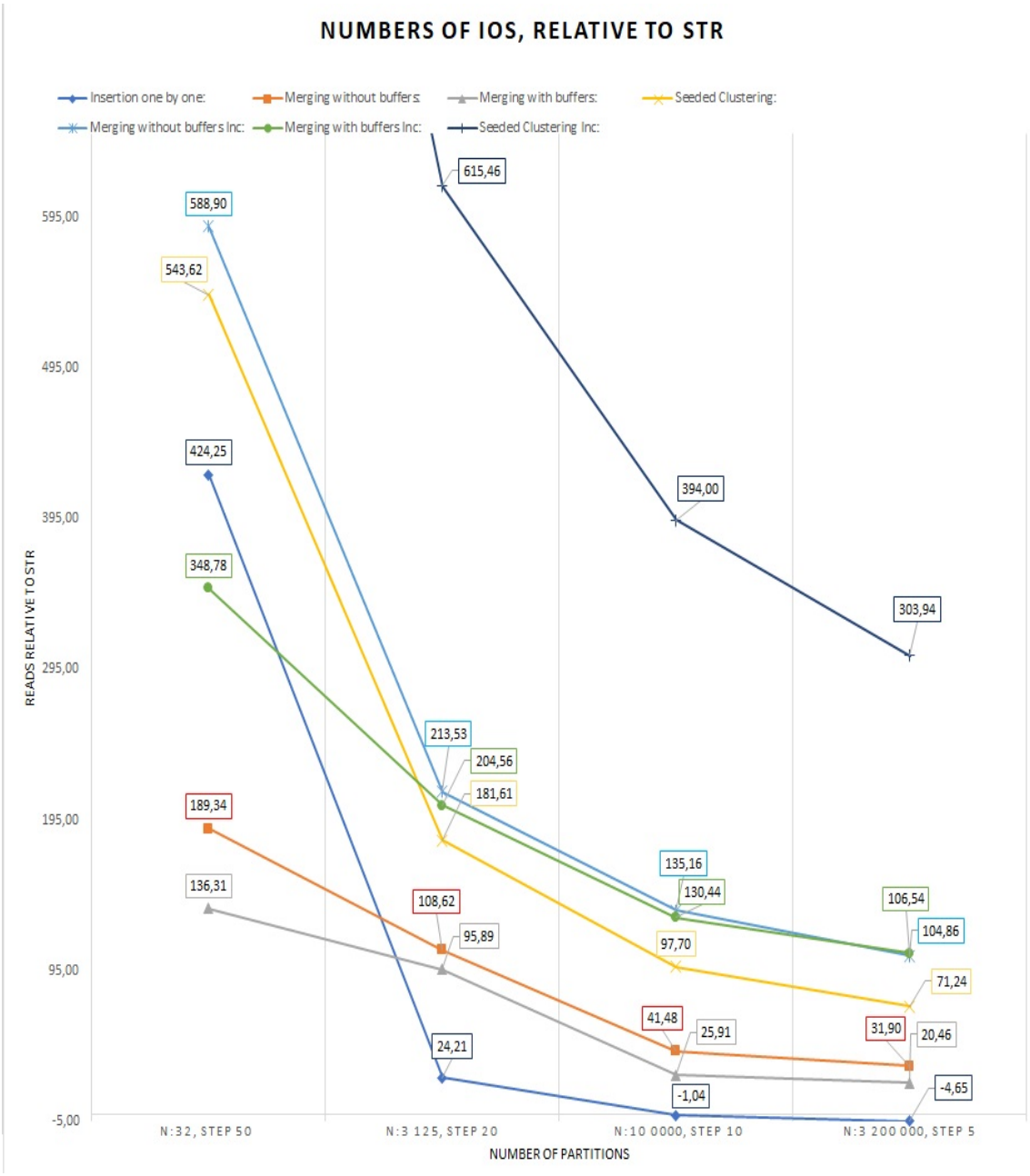


Figure 7.3: Number of IOs by querying the R-tree with 5D points, Number Reads Relative to STR:

Method	n:32 step 50	n:3 125 step 20	n:10 0000 step 10	n:3 200 000 step 5
STR:	281 862.94	4 130.23	241.44	38.561
Inserting one by one:	282 287.19	4 154.44	240.40	33.908
Merging without buffers:	282 052.28	4 238.85	282.92	70.464
Merging with buffers:	281 999.25	4 226.12	267.35	59.017
Seeded Clustering:	282 406.56	4 311.84	339.14	109.798
Merging without buffers inc:	282 451.84	4 343.76	376.60	143.418
Merging with buffers inc:	282 211.72	4 334.79	371.88	145.099
Seeded Clustering inc:	283 229.63	4 745.69	635.44	342.497

Table 7.9: Number of IOs by querying the R-tree with 5D data points uniform distribution

Observations:

For 5 dimensions we can observe the diminishing benefit of Sort Tile Recursive. The possible ways minimum bounding "hypercubes" can overlap increases with the number of dimensions. For queries with more fine grain partitions of the data space, i.e search regions with step size (length of sides of hypercube) less than 5, the same pattern of graphs from the previous dimensionality will emerge. A inserting fact is the performance gap between the STR and the merging methods. Seeded Clustering suffers the same performance gap to a larger extent. This shows merging and bulk insertion to unify indexed data sets becomes an increasingly difficult task with additional dimensions.

7.3.1 Construction time: Synthetic data

A new dataset was generated each time construction time was measured. This explains the drop in construction time for STR at 9 million in figure 7.5, due to the data points needed less work to be sorted.

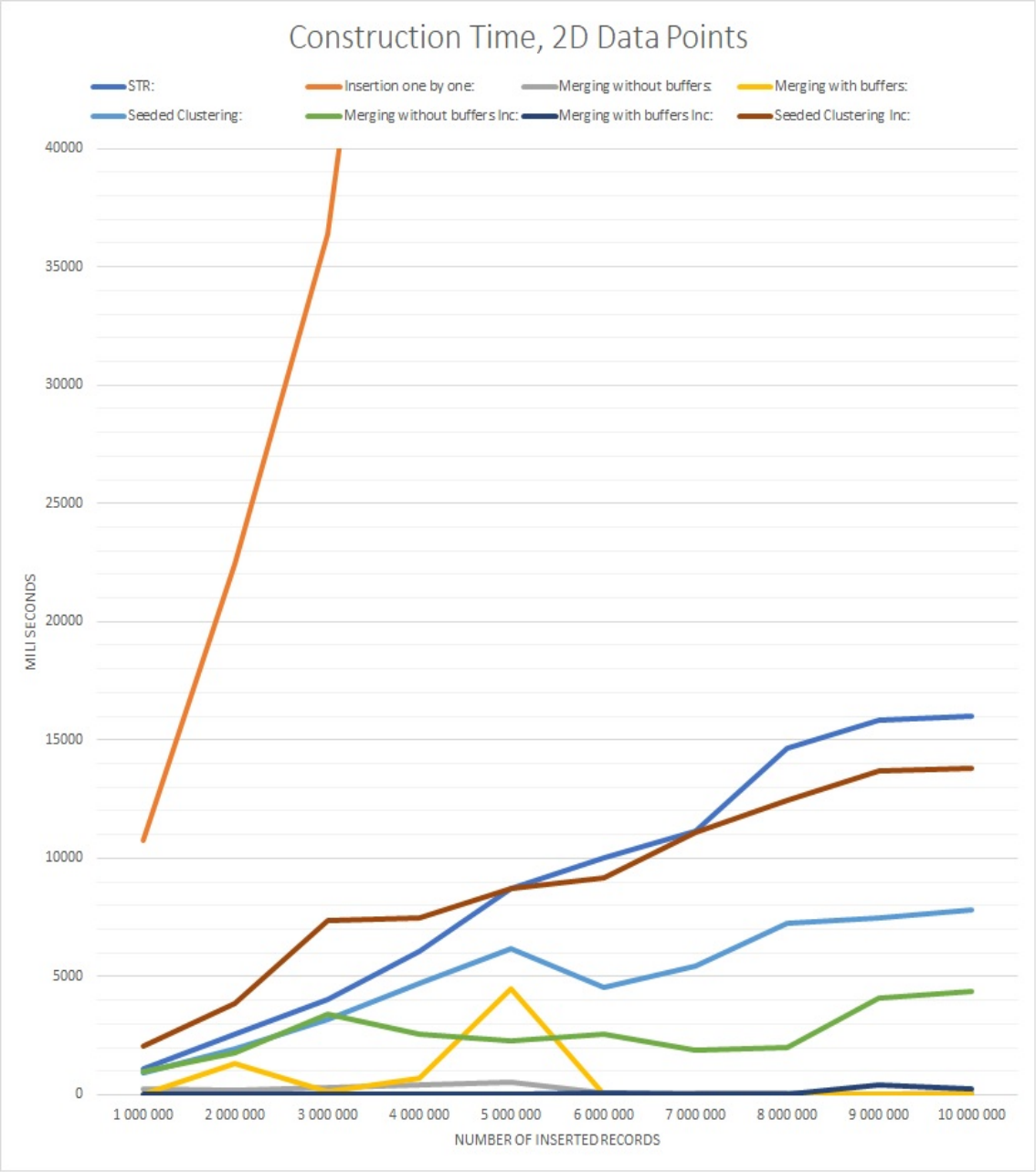


Figure 7.4: milliseconds for constructing the R-tree, increasing dataset of 2D Data points

Method	n:1000 000	n:2000 000	n:3000 000	n:4000 000	n:5000 000
STR:	1121,8	2576,8	4039,8	6063,2	8710,8
Inserting one by one:	10750,0	22433,8	36412,2	65285,8	71195,8
Merging without buffers:	246,8	168,4	293,0	397,0	513,2
Merging with buffers:	3,2	1340,6	157,8	692,2	4463,0
Seeded Clustering:	906,2	1927,0	3170,4	4736,2	6167,6
Merging without buffers inc:	987,6	1747,8	3420,2	2566,4	2301,6
Merging with buffers inc:	3,2	2,6	7,6	3,4	10,2
Seeded Clustering inc:	2078,0	3862,6	7353,4	7497,4	8723,4

Table 7.10: milliseconds for constructing the R-tree, increasing number of 2D data points in the Data-Set, Part 1

Method	n:6000 000	n:7000 000	n:8000 000	n:9000 000	n:10 000 000
STR:	10000,8	11140,0	14671,8	15846,6	16004,0
Inserting one by one:	84745,6	100668,6	120631,8	127896,4	135670,8
Merging without buffers:	58,8	85,8	62,2	51,8	57,6
Merging with buffers:	<0,0	1,6	1,2	<0,0	<0,0
Seeded Clustering:	4538,8	5421,4	7230,0	7482,8	7839,4
Merging without buffers inc:	2545,0	1914,2	2006,6	4066,8	4353,4
Merging with buffers inc:	87,8	4,4	4,6	394,6	237,8
Seeded Clustering inc:	9150,8	11066,8	12453,4	13713,6	13784,8

Table 7.11: milliseconds for constructing the R-tree, increasing number of 2D data points in the Data-Set, Part 2

Observations:

in the case of increasing number of entries in the dataset, it is clear that the merging strategies outperforms the other methods.

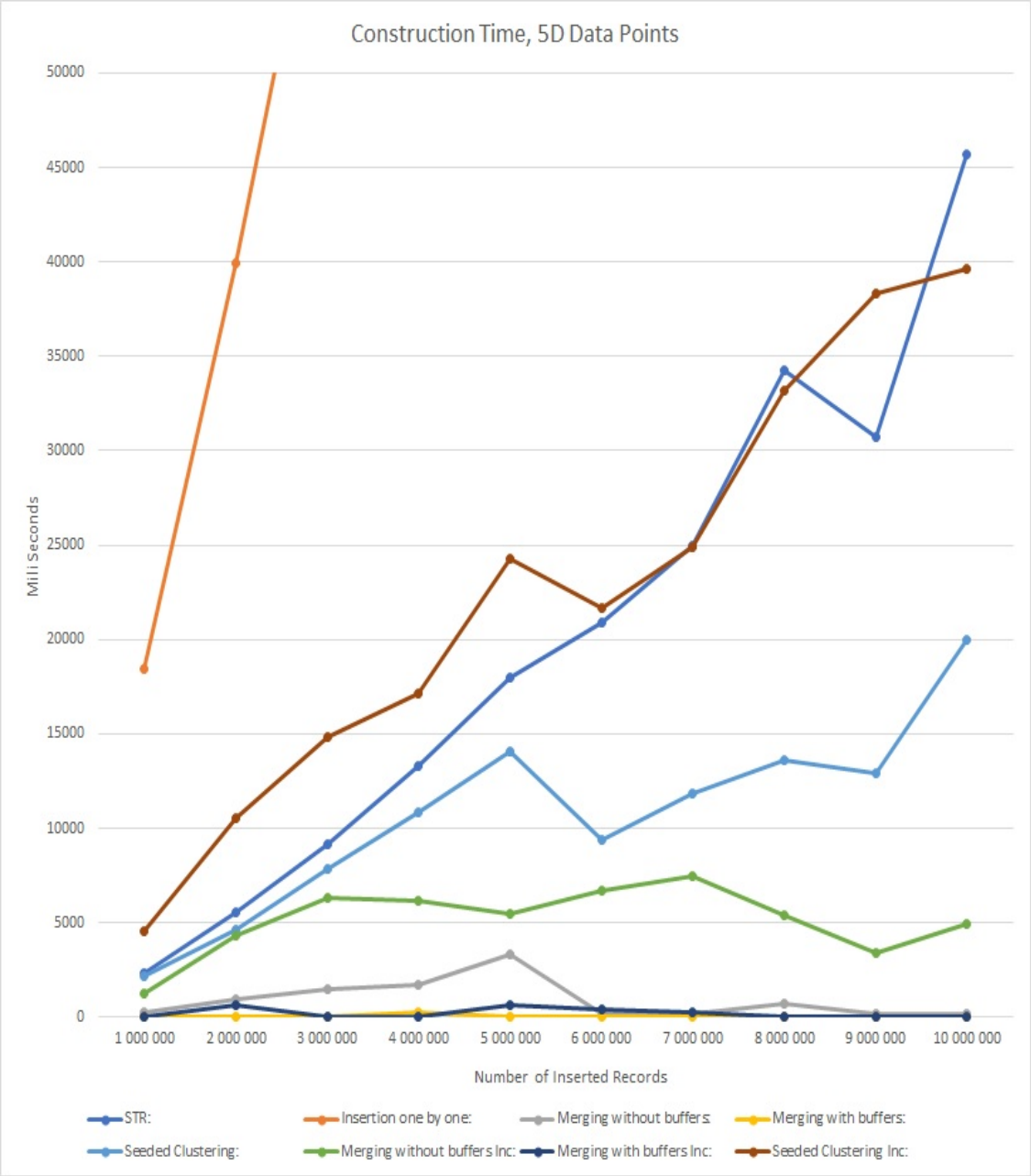


Figure 7.5: milliseconds for constructing the R-tree, increasing dataset of 5D Data points

Method	n:1000 000	n:2000 000	n:3000 000	n:4000 000	n:5000 000
STR:	2356,2	5533,0	9161,8	13324,2	17967,4
Inserting one by one:	18452,2	39937,6	63945,4	93551,6	121845,0
Merging without buffers:	254,8	960,8	1448,0	1713,0	3287,4
Merging with buffers:	3,0	4,0	5,0	233,8	9,0
Seeded Clustering:	2188,8	4636,6	7871,6	10851,6	14066,8
Merging without buffers inc:	1225,2	4343,6	6283,0	6192,0	5489,8
Merging with buffers inc:	2,0	667,2	5,2	6,6	607,0
Seeded Clustering inc:	4584,0	10521,0	14825,8	17118,4	24238,8

Table 7.12: milliseconds for constructing the R-tree, increasing number of 5D data points in the Data-Set, Part 1

Method	n:6000 000	n:7000 000	n:8000 000	n:9000 000	n:10 000 000
STR:	20911,8	24975,4	34280,2	30693,0	45725,0
Inserting one by one:	150126,6	183569,0	203925,6	210409,6	300574,8
Merging without buffers:	153,2	140,2	689,8	165,6	168,2
Merging with buffers:	0,6	<0,0	0,2	0,2	0,4
Seeded Clustering:	9374,0	11826,4	13570,2	12909,0	19996,2
Merging without buffers inc:	6685,6	7500,8	5416,0	3384,8	4942,0
Merging with buffers inc:	410,6	254,4	5,8	5,6	6,8
Seeded Clustering inc:	21644,8	24863,2	33197,6	38357,8	39601,6

Table 7.13: milliseconds for constructing the R-tree, increasing number of 5D data points in the Data-Set, Part 2

Observations:

We observe the same pattern as the 2 dimensional counterpart, however the rate of construction cost is steeper.

7.4 Discussion

The first issue to address is to what degree the merging methods was dependent on the packing algorithm implemented. STR tend to generate overlap in the directory/non-leaf nodes[51]. By examining the results, it seems that merged R-trees share the same characteristics as the method used to construct the individual composed components. If the R-tree-components have MBRs with high degree of overlap, MBRs that span a large area or have a severe degree of dead space, the merged tree would, to a certain extent, share those characteristics. Figure 7.8 illustrates the resulting MBRs from different methods for ingesting data into the tree. The structure of the R-tree is a result of the method used for construction, the distribution of residing spatial objects and possible restructuring methods, i.e reinsertions, repacking... etc, applied. Would it be possible to improve the structure of the merged R-tree by employing components that are optimized for merging? The bulk loading method by Böhm and Kriegel [15], that is referred to

in *merging R-trees* by Vasaitis, Nanopoulos and Bozanis [55], can be more suitable for merging, since they found no noteworthy difference in quality between the trees produced by merging and bulk loading. The experimental results presented here shows a clear performance gap in the resulting query performance between the the bulk loading method and both merging methods. The merged methods seem to have better query performance when each component tree is built from different sections of the data space, this is shown in the query performance of the real data set.

The merging methods outperformed Seeded clustering on quires with large search area. If the R-tree components was constructed by a top-down bulk loading method, with reduced overlap in the directory nodes, or a more optimal bottom-up method[24], the merged trees would outperform Seeded clustering on smaller search areas as well.

Would it be possible to examine or predict the structure of the R-tree components in order to avoid applying a merging algorithm that has a worse construction complexity then bulk loading the R-tree from scratch? The variance of construction cost for the *merging with buffers* method is striking, ranging from less than 1 milliseconds up to a few seconds. Given an unsuitable pair of R-trees, we might be better of rebuilding the whole dataset with Bulk loading. In the same way we can estimate the query performance of a R-tree [58], a priori of experimental evaluation, it is arguably possible to build a similar model with respect to merging.

Comparing the two merging methods it is not trivial to pick a winner as they both seems to have strengths and weaknesses. The merging without buffers method had more stable construction time, slightly better query performance on smaller search areas and less deterioration of the structure with repeated merges. the merging with buffers method on the other hand, had better query performance on larger search areas and higher dimensional datasets, in addition to not relying on re-insertions that limits concurrency control. Query performance was also good on skewed datasets.

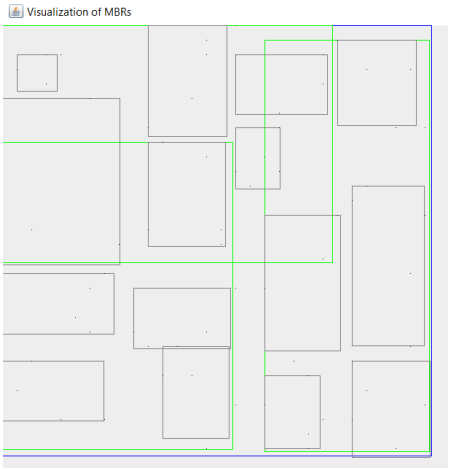


Figure 7.6: MBRs with STR

Figure 7.7: MBRs with inserting one by one

Figure 7.8: MBR comparison: Equal M and m, with same amount of entries

7.5 Further investigation

Further investigation was conducted to confirm the explanations for the results on the query performance. STR performed inadequately on queries with small search areas. In order to verify that the overlap in the directory nodes caused worse query performance, merged indices depend to a large degree on the packing method used for the components, and building components from smaller separate regions of the data space increase the query performance of the merged index. In order to create smaller and separate regions we sort the dataset on one dimension. A new variant of STR with the goal of minimizing the overlap in the directory node at the expense of space utilization, STR* was created for this purpose. STR* pack the leaves in the same way as STR, however the upper-levels are created by the generalized R* quadratic split. This packing method is unsuited in practice due to the high construction cost, in addition to worse performance on larger search areas. The merging methods used in this investigation consist of two components. The assertions made in the previous discussion were confirmed by the results, see Figure 7.9. Surprisingly, the **merging without buffers** on separate and smaller regions outperformed bulk loading with STR. Another surprising result is that separate and smaller regions deteriorated the query performance of **merging with buffers** on smaller regions. On larger search areas, seen on the performance of the real dataset with default ordering, smaller and separate regions are beneficial. We can deduce from the results that what merging method to choose, should be decided by the size of search areas in queries of the workload.

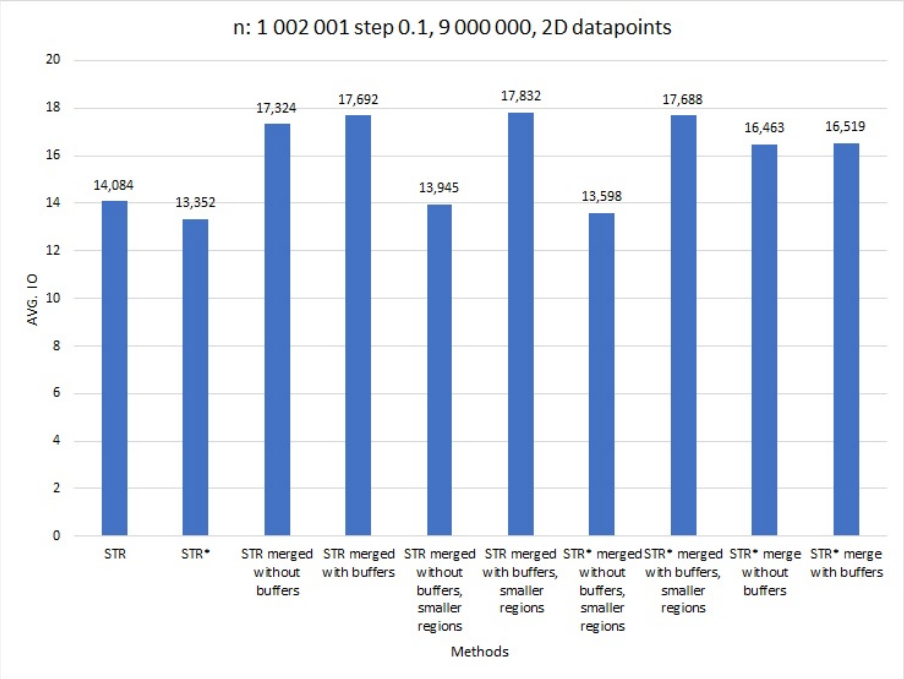


Figure 7.9: Average number of IOs for n:1 002 001, 2D datapoints

Method	Average IOs:
STR:	14,084
STR*:	13,352
STR packed, merge without buffers:	17,324
STR packed, merge with buffers:	17,692
STR packed, merge without buffers, separate and smaller regions:	13,945
STR packed, merge with buffers, separate and smaller regions:	17,832
STR* packed, merge without buffers, separate and smaller regions:	13,598
STR* packed, merge with buffers, separate and smaller regions:	17,688
STR* merge without buffers:	16,463
STR* merge with buffers:	16,519

Table 7.14: Average number of IOs for n:1 002 001, 2D datapoints

Chapter 8

Conclusion and Further Work

8.1 Conclusion

In this thesis, the access method R-tree and spatial data was investigated. We showed how to adapt the R-tree to meet new challenges and exploit multiple resources. Methods for ingesting large amount of spatial data were studied and implemented. Merging and equivalent methods was tested on real and synthetic dataset, with skewed and uniform distributions. Performance with respect to higher dimensions was also tested. The heuristics from bulk insertion was successfully adopted by a merging method devised in this project. merging with Post-processing the overlap and reinsertions proved to be better with increased number of components. Due to how the implemented procedures from bulk insertion tend to restructure the tree during insertions, concurrent control of the R-tree becomes limited. An extension of a merging method similar to the one from Vasilis Vasaitis [55] must be applied if we want to extend merging to parallel systems. It is shown in the results that merged R-trees inherit the characteristics from the R-tree components. Merging trees with entries spanning from smaller and separate can improve performance given the method and search size of the query. Merged R-trees can outperform bulk insertion, depending on the methods used to create the R-tree components to be merged. For some potential merging methods construction time may vary significantly. Overall, compared to equivalent methods merging gave a sufficient query performance with respect to the required construction time. The reason for parallelizing the bulk loading operations being the state of the art solution is the unbeatable query performance and the ability to horizontally scale with more available resources. An optimal bulk loaded R-tree seem to lose query cost optimality by occurring dynamic updates, bulk insertion or the tree is merged with another portion of the dataset. Given restrictions on downtime of the R-tree, we observe that it is possible to trade construction time at the expense of query performance, and vice versa. Merging remains an unexplored and promising territory.

8.2 Further Work

Investigate merging with components build from state of the art solution bulk loading methods is the natural next step. Given the variety methods to create R-trees, how to best combine bulk loading, merging and bulk insertion requires further examination. Merging methods for spatio-temporal data needs to be discussed. The experiments did not apply how blocks sizes influence performance, but rather chose a specific maximum and minimum fanout. usually, we tend to

select a fanout that suits the filesystem's block size.

8.2.1 Parallelizing Merging

If merging is to arise as an alternative to bulk loading, the methods must be adapted to work efficiently on a distributed or parallel environment given the amount and rate of spatial data generated. Assuming an architecture similar to what is presented in [11],[16],[24] and [14]. Given that the dataset is not known in advance and arrive in portions, merging can be a tool applied to newly packed portions of the dataset. It is not trivial how to best coordinate merging R-trees on a MPC model or a Master-Client R-tree. No work is known for this topic, and it is not entirely clear if the task is beneficial in terms of the resulting query performance. Given a skewed dataset, merging can be done in a similar way of building the upper levels of the R-tree by bulk loading[24] [11]. Further investigation is needed to find an optimal parallel R-tree best suited for merging. Considering the *merging with buffers* method implemented in this thesis, the downtime for the index happens when the *construction and splitting* phase occur. The results for the *parallel bulk loading algorithm using z-ordering*[24] described in **Related Work** had a downtime of 25.7 minutes to rebuild a R-tree with 100 million data objects. If a parallelized merging method were to exist, then it would require a far less downtime to be competitive with existing solutions, due to the potential loss of query cost optimality.

8.2.2 The Case For a Learned Cost Model In Spatial Databases

Learned cost models, described in [50], are generic access patterns synthesized into arbitrary optimal algorithms. As a bonus, they allow computing the performance of a data structure design on a target workload and hardware without explicit implementation and testing. similar ideas to learned cost models have been successfully applied in the LSM-based key value store Dostoevsky [37]. In other words, they made the LSM-based key value store *fluid*. The insight is that different LSM-merging methods tiering, leveling and lazy leveling all optimize for different workloads. Tiering are optimal of updates, lazy leveling for updates and point queries, leveling has best performance with respect to short range queries. The idea is to tune the different parameters of the LSM-tree to best suit the workload. The further work section of the paper from Stratos Idreos [50], describes a desire to expand the supported design space to include spatial data. It may be possible to apply ideas from a learned cost model to tune the parameters of the R-tree and choice of ingestion method, on the workload. The variants of bulk loading and insertion, in addition to the unexplored area of merging, result in different query performance and construction time. Different heuristics(i.e space utilization, minimizing overlap and enlargment...etc) give different optimizations with respect to the size of the search region of the query. Given the variety of methods to ingesting spatial data into a R-tree. A learned cost model would decide the optimal ingestion methods and data structures given distribution of spatial objects, dimensionality, size of the dataset, need for newest entries(i.e temporal constraints on incoming queries), workload(types of quires and update frequency), architecture, computational resources and memory constraints, numbers of concurrent users and associated features(i.e time-stamps and textual data). The goal is to find the best trade-off between the time to load the new data into the R-tree and the quality of the resulting data structure. The workload may vary across time and hardware may be specialized for the given application, we need the flexibility to tune the parameters of access method [33]. Although, Parallelized bulk loading methods gives the best query performance, flexibility is lacking. Having the option of multiple R-tree components that can be merged, bulk insertions and dynamic updates, provides flexibility on the trade-off line between query performance and construction time.

Chapter 9

Appendices

The code for the R-tree is found at https://github.com/thespooner/RtreeMethods

Bibliography

[1] Bigtable https://cloud.google.com/bigtable/. URL https://cloud.google.com/bigtable/.

[2] Apache cassandra http://cassandra.apache.org/. URL http://cassandra.apache.org/.

[3] Leveldb https://github.com/google/leveldb. URL https://github.com/google/leveldb.

[4] mpc-slides. http://grigory.us/.

[5] Rocksdb https://rocksdb.org/. URL https://rocksdb.org/.

[6] Foursquare https://foursquare.com, 2019. URL https://foursquare.com.

[7] Mohamed F. Mokbel Ahmed Eldawy. The era of big spatial data, 2015. URL https://www-users.cs.umn.edu/~mokbel/papers/clouddm15.pdf.

[8] Walid G. Aref Ahmed Mahmood, Sri Punni. Spatio-temporal access methods: a survey (2010 - 2017), 2018. URL https://www.researchgate.net/publication/328173829_Spatio-temporal_access_methods_a_survey_2010_-_2017.

[9] Hongzhi Wang Amina Belhassena. A survey on trajectory big data processing, 2018.

[10] Hongzhi Wang Amina Mina Belhassena. Parallel trajectory search based on distributed index, 2017. URL https://www.researchgate.net/publication/312144561_Parallel_Trajectory_Search_Based_on_Distributed_Index.

[11] Yannis Manolopoulos Apostolos Papadopoulos. Parallel bulk-loading of spatial data, 2003. URL http://delab.csd.auth.gr/~apostol/pubs/parco03.pdf.

[12] Lars Arge. The buffer tree: A new technique for optimal i/o algorithms, 1996.

[13] Brian Barrett. Pokemon go is doing just fine, with or without you, 2016. URL https://www.wired.com/2016/09/pokemon-go-just-fine-without/.

[14] Scott T. Leutenegger Bernd Schnitzer. Master-client r-trees: A new parallel r-tree architecture, 1998. URL http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=C60226BFD2E86194669357E53DE711EC?doi=10.1.1.31.2493&rep=rep1&type=pdf.

[15] Hans-Peter Kriegel Christian Böhm. Efficient bulk loading of large high-dimensional indexes, 1999.

[16] Markus Schmidt Bernhard Seeger Daniar Achakeev, Marc Seidemann. Sort-based parallel loading of r-trees, 2012. URL https://dl.acm.org/citation.cfm?id=2447489.

[17] Yannis Theodoridis Dieter Pfoser, Christian S. Jensen. Novel approaches to the indexing of moving object trajectories, 2000.

[18] Vipin Kumar Gowtham Atluri, Anuj Karpatne. Spatio-temporal data mining: A survey of problems and methods, 2017. URL https://arxiv.org/abs/1711.04710.

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

[20] Seema Bawa Hari Singh. A survey of traditional and mapreduce-based spatial query processing approaches, 2017. URL https://sigmodrecord.org/publications/sigmodRecord/1706/pdfs/04_surveys_Singh.pdf.

[21] Amina Mina Belhassena Hongzhi Wang. Parallel trajectory search based on distributed index, 2017. URL https://www.researchgate.net/publication/312144561_Parallel_Trajectory_Search_Based_on_Distributed_Index.

[22] Christos Faloutsos Ibrahim Kamel. Parallel r-trees, 1992. URL http://www.cs.cmu.edu/~christos/PUBLICATIONS.OLDER/sigmod92.pdf.

[23] Y. H. CHIN J. K. CHEN, Y. F. HUANG. A study of concurrent operations on r-trees, 1997.

[24] Yanchuan Chang Rui Zhang Jianzhong Qi, Yufei Tao. Theoretically optimal and empirically efficient r-trees with strong parallelizability, 2018.

[25] Daniel J. Abadi Jose M. Faleiro. Latch-free synchronization in database systems: Silver bullet or fool's gold?, 2017. URL https://15721.courses.cs.cmu.edu/spring2018/papers/07-latching/faleiro-cidr17.pdf.

[26] Nicholas Jing Yuan Yi Yang Kai Zheng, Shuo Shang. Towards efficient search for activity trajectories ; gat, 2013. URL http://vbn.aau.dk/files/71177462/ICDE13_conf_full_220_1_.pdf.

[27] Harold Kuhn. Hungarian algorithm. URL https://en.wikipedia.org/wiki/Hungarian_algorithm.

[28] Herman J. Haverkort Ke Yi Lars Arge, Mark de Berg. The priority r-tree: A practically efficient and worst-case optimal r-tree, 2004.

[29] Michele Lazzarini. Spatial big data in space and security, 2015. URL https://www.big-data-europe.eu/spatial-big-data-in-space-and-security/.

[30] Karen Lewis. Where's my stuff? how location and iot play well together, 2016. URL https://www.ibm.com/blogs/internet-of-things/location-iot/.

[31] Mohamed F. Mokbel Louai Alarabi. A demonstration of st-hadoop: A mapreduce framework for big spatio-temporal data, 2017. URL http://www.vldb.org/pvldb/vol10/p1961-alarabi.pdf.

[32] Patrick Valduriez M. Tamer Ozsu. Distributed and parallel database systems, 1996.

[33] Lukas M. Maas Radu Stoica Stratos Idreos Anastasia Ailamaki Mark Callaghan Manos Athanassoulis, Michael S. Kester. Designing access methods: The rum conjecture, 2016. URL https://stratos.seas.harvard.edu/files/stratos/files/rum.pdf.

[34] Jefferson R. O. Silva Mario A. Nascimento. Towards historical r-trees, 1998.

[35] Walid G. Aref Mohamed F. Mokbel, Thanaa M. Ghanem. Spatio-temporal access methods, 2003.

[36] I. Kamel Nick Koudas, Christos Faloutsos. Declustering spatial databases on a multi-computer architecture, 1996. URL https://www.researchgate.net/publication/221103345_Declustering_Spatial_Databases_on_a_Multi-Computer_Architecture.

[37] Stratos Idreos Niv Dayan. Dostoevsky: Better space-time trade-offs for lsm-tree based key-value stores via adaptive removal of superfluous merging, 2018. URL http://nivdayan.github.io/dostoevsky.pdf.

[38] Stratos Idreos Niv Dayan, Manos Athanassoulis. Monkey: Optimal navigable key-value store, 2017. URL http://nivdayan.github.io/monkeykeyvaluestore.pdf.

[39] Bernhard Seeger Norbert Beckmann. A revised r*-tree in comparison with related index structures, 2009.

[40] Ralf Schneider Bernhard Seeger Norbert Beckmann, Hans-Peter begel. The r*-tree: An efficient and robust access method for points and rectangles, 1990.

[41] Dieter Gawlick Elizabeth O'Neil Patrick O'Neil, Edward Cheng. The log-structured merge-tree (lsm-tree), 1996. URL https://www.cs.umb.edu/~poneil/lsmtree.pdf.

[42] Abhijith Kashyap Vagelis Hristidis Vassilis J. Tsotras Pritom Ahmed, Mahbub Hasan. Efficient computation of top-k frequent terms over spatio-temporal range, 2017.

[43] Abhijith Kashyap Vagelis Hristidis Vassilis J. Tsotras Pritom Ahmed, Mahbub Hasan. Efficient computation of top-k frequent terms over spatio-temporal ranges, 2017. URL https://dl.acm.org/citation.cfm?id=3064032.

[44] Li Chen Rupesh Choubey and Elke A. Rundensteiner. Bulk insertions into r-tree, 1998. URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.51.6547&rep=rep1&type=pdf.

[45] Li Chen Rupesh Choubey and Elke A. Rundensteiner. Gbi, a generalized r-tree bulk-insertion strategy, 1999. URL https://www.researchgate.net/publication/2268595_GBI_A_generalized_R-tree_bulk-insertion_strategy.

[46] Raghu Ramakrishnan Russell Sears. blsm a general purpose log structured merge tree, 2012. URL https://www.eecs.harvard.edu/~margo/cs165/papers/gp-lsm.pdf.

[47] Mario A. Lopez Scott T. Leutenegger, Jeffrey M. Edgington. Str: A simple and efficient algorithm for r-tree packing, 1997.

[48] Songnian Li Qing Zhu Xintao Liu Yeting Zhang Shengnan Ke, Jun Gong. A hybrid spatio-temporal data indexing method for trajectory databases, 2014.

[49] Hans-Peter Kriegel Stefan Berchtold, Daniel A. Keim. The x-tree: An index structure for high-dimensional data, 1996. URL https://dl.acm.org/citation.cfm?id=673502.

[50] Manos Athanassoulis-Niv Dayan Brian Hentschel Michael S. Kester Demi Guo Lukas Maas Wilson Qin Abdul Wasay Yiyou Sun Stratos Idreos, Kostas Zoumpatianos. The periodic table of data structures, 2018. URL https://stratos.seas.harvard.edu/files/stratos/files/periodictabledatastructures.pdf.

[51] Sukho Lee Taewon Lee. Overlap minimizing top-down bulk loading algorithm for r-tree, 2003. URL https://www.researchgate.net/publication/220920750_OMT_Overlap_Minimizing_Top-down_Bulk_Loading_Algorithm_for_R-treef.

[52] Sukho Lee Taewon Lee, Bongki Moon. Bulk insertion for r-tree by seeded clustering, 2003. URL http://dbs.snu.ac.kr/papers/dexa03bulk.pdf.

[53] Bernhard Seeger Thomas Brinkhof, Hans-Peter Kriegel. Parallel processing of spatial joins using r-trees, 1996.

[54] Jignesh M. Patel V. Prasad Chakka, Adam C. Everspaugh. Indexing large trajectory data sets with seti, 2003.

[55] Panayiotis Bozanis Vasilis Vasaitis, Alexandros Nanopoulos. Merging r-trees, 2004. URL http://delab.csd.auth.gr/papers/SSDBM04vnb.pdf.

[56] Apostolos N. Papadopoulos Yannis Theodoridis Yannis Manolopoulos, Alexandros Nanopoulos. *R-Trees: Theory and Applications*. 2006.

[57] Timos Sellis Yannis Theodoridis. Optimization issues in r-tree construction, 1994. URL https://pdfs.semanticscholar.org/b6ff/05250b1b50c03002bb00d8f807d026dfdc19.pdf.

[58] Timos Sellis Yannis Theodoridis. A model for the prediction of r-tree performance, 1996. URL http://www.dbnet.ece.ntua.gr/pubs/uploads/TR-1996-6.pdf.

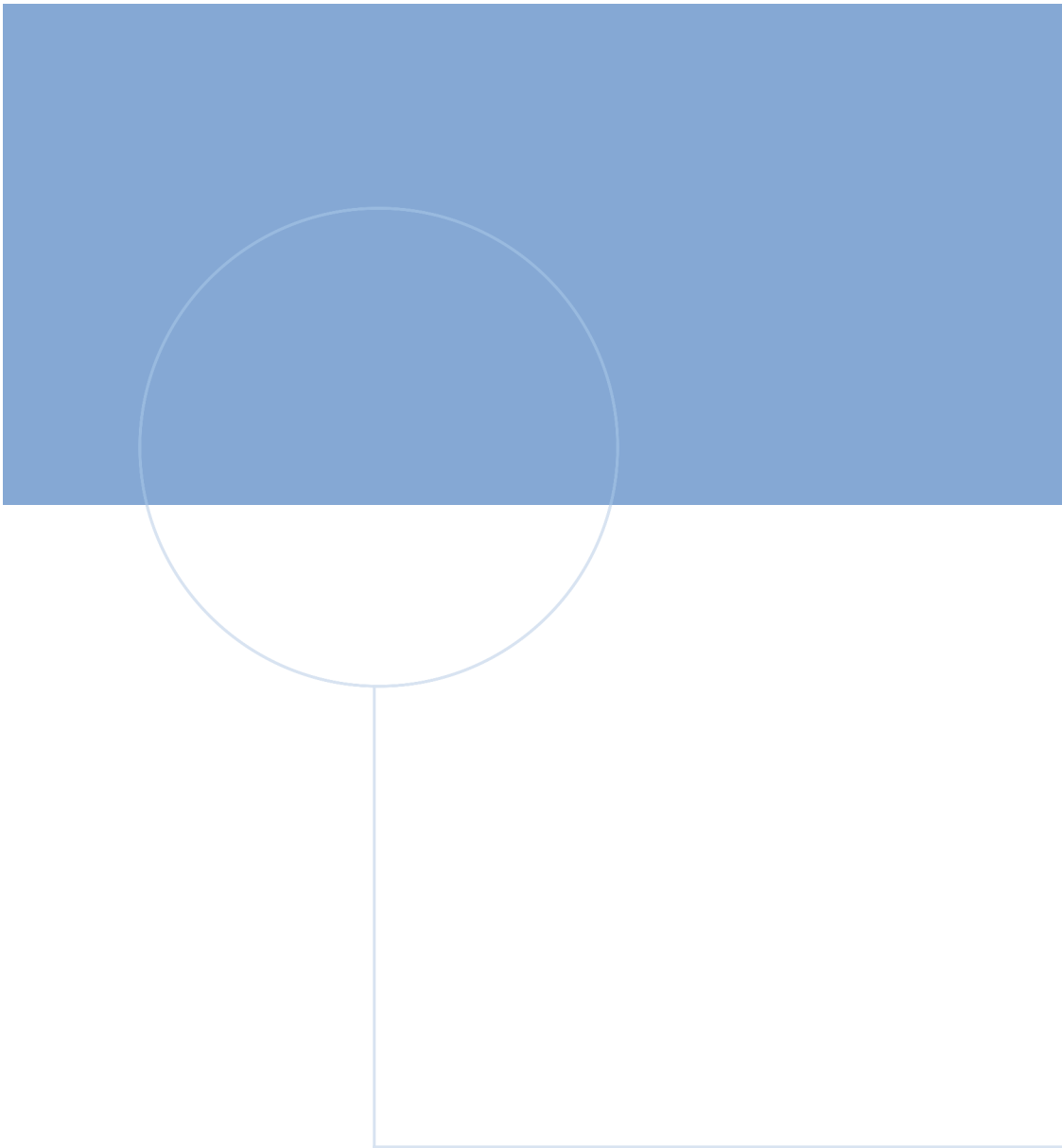
[59] Michael J. Carey Sharad Mehrotra Young-Seok Kim, Chen Li. Transactional and spatial query processing in the big data era, 2016. URL http://asterix.ics.uci.edu/thesis/YoungSeok_Kim_PhD_thesis_2016.pdf.

[60] Michael J. Carey Taewoo Kim Young-Seok Kim, Chen Li. A comparative study of log-structured merge-tree-based spatial indexes for big data, 2016. URL https://chenli.ics.uci.edu/files/icde2017-AsterixDB-Spatial-Comparison.pdf.

[61] Dimitris Papadias Yufei Tao. Mv3r-tree - a spatio-temporal access method for timestamp and interval queries, 2001.

[62] Mario A. Scott T. Leutenegger Yvan J. Garcia R. A greedy algorithm for bulk loading r-trees, 1998. URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.20.9245&rep=rep1&type=pdf.

[63] Scott T. Leutenegger Yvh J. Garcia R, Mario A. L6pez. On optimal node splitting for r-trees, 1998.



NTNU
Norwegian University of
Science and Technology