NTNU
Norwegian University of
Science and Technology

Similarity Search in Large Databases using Metric Indexing and Standard Database Access Methods

Erik Bagge Ottesen

Master of Science in Computer Science
Submission date: June 2009
Supervisor: Svein Erik Bratsberg, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Problem Description

Improving, experimenting with and implementing methods for performing similarity search in large databases using metric indexing and standard database access methods. With special emphasis on parallelization, distribution and continually increasing data volumes.

Assignment given: 15. January 2009
Supervisor: Svein Erik Bratsberg, IDI

Abstract

Several methods exists for performing similarity searches quickly using metric indexing. However, most of these methods are based on main memory indexing or require specialized disk access methods. We have described and implemented a method combining standard database access methods with the LAESA Linear Approximating Eliminating Search Algorithm to perform both range and K nearest neighbour (KNN) queries using standard database access methods and relational operators.

We have studied and tested various existing implementations of R-trees, and implemented the R*-tree. We also found that some of the optimizations in R*-trees was damaging to the response time at very high dimensionality. This is mostly due to the increased CPU time removing any benefit from reducing the number of disk accesses. Further we have performed comprehensive experiments using different access methods, join operators, pivot counts and range limits for both range and nearest neighbour queries. We will also implement and experiment using a multi-threaded execution environment running on several processors.

We found that the number of pivots, but most importantly the range limit, is the most important factors in reducing the number of candidates. This again reduces the number of data objects that must be fetched from disk and distance comparisons that must be performed. Selecting the appropriate range limit for a KNN query is especially important, and has an enormous impact on the response time. This can be made easier by using a range distribution histogram for the data set, allowing a reasonable range limit to be found for KNN queries using the expected result set size.

Our biggest contributions is the investigation of data sets and metrics, implementation and comparison of different access methods, specifically heap files, B+-trees and R*-trees, for use in metric indexing. Our experiments show that R-trees are well suited for this purpose, and perform significantly faster than B-trees. Specifically filtering can be performed more than five times faster, and both range and nearest neighbour queries have under half the response time when using R-trees. In addition we have shown that R-trees have a high advantage when using multi-processor and multi-core systems.

Preface

This report was written as a part of a master project for the Department of Computer and Information Science at the Norwegian University of Science and Technology. The report was written under the supervision of Svein Erik Bratsberg during five months during spring 2009, under the Data and Information Management section, Database Systems group, and as a part of the Information Access Disruptions (iAd) research group.

The report is largely based on work performed and results gathered during a specialization project during autumn 2008 by the same author and supervisor. In addition parts of the report is based on the experiments and final report written during the specialization project, both to give adequate background and context information.

The author would like to thank his supervisor Svein Erik Bratsberg for valuable guidance and assistance during the project. In addition Magnus Lie Hetland provided much insight and advice in the field of metric indexing.

Erik Bagge Ottesen
Trondheim, June 18, 2009

Contents

List of Figures

Chapter 1

Introduction

Similarity search is important in domains where canonical ordering of data is not possible, for instance multidimensional vector spaces or general metric spaces. In a general metric space only the distance between objects can be calculated, and as such it is trivial to return all objects within a given range or the K nearest objects. The domains we will investigate contain either large data volumes, or prohibitively expensive exact distance calculation, making a full scan to answer similarity queries unfeasible. Examples of data types include large text documents, images, video files, web pages or any heterogeneous data set where exact matches are not likely.

To solve the problem of reducing the number of needed distance calculations, several solutions for indexing metric spaces, known as metric indexing, have been proposed. However very few of these solutions allow direct adoption in a standard database context, and none have compared the usage of standard access methods and relational operators. Our report will investigate the usage of standard database access methods, specifically heap files, B-trees and R-trees, with standard join methods.

This will including indexing a large data volume in one or more index files using the different access methods. Performing queries using range predicates inherit to the given access methods, combined with standard join methods, to perform pre-processing or filtering. This will result in a candidate set that will then be post-processed in the application layer to discard false matches.

Our main motivation is to compare the performance of B-trees against R-trees for this purpose. This will mainly be a measure of how suited the usage of either multiple B-trees or a single or multiple R-trees of varying dimensionality, is at performing pre-processing in a metric space designed for read-mostly database activity. Heap files are also included as a baseline for comparing performance with a full file scan.

To give a better foundation we will first investigate various real-life databases from various sources. In addition we will look at a range of

different metric distance functions to see how they differ in their performance and statistical characteristics. We will show how the different distance functions are distributed differently even on the same data set.

Finally we will implement support for multi-threading in an attempt to both increase performance using multi-processor and multi-core systems, and lay the groundwork for a future effort to distribute the method over several computers in a shared-nothing environment.

Chapter 1 has given a short introduction to the report subject. Chapter 2 and 3 will give some background information on metric indexing and describe the theory and implementation of several related and existing methods. Chapter 4 will outline the theory and describe our method, while chapter 5 will give details of the implementation.

Chapter 6 will experiment with various data sets and metrics, and show how they differ. In chapter 7 the indexing and query methods will be investigated and experimented with, while chapter 8 will look at indexing and querying performed in a parallel on multiple processors. Finally a summarized conclusion and ideas for further work are given in chapters 9 and 10.

Chapter 2

Background

While traditional queries are performed either in a relational database for exact field matches, or in a full text search engine for word matches, similarity searches allows querying a much wider range of information. Similarity search is often performed in a non-vector space where only the distance between objects can be measured, such a space is known as a metric space. The search criteria is given using a query object, or prototype, and additional parameters depending on the query type. While exact matches are seldom feasable, the system will return the nearest matches to the query object and criteria.

In this chapter we will give some background information on the concept of metric spaces, especially with regard to similarity search and metric indexing. We will give special attention to pivot based filtering, and especially the LAESA Linear Approximating Eliminating Search Algorithm as this is used in the implementation. We will however assume that the reader is fairly comfortable with the subject and will not go into any detailed descriptions on either subject. For a more thorough description of metric indexing the reader is advised to read [39] or [13].

It is also assumed the reader has knowledge of database technology, and specifically database index methods. For a description of the database access methods used [27] or any text book about database management systems will provide the relevant information.

2.1 Metric Space

A metric space is a generalization of Euclidean space, where only a few properties about the objects in the space is known. Specifically the distance between objects can be measured, but the exact position of any one object can not be defined. For this purpose the function *distance function* $d(x, y)$ is defined for any objects x, y in the metric space. For the definition of metric space in this report we assume the following postulates hold true:

$$
\begin{aligned}
d(x,y) &\geq 0 && \text{(non-negativity)}\\
d(x,y) &= d(y,x) && \text{(symmetry)}\\
d(x,y) &= 0 \Leftrightarrow x = y && \text{(identity)}\\
d(x,z) &\leq d(x,y) + d(y,z) && \text{(triangle inequality)}
\end{aligned}
$$

2.1.1 Distance Measurements

The distance function $d(x, y)$ in a metric space measures the distance (or inverse closeness) between two objects in the given metric space. Distance functions can be either discrete, with a fixed set of possible values, or continuous, with unlimited or very large maximum values and possibly endless divisibility. Virtually unlimited distance functions exists, as these are often specific to both the data domain and application requirements. However, we will give examples of two common distance functions.

Levenshtein distance

Levenshtein distance, or edit distance, was first presented in [20] as a way to measure the distance between two strings. The measure is actually the number of edit operations that must be performed to transform one string into the other. Here an edit operation is defined as one of the following:

insert a single character is inserted into the string

delete a single character is deleted from the string

update a single character is deleted, and a new character inserted in the same position

This results in a discrete function, assuming that string lengths are constrained to a certain maximum. The edit operations can also be weighted, however to conform with symmetry insertions and deletions must be weighted equal. The results of the Levenshtein distance function is very dependent on string lengths, preferring short strings over longer ones. This can be handled by performing normalization as described in [38].

Euclidean distance

Euclidean distance is a well known geometric distance function that represent the shortest distance between two points. In a two dimensional space it is given by the function $d(x, y) = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2}$. This function is a *Minkowski distance* functions which can be defined generally for a

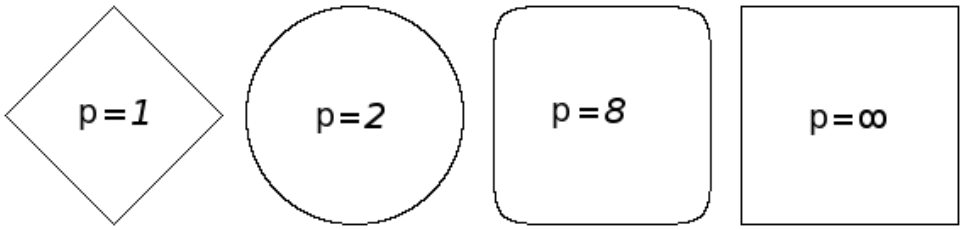


Figure 2.1: Displays the border at a constant distance from the center for various Minkowski distance functions L_p

n-dimensional space as:

$$L_p = \sqrt[p]{\sum_{i=1}^{n} |x_i - y_i|^p}$$

Euclidean distance is here the L_2 function, while the L_1 function is the city block, or $Manhattandistance$. This method assumes independence among coordinates when measuring distance, and as such will be influenced by codependent coordinates. See Figure 2.1 for a illustration of Minkowski distances in a two-dimensional space.

2.2 Similarity Queries

Several query methods exists in metric spaces. We will concentrate on the two most basic method, range queries and nearest neighbour queries. Other, more complex, query types exists and are in use. However, these will not be used in our implementation and will therefor not be discussed.

2.2.1 Range Query

One of the most basic query types that can be performed in a metric space is the range query. This returns all objects within a given distance from the query object. Any object with a distance $d(q, x) \leq r$ is included in the result set, while any object with $d(q, x) > r$ will be discarded. This can easily be performed naively by scanning the data set and calculating the distance between the query object and every data object, however this will cause a large amount of distance calculations, an operation that is often very expensive.

2.2.2 Nearest Neighbour Query

A nearest neighbour (NN) query finds the closest object in the data set to the given query object q. In other words it finds the object with the least

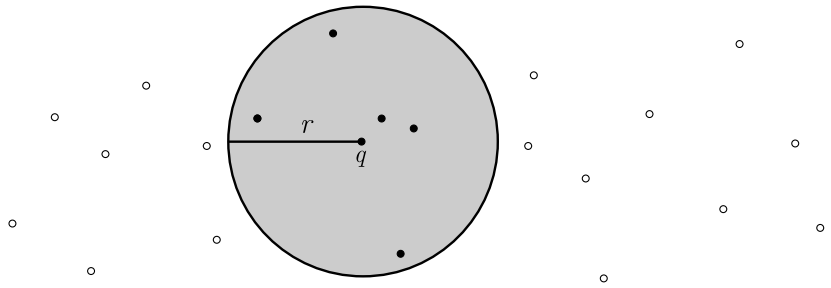


Figure 2.2: Range query in a two-dimensional Euclidean space. The data set is represented by dots, where the black dots are part of the result set. Five circles have distance from the query object $d(q, x) \leq r$, and are included in the range query with limit r.

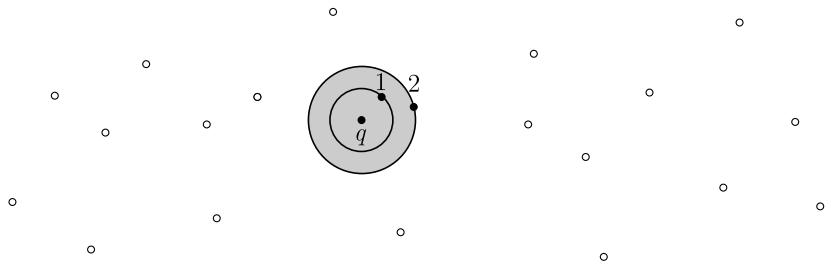


Figure 2.3: Nearest neighbour query with $k = 2$ in a two-dimensional Euclidean space. The two nearest objects are marked and will be returned as the result set.

distance $d(q, o)$, so that no other object exists with a smaller distance to the query object. More generally a K-nearest neighbour (KNN) query finds the K closest objects to the given query object. As several objects can be at the same distance from q, which is returned in a KNN-query containing one of the objects is undefined. Some implementations include all objects, even if thus exceeding the limit of K objects.

Several variations of this query type exists. For instance nearest neighbour queries can be incremental, continually returning the next nearest neighbour. Reverse nearest neighbour is also possible, returning all objects that has the given query object as one of its nearest neighbours. We will describe and implement the KNN query type.

2.3 Metric Indexing

Metric indexing is the concept of using indexing techniques to reduce response times for similarity queries. This is most often done by reducing the number of distance calculations, but for a disk based index reducing the number of disk pages fetched is just as important. Several methods exists for metric indexing, and we will describe a few in the next chapter. In this section some basic principles of metric indexing will be described, especially those related to the subject of the report. First we will briefly discuss partioning, before describing pivot based filtering and two related algorithms, AESA and LAESA.

2.3.1 Partitioning

Partitioning is used to dive the data set into two or more subsets. This allows queries to be performed on only a subset of the data, and possibly only a subset of the subset, and so on. Partitioning is the method used in indexing canonical data, for instance binary searches and trees divide the search space in half for each step. Metric spaces can also be partitioned, usually by the use of one or more pivot objects and dividing the data set according to the distance from the nearest pivot. However, due to the inexact nature of the similarity queries, it is not guaranteed that partitioning will in fact reduce the number of distance calculations or data pages that must be fetched by any relevant margin. However, this technique is used by many of the existing indexing methods.

2.3.2 Pivot Filtering

Pivot filtering consists of indexing the distance between each object and a set of pivots. The pivots can either be chosen from the data set, or be a separate set of real or fictitious objects, and can be chosen randomly or by a variety of different selection algorithms.

These pivots are then used during querying to reduce the data set to a candidate set, by eliminating objects that are known to be outside the query region. The candidate set will usually still have to be checked for exact matches, as false positives are fairly common.

AESA

The Approximating and Eliminating Search Algorithm (AESA) [30, 35] treats all objects as pivots, and indexes the distance between every pair of object. During querying a pivot object p is chosen from the pivot set and the distance to the query object calculated. From this a lower bound in the distance to all other objects is found by using the lower bound function $\check{d}$. All objects that have a lower bound higher than the current range limit can be eliminated immediately.

$$d(p,o) \leq d(p,q) + d(q,o) \Rightarrow d(q,o) \geq d(\check{q},o) = |d(p,o) - d(p,q)|$$

A new pivot is then chosen from the candidate set, and the filtering is performed again with the greatest lower bound stored in the candidate set. Note that the upper bound could also be used for fast inclusion, that is including objects that are known to be within the range limit. However as lower bounds are much tighter than upper bounds, and usually far more objects are discarded than included, lower bounds are the most important. With the set of used pivots $\mathbb{P}$ the lower and upper bounds are given by:

$$\check{d} = \max_{p \in \mathbb{P}}(|d(p,o) - d(p,q)|)$$
$$\hat{d} = \min_{p \in \mathbb{P}}(d(p,o) + d(p,q))$$

Pivots are usually chosen as the object with the lowest lower bound. As pivots near the query object will be more effective, it is intuitive to take a pivot that has the possibility of being the closest object. This is however not necessarily the most effective, as demonstrated by the improved performance in iAESA [10] by using a different pivot selection heuristic.

The filtering algorithm just described require a range limit to be known. For a KNN query, the range limit can either be estimated or set to infinity. On each subsequent distance calculation that results in an object that is within the K nearest objects, the range limit can be reduced to the Kth nearest distance, and candidates with a higher lower bound immediately discarded.

AESA is the best known algorithm for performing both nearest neighbour and range queries when measured in the number of distance calculations. This comes at a price of $O(n^2)$ memory consumption and similar worst-case processing time. The latter is reduced by Reduced Overhead AESA [36], while the former will be discussed shortly.

LAESA

Linear AESA [23] solves the main problems with AESA, of quadratic time and space complexity, by reducing the number of pivots. Instead of indexing the distance between all pairs of objects, only a fixed size subset is used as pivots. Query processing is performed as before, except the pivot used for filtering must be chosen from the pivot set and not the entire data set. This also means that pivots should be handled specially when it comes to discarding objects, or kept in a separate set where they can be utilized as pivots despite being discarded from the candidate set.

Using a reduced pivot set reduces the time and memory complexity to $O(np)$ where p is the number of pivots, while increasing the number of distance calculations needed during query processing. This is a trade-off between indexing time and memory usage against query processing time. The tightness of the lower bound is also reduced by the use of non-optimal pivots for the query object.

Chapter 3

Related Work

There are several existing methods for indexing a metric space to provide fast range and nearest neighbour queries. However, most of the existing methods are designed for use in main memory and does not work especially well when either the data set or index structure does not fit in main memory. Of the remaining methods that do allow the index structure to grow beyond the available main memory, specialized access structures are often required for the index files, making them unsuitable for use in a standard database system.

There exists however a few notable methods that work well within the framework of database systems, most of these are based on B-trees or R-trees similar to the method we will propose. We will discuss the most relevant methods in this chapter. For a more detailed survey of existing methods see [13].

3.1 AESA

We have already described AESA [30, 35] and LAESA [23] in section 2.3.2. AESA is the fastest known nearest neighbour algorithm, however slightly surpassed by its variant iAESA [10] that modifies how the pivots are selected during query processing. The main issue with AESA is a quadratic time and space complexity, making the solution infeasible for larger data sets. This problem is solved by LAESA which utilizes a fixed number of pivots that is much fewer than the number of data objects, and as such only incurs linear time and space complexity with only a slightly higher query processing cost [28].

Several algorithms have improved on, or is based on, the LAESA method. TLAESA [22] reduces the processing CPU time by utilizing a tree structure for storing the distances. Spaghettis [5] stores the distance matrix in columns ordered according to distance from the pivot, with pointers between the cells to allow intersecting the arrays of different pivots. This is

similar to our method of storing the distances from each pivot ordered by distance with a key that can be used for intersection and retrieval of data objects.

3.2 M-tree

The M-tree [7] is a disk based, balanced tree structure, very similar to B-trees and R-trees. The tree contains several nodes, each node contains a series of objects that are all contained inside a sphere or ball, similar to bounding regions in R-trees. The tree is built in a bottom-up fashion, inserting items into the single root node until it is full. The node is then split in two and both parts (now leaves) are inserted into a new root node. This is equal to the way both B-trees and R-trees are built, and ensures a minimum fill factor of 50% in all nodes (except the root).

Several variations of the M-tree have been proposed, one of the most compelling is the Slim-tree [4]. The Slim-tree defines the fat-factor of a tree as a measure of the overlap between nodes and their metric regions. It uses new insertion and split algorithms to reduce the fat-factor, and as such improve both insertion and query performance greatly. In addition a fat-reduction algorithm called the Slim-Down algorithm is specified that attempts to reduce the fat-factor of an existing tree by moving outlier objects from a node to its overlapping siblings. This algorithm can be used on most other variants of the M-tree, and in fact also on other trees using overlapping regions, for instance R-trees.

The Pivoting M-tree [33] combines an M-tree with the LAESA algorithm to allow better pruning of nodes. This variant stores a matrix of precomputed distances between a set of pivots and the data objects in each node. While this will increase the number of disk accesses due to an increased storage need, it has a positive effect on the number of distance calculations that must be performed.

The Metric B+-tree [15] is not based on the M-tree, but a variant using a standard B+-tree as the data store and a block tree for auxiliary information. The data is partitioned using slices rather than spheres as in M-trees, leading to less overlap and thus potentially increased performance due to fewer nodes to traverse. The unique aspect of the MB+-tree is the usage of standard disk based access methods. There is however little evidence of how well it will fare in complex metric spaces where their approximation might not work as well.

3.3 Omni-concept

Omni-concept [11] is based on the selection of several *foci* (pivots), and indexing the distance from all objects to all pivots. Range queries can then be

performed using range queries from each foci and intersecting the results, while KNN queries can be performed with a predefined or estimated range query followed by a post processing step. This is in fact an implementation of LAESA, despite the authors seemingly being unaware of the existence of LAESA.

The scheme can be implemented on top of several existing indexing structures, and the authors describes the usage over sequential files, B+-trees and a single R-tree. Each have its own specialized implementation, and as such the Omni-concept as described in [11] is not a general scheme that can be implemented on any access structure.

This is the existing method that most resembles the method described in this paper. The major difference is our focus on comparing the B-tree and R-tree indexing structures, as well as our methods generality allowing it to be used with virtually any access structure. We add the possibility of using several R-trees of a fixed or variable dimensionality to increase performance, parallelism and high-dimensionality scalability.

3.4 iDistance

iDistance was first proposed in [37] and later refined and compared with both Omni-sequential and M-tree in [16]. It is based on using several pivots (reference points) and partitioning the data set according to distance to the nearest pivot, a ball partitioning scheme using several pivots. The unique concept of iDistance is how the distances are stored. The distance between pivots and their data objects are stored in intervals as a part of one large B+-tree. Each interval of the B+-tree containing the pivot and all data objects with this pivot as its nearest, ordered according to their distance from the pivot.

This allows queries to be performed as one or more range queries within the single B+-tree. One range query must be performed for each partition that the query range intersects with, and the union of the result of all the range queries are returned for exact distance calculation. For KNN queries the range limit is gradually extended until K objects are found within the limit. As only the distance between each object and a single pivot is known, the lower bound will usually be very loose except when the query object is very close to a pivot. This makes using lower bounds for finding and pruning nearest neighbours, similar to AESA, infeasible. Thus leading to more distance calculations than other, similar methods.

The experiments performed with iDistance prefer comparisons of page reads and response time, and does not take any account of the number of distance calculations. All the experiments are also performed using Euclidean data sets, ensuring an insignificant distance cost. With data sets of this type, sequential scan becomes one of the most effective methods and

the purpose of metric indexing is lessened. In fact iDistance aims to reduce the number of page accesses at the cost of increased number of distance calculations. This conflicts with experiments performed for most other methods, as they usually concentrate on reducing the number of distance calculations, and makes it harder to compare iDistance with other methods as well as our own.

Chapter 4

Indexing and Query Method

Several methods for performing similarity search have been proposed, and implemented, in earlier works. While most of these concentrate on memory only systems, there are also a few that work with disk based systems. Some are even suited for implementation in database systems, especially the many M-trees based methods. However, few or none of these are suited for use with the currently available commercial database management systems.

The method we will propose is actually an implementation of earlier ideas using only what is available in a standard database management system. Our method uses only standard database access methods available in a large range of currently used systems, in addition to a combination of standard relational operators and custom operators to handle exact distance calculations and KNN-queries more effectively. The latter can easily be implemented in the application layer, with the database layer responsible for the filtering and the application layer responsible for post-processing, including application-specific exact distance calculations.

4.1 Indexing

Our database contains one data file and one or more index files. The data file contains all objects and an unique key for the object, that is it contains a set of tuples (i, o_i) for each data object in $\mathbb{D}$. The index is based on one or more independent pivots, equal to the index in LAESA [23] only with the index stored in files on disk. The number of index files depends on both the number of pivots and the access method used.

When a pivot is added to the index, a new index file for this pivot is created. Then the distance between the pivot and all data objects is calculated, and the distance and key added to the index file. That is for each object a tuple $(d(p, o_i), i)$, where i is the object key, is added to the pivots index file.

How the objects are ordered in the index file depends on the access

method, they can be added consequently as they are indexed, or they can be ordered on the result of the distance function. The formed is used for our heap file based implementation, while the latter is used for the B-tree based implementation.

Pivots can also be added in a group, either with separate index files or, in the case of spatial access methods, combined in a single file. This is used for the R-tree access method described later. Adding several pivots at once saves disk reads from the data file, as the data file only needs to be scanned once for the entire group instead of once for each pivot.

4.2 Number of Pivots

It is important to note that the result size for both range queries and nearest neighbour queries remains constant and is independent of the number of pivots used. This is because the filtering process guarantees that *all* data objects within the limit is returned, but gives no guarantee that every object returned is in fact inside the range limit. The number of pivots is therefor only related to how many candidates outside the limit will also be returned and will have to be purged by post processing. This is also true for nearest neighbour queries, as only objects inside the range limit will be added to the result set despite being a part of the candidate set.

We then have that the only difference, with regard to both query types, is how many distance calculations must be performed during filtering and in post processing. During filtering the number of distance calculations equals the number of pivots, as the only distance that is calculated is between pivot objects and the query object.

For post processing the number depends on the query type. Range queries will need to process every candidate, and as such will need as many distance calculations as there are candidates. Nearest neighbour queries are a bit more complex, and depend on the quality of the lower bound estimates. Better lower bound estimates allows the post processing to terminate faster, as it can purge all candidates with a lower bound higher than the distance to the Kth nearest neighbour. In general the more pivots are used for filtering, the better the lower bound estimates will be.

4.3 Pivot Selection

For the initial pivot we employ the technique described in [11]. A data object is chosen at random, or simply using the first object in the data set, and the distance from this object to all other objects are calculated. The object that is furthest away, that is it has has the highest distance $d(o_1, o_i)$ where o_1, is the selected object, and will be used as the first pivot. The initial pivot thus requires $n = |\mathbb{D}|$ distance calculations to be found.

For selecting further pivots, we aim at exploiting the known information regarding distance between the set of all objects and all current pivots. In order to maximize the pruning power of the newly selected pivots, any new pivot candidate should be as far away from the current pivot set as possible. This is done by scanning all the existing pivot indexes and compiling a list of all objects and their distance to the nearest pivot. The object that has the highest minimum distance, in other words is furthest away from their nearest pivot, will be the new pivot candidate. Using the notation introduced earlier, we select the candidate o_i with the highest value of $\min_{p \in \mathbb{P}}(d(p, o_i))$.

Our method is similar to the method used in [11] as well as the incremental selection method described in [3]. Unlike the latter however, we rely solely on maximizing the distance between existing pivots and the new candidate. This results in the selection of several outliers as pivots, something that while suitable in a Euclidean space, is not optimal in a general metric space [3, 25]. Our method however has the advantage of determinism and fast pivot selection without performing a single distance calculation during pivot selection.

A better solution to the pivot selection problem in non-Euclidean metric space is selecting pivots that are suitably far apart, instead of maximally separated, as done in [25]. This greatly reduces the number of outliers that become pivots. Our algorithm can similarly be extended by selecting candidates that have $\min_{p \in \mathbb{P}}(d(p, o_i)) > M\alpha$ as in [25], while still exploiting the known information and eliminating the need for distance calculations during pivot selection.

4.4 Filtering

The filter process must be supplied with two parameters, the query object q and a range limit r. In addition a set of index files supporting range queries are needed. The result of the filtering is a candidate set, and filtering is most often a pre-processing step to reduce the number of data objects before the actual query is processed.

The filter process starts by performing a range scan on each index file. For a data object to be withing the given range, the distance from the pivot to the data object must also be within a given range. This is given by the triangle inequality, and results in the inclusion of only objects that satisfy $|d(p, q) - d(p, o_i)| \leq r$. As $d(p, o_i)$ is pre calculated and stored in the index file, only $d(p, q)$ must be calculated and all objects between $d(q, p) - r$ and $d(q, p) + r$ are returned as candidates for this pivot. This also allows us to find the lower and upper bounds as seen from this pivot, without performing any distance calculations, respectively $\check{d} = |d(p, o_i) - d(q, p)|$ and $\hat{d} = d(p, o_i) + d(q, p)$.

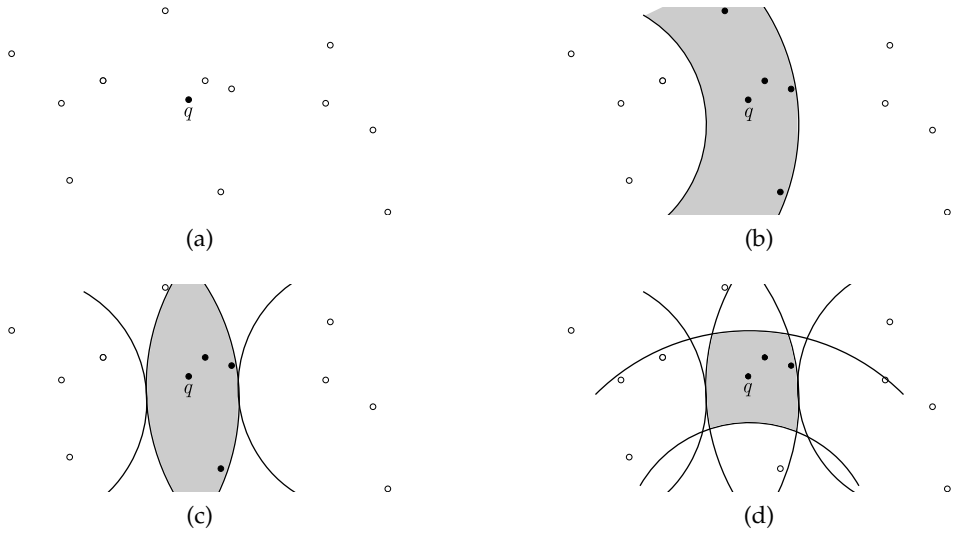


Figure 4.1: Range query filtering process in a two-dimensional Euclidean space using three pivots: (a) The query object with surrounding data objects; (b) First pivot, objects outside the range $d(p_1, q) \pm r$ are discarded; (c) Two pivots, objects outside $d(p_2, q) \pm r$ are also discarded. (d) Three pivots, only candidate set remains.

The candidate set returned from each index file is joined with the candidate set of every other index file, and only the objects that exists in every candidate set is returned. In practice this is done by intersecting the candidate set from the first index file with the candidate set of the second index file, and further intersecting the result with the next index file until the final candidate set. This is shown in Figure 4.1 using three pivots.

The lower bound distances returned from each index file is combined and returned as a single greatest lower bound, that is $\check{d} = \max_{p \in \mathbb{P}}(|d(p, o_i) - d(q, p)|$, while the upper bound is discarded. In a relation database this is performed using a series of range scans for each index, joined using an equi-join operator and with the highest distance returned for each join. The result of the filtering process is a set of tuples $(i, \check{d})$ containing the object key i and the lower bound distance $\check{d}$ between the query object and each data object. The lower bound can be used to reduce the number of distance calculations needed in post-processing steps, as will be seen for nearest neighbour queries.

4.5 Range Query

Range queries must be supplied with two parameters, the query object q and a range limit r. The query will return any object o that has $d(q, o) \leq r$,

as well as the exact distance from the query object to each object in the result set. The query is performed in two separate steps. The first is a filtering process as described above, and the second an exact distance calculation post-processing on each of the candidates resulting in a final result set including only objects that are known to be within the range limit.

Post-processing is performed on each candidate as it is returned from the filtering process, this means that filtering and post-processing can be performed in parallel. For each candidate, the data object is fetched from the data file using the key, and the exact distance between the query object and the data object is determined. If it is within the range limit, that is if $d(q, o_i) \leq r$, the object will be included in the result set, otherwise discarded. The result set contains tuples $(o_i, d(q, o_i))$ with the object o_i and the exact distance between the query object and each object. The order of the result set is undefined, and independent of both the insert order and distance to query object.

The cost of the query depends on the range limit and number of pivots, as well as the data set, and will be further explored in chapter 6 and 7. The number of distance calculations is given by $n_{distance} = n_{pivots} + n_{candidates}$, where n_{pivots} distance calculations must be performed to find the distance between the query object and each pivot during filtering, while $n_{candidates}$ are performed during post-processing to find exact distances to the result set and discard the remaining objects.

4.6 Nearest Neighbour Query

Nearest neighbour (NN) query or K nearest neighbours (KNN) is supported by first performing a range filter to remove objects from the data set that are too far away to be interesting. Both parameters for the filter process, a query object q and a range limit r, must be supplied, in addition to the maximum number of nearest neighbours k that should be returned.

The range criteria used in the range query can either be determined from the intrinsic dimensionality as in [11], or it can be provided by the user. It can also be found by trial and error, performing an incremental search with increasing query ranges until the wanted number of candidates are returned. Too large a range criteria will lead to a large candidate set that needs post-processing, and too small a range criteria might lead to too few candidates and as such less than the K nearest neighbours returned.

After the initial range-based filtering is performed, the exact distances needs to be calculated and the candidates ordered according to shortest distance to the query object. This is done by first ordering the candidate set ascending according to the lower bounds that are returned from the filtering step. Then the exact distance is calculated for each candidate and the candidate moved to an ordered list limit of K entries. The ordered list is

i	$\check{d}$	d
9	0.02	-.–
3	0.04	-.–
6	0.12	-.–
1	0.32	-.–
5	0.40	-.–

k	i	d
1	-	-.–
2	-	-.–

(a)

i	$\check{d}$	d
9	0.02	0.22
3	0.04	-.–
6	0.12	-.–
1	0.32	-.–
5	0.40	-.–

k	i	d
1	9	0.22
2	-	-.–

(b)

i	$\check{d}$	d
9	0.02	0.22
3	0.04	0.05
6	0.12	-.–
1	0.32	-.–
5	0.40	-.–

k	i	d
1	3	0.05
2	9	0.22

(c)

i	$\check{d}$	d
9	0.02	0.22
3	0.04	0.05
6	0.12	0.12
1	0.32	-.–
5	0.40	-.–

k	i	d
1	3	0.05
2	6	0.12

(d)

Figure 4.2: KNN query example with $k = 2$. Top table contains candidates, and bottom table contains the result set for each step. (a) 5 candidates returned from filtering, sorted on lower bound distances $\check{d}$, and empty result set. (b) First candidate has distance $d_9 = 0.22$, and is added to the result set. (c) Second candidate has distance $d_3 = 0.03$. (d) Third candidate has distance $d_6 = 0.12$, object 9 is discarded from the result set and replaced by 6. As all further candidates have lower bounds $\check{d}$ exceeding $d_6 = 0.12$, the process terminates with only 3 distance calculations performed.

also what will become the result set, and will always contain the K nearest objects found up to this point. Any object that is found to be further away than the Kth object currently in the result set is discarded, as it is known that at least K objects are closer.

Further, the exact distance calculations can be stopped, and all further candidates discarded, once the lower bound of the next candidate exceeds the Kth entry in the result set. This is because all further candidates have a lower bound higher than the K objects that are currently in the result set, and as such must have a distance that is also greater than each of these objects. This post-processing is illustrated in Figure 4.2.

For this to work efficiently a good estimate for the range query is required, as too low a range limit will result in too few candidates to guarantee a result set if size K. Even if more than K candidates exists, all the candidates with a higher distance than the range limit must be discarded as there might exists other objects closer to the query object that have been filtered away. As such it can not be guaranteed that the result set contains K objects, but the objects in the result set can be guaranteed to be the nearest neighbours and in increasing order of distance.

The post processing described here is identical to the algorithm used in LAESA [23] and k-LAESA [24], and results in far fewer actual distance calculations than number of candidates returned from the filtering process. As

the presorting of candidates according to lower bound requires all objects to be returned from the filtering process before post-processing begins, it can not be performed in parallel with the filtering process. The exact cost of the KNN query depends on how many candidates must be tested before termination, however the cost can be upper bounded for the worst case with the same cost as for range query described in the previous section. The average cost is usually less than half of this for normal queries.

Chapter 5

Implementation

The methods described in the previous chapter has been implemented in order for experiments to be performed and for the usage of future applications. The implementation is done in the Java programming language, and is platform independent with the prerequisite of supporting the Java Virtual Machine with access to a file based storage device. To ease development of the database layer, the database framework is based on the NEUStore package [9].

In this chapter we will detail how the various parts of the database and application have been implemented. We will start with the database layer, and then explain the indexing and query processing that is performed both in the database and application layers.

5.1 Database Framework

The database framework creates a framework for supporting a disk based database with varying access structures, query processing methods and buffer handling. This is based on NEUStore, *a Java package that aims to support the development of disk-based, paginated, and buffered index structures.* NEUStore provides only the basic building blocks for a database system, such as page and buffer handling. In addition example implementations of heap files and buffers are provided. While lock handling is supported for atomic operations, transactions are not used.

On top of this package we implemented several access structures, query processing methods, join methods and relational operators. As we have based our system on the NEUStore package, we are also limited to some of the limits in this package. Specifically NEUStore assumes distinct keys, and handles keys and data objects differently. It also assumes that all keys have a total ordering, while data objects need only be comparable for equality. To be able to fulfill our requirements the division between key and data will be loosened, while some access methods will only allow a certain type

of keys.

5.1.1 Buffer Handling

Buffer handling is performed during both read and write operations on any database page. The buffer has a fixed number of pages that can be held in memory, and must also control which pages are purged when the buffer is full. For read operations the page can either be fetched from disk or from memory, while write operations must additionally decide when and how to write the page to disk. There are also many methods for deciding which page should be swapped out when the buffer is full.

In our experiments we have used a naive Least Recently Used buffer implemented in the `LRUBuffer` class, where the page that was last accessed is removed from the buffer whenever it is full. Writes are handled in a no-steal/no-force method, that is buffer pages can not be written before they are unpinned, and are not forced to disk at commit. This implementation does not differ between dirty or non-dirty pages, nor between index and leaf pages, when choosing a page for purging. It also has no special handling of file scans, and as such perform badly when many or large files are scanned.

5.1.2 Cursors

For query processing cursors are the most important objects. The `Cursor` interface defines three methods: `next()` for moving to the next record and returning true or false indicating if any more records are available; `getKey()` for fetching the current key; and `getData()` for fetching the current data object. This simple interface must be implemented and be the result of any query, join or relational method. In addition any join and relational operator should take cursors as input parameters, so as to allow combinations of several operators.

As virtually any relational method must take as input zero or more cursors and result in a cursor object, several different cursor have been implemented. Most of these are used in special operations, and will be described in the appropriate section. A few general ones will be described here. Many of these have since implementation been removed from the project as they were superseded and no longer in use, but are still included here for descriptive purposes.

`SelectionCursor` is a relational operator discarding any object that does not pass the selection criteria given. The selection criteria is defined using a `Selection` class.

`RangeCursor` is a specialization of the selection cursor, implementing the SQL BETWEEN method. This discards any object outside the given

range. A single input cursor, as well as minimum and maximum boundaries must be given as input.

`UnionCursor` is a set operator taking one or more input cursor and returning all objects from all input cursors as a single cursor. This does not discard duplicate objects, and the data is returned in the same order as in the input cursors, and in the order of cursors. Similar to SQLs UNION ALL.

`IntersectCursor` is a set operator taking two input cursors and returning the key and data objects that are in both cursors. Both key and data must be equal, and the resulting order is undefined. Similar to SQLs INTERSECT.

`SwapCursor` swaps the key and data objects of the input cursor. However, it does not change the data type of these, and as such both objects should implement the `Key` *and* `Data` interfaces of NEUStore.

`NestedLoopJoinCursor` implements a standard nested loop equality join on two input cursors and returns the data object of the first cursor. As most cursors does not support rescanning, this implementation requires the entire inner cursor to fit in available memory.

`HashJoinCursor` implements a standard hash based equality join on two input cursors and returns the data object of the first cursor. Again the inner cursor must fit in available memory as hash buckets are not written to disk in this implementation.

`BufferedCursor` is used for implementing parallelization and will be described further in section 5.5

5.2 Access Structures

Several access structures were implemented or refined as a part of this project. All the access methods are extended to support `LaesaIndex`. As `LaesaIndex` only requires support for insertion and range queries, virtually any access method can be used. As our experiments will not contain updates or deletes, these were not implemented in all the access methods.

A specialized `LowerBoundCursor` is implemented for each index method to return the lower bound on the distance between the query object and the results, and not the distance between the pivot and the results, as a data object. In addition the object key is returned as the key, as this will be used later for joins.

5.2.1 Heap Files

Heap files are unsorted files where each record is append to the end of file or in the first unfilled page. The original heap file implementation was performed by *Donghui Zhang* and is a part of neustore, this was however modified during the project to be better suited for generalized usage. The heap file implementation is located in the package `neustore.heapfile` and the main class is `HeapFile`. This contains a linked list of full and non-full `HeapFilePage` pages, that again contain a list of `HeapFileRecord`.

When a record is inserted a new record is created and appended to the first non-full page. If no non-full page exists, a new page is allocated and added to the non-full list. If the page is now full it is moved to the list of full pages. This allows insertions in constant time. The only available direct access method is a full scan. Searches and range queries are implemented by performing a full scan followed by a selection operator. Deletions are similar to a search, followed by removing the record from the page if found and moving the page to the non-full list.

5.2.2 B+-tree

B+-tree is implemented as described in [27, ch. 10] but without support for deletion and updates. The implementation consists of a tree hierarchy of `BTreePage` nodes that can represent both index and leaf nodes. Both contain a `SortedRecordList` for maintaining a sorted list of records, with each record consisting of a key and either a page id for the next level or data objects for the leaf pages.

Most of this implementation was performed by the project supervisor *Svein Erik Bratsberg* as a part of another, earlier project. Some tweaks and bug fixes has been performed as part of this project by the author, mostly to allow the B+-tree to function properly within the general metric indexing framework. The implementation was also extended with the subclass `LaesaBTree` that implements `LaesaIndex` and allows for the usage of the B+-tree in LAESA indexing and querying.

Searches are performed by traversing the tree from root to leaf following the paths that contain the interval of the desired key. Binary searches are performed in each node. Range queries are implemented by a tree traversal for start key, followed by a scan of leaf nodes until the end key is found. As the record are sorted in each page, the results of range queries is sorted on the key in ascending order.

Insertions are performed by a search for the key, followed by an insertion and possible split of the leaf node that was traversed last in the search. Duplicate keys are allowed, as this is a prerequisite for distance indexing objects that can have the same distance from the pivot.

5.2.3 R*-tree

Several R-tree methods was investigated before choosing the R*-tree [2]. The main reason for this choice was because this is the most commonly used R-tree type used in database systems, due mostly to it's engineering approach to reducing overlap and thus increasing query performance [21]. The implementation is faithful to the description in [12] and [2] except for two key areas.

During experimentation we found that our usage of the R*-trees was mainly CPU bound, and thus disabled optimizations that attempt to reduce file accesses at the cost of increased CPU time. Beckmann [2] states that *minimum-overlap* has a minimal effect on the number of file reads, with the cost of increased cpu-time. Reinsertions were also disabled as these were found to have no effect on query processing, but a detrimental effect on file accesses and time during indexing, especially for high-dimensional trees.

The implementation is located in the `ntnustore.rtree` package, with the main class `RTree`. The main tree has a root node, with a links to page ids of all children. Leaf pages have a list of data objects instead of page ids. The `RTreePage` represents both leaf and index pages. Each page a `RTreeRecordList`, which is basically a list of `RTreeRecord`. Each record contains a key and a data object, however the key is not user defined and can only be a `RTreeRegion`. This limitation was added to ensure the algorithms used work properly, as there was a need for multidimensional data, and also support for several algorithms available for those data types. The `RTreeRegion` contains methods for area, margin, union, intersection and containment calculations on region objects.

Insertions are handled by first using the `chooseSubtree()` method to find the best leaf node to place the new object. The best node is defined as the node that needs the minimum overlap or area enlargement, depending on the node type. When a leaf node is found the object is inserted and the MBR of the leaf node and all parent nodes are adjusted accordingly. If the node is overfull after insertion it is split according to the best axis and grouping of objects within this axis. The new node is added to the parent, and further splits could occur propagating up the tree. If the root node is split a new root is created. Duplicate keys are allowed, and for this reason deletes and updates are not implemented.

Queries are implemented in `RTreeRangeQueryCursor` and performed using a depth-first traversal of the tree from the root node. A list of page ids for unvisited nodes are kept on a stack. For each node visited the MBR of each children is tested for intersection with the query region. If they intersect the child is either added to the result set, if this is a leaf node, or to the unvisited node list. The order of results is undefined and not related to either the key or data objects ordering. Due to the search method used the stack is limited to the number of records per index node times the height of

the tree, and with a page id of 4 byte it can easily be kept in memory. This method ensure that no page (except possibly the root) is fetched from disk unless their MBR intersects with the query region.

`LaesaIndex` is implemented in `RTreeLaesaIndex`, and differs from the heap file and B-tree index in an important way. Because the R-tree is multidimensional, and only the lower bound should be returned with the query cursor, this will need to calculate the greatest lower bound. This is done by calculating the lower bound for each pivot, between the query object and result object, and then calculating the greatest lower bound. This is equal to what is done during joining, and will be described later in this chapter.

5.3 Metric Indexing

While the bottom layer of the index is dependent on the access methods used, the higher level application logic is independent of access method, and can in fact simultaneously be used with different access methods for each index file. While the access methods ensure that it is possible to insert and query for each pivot (or set of pivots in the case of R-trees), the higher layer application code is needed to coordinate insertions in multiple index files, perform distance calculations, and distribute, manage and join query results.

This section will describe the concepts, classes and methods used during indexing, while the next section will concentrate on query processing. It should be noted that the various `LaesaIndex` implementations described in the previous section is also a part of the application logic. However, as these are dependent on the access method used, they have been described previously.

5.3.1 Distance

A general interface for distance calculations is given in `Distance`. All distance implementations must implement this interface, with either a generalized data object or specified data type. All implementations are located in the package `ntnustore.distance`. We have implemented several distance metrics specifically for this project, as well as a wrapper for the SimMetrics project [32] (albeit none of the SimMetrics distance measures are used in the experiment). For more information regarding various string metrics and the implementation of these see [31].

Two main methods are defined in the interface: `initialize(o1)` is used to initialize the distance calculation using the given (`o1`) data object (usually the pivot or query object); `distance(o2)` returns the distance between `o1` and `o2`, that is $d(o_1, o_2)$. In addition `getDistances()` is used

for experimental purposes, and should return the number of distance calculations performed or, if the cost is more complex, a similar value describing the cost of all distance operations performed since initilization.

For string based data sets we have implemented the Levenshtein distance, or edit distance, in the class `LevenshteinDistance`. In addition we have introduced a variant using words as the methods alphabet, for increased speed and reliability in the class `LevenshteinWordDistance`. For vector data a general Minkowski distance has been implemented in the class `LXDistance`, and in addition the Euclidean and Chebyshev distance have been implemented in `L2Distance` and `ChebyshevDistance`. All the distance algorithms are described in section 2.1.1.

5.3.2 LaesaIndex Interface

The `LaesaIndex` interface is implemented by all access methods that support LAESA indexing and querying. In practice this is done by a subclass, and not the actual index structure. The interface contains two relevant methods: `insert(key, data)` allows the insertion of new data objects; `query(query, range)` returns all data objects within the given range of the query object, as well as the lower bound distance between the query object and the data objects. The result set might contain false matches, and should only be treated as a candidate set until post-processed.

In addition the class `LaesaIndexList` implements the interface, and thus allows a list of index files to be grouped together and handled as one. This class is also used by applications and contains the high level interface and application logic for distributing inserts, handling queries and joining the results from the list of indexes. The detailed operation of this class will be described in the relevant sections below.

5.3.3 Pivot Selection

There are several methods to selecting pivots, as described in section 4.3. We have chosen to select pivots that are as far away from each other as possible, and have implemented this as three methods in `LaesaIndexList` all named `getPivotCandidate()` with varying number of parameters and purposes. The methods will return either a single key, or a list of keys, for the data object(s) chosen as the next pivot candidate.

The first method requires no existing pivots and is only useful for finding the first pivot. This takes a random object, in fact our implementation uses the first data object, and measures the distance to every other object in the data set. The object furthest away from the random object is the first pivot. This requires n distance calculations to be performed, where n is the number of data objects, as well as a single scan of the data set.

The second method finds the next pivot candidate using the current set of pivots and index files. This is done by first creating a candidate set containing all objects, and then scanning all the index files from $d = 0.0$. Every object encountered is then removed from the candidate set until there are no more candidates, then the last candidate or the candidate is the new pivot. This method requires no data objects to be fetched, and no distance calculations to be performed. However, it does require all pivots to be scanned and has a worst case complexity of $O(n * p)$ where p is the number of pivots. Experiments have shown that the average complexity is only $O(\max(n, k * p))$, where k is constant albeit fairly large. This is because the increased pivot count has extra cost for initiating the scan, but the candidate set is more quickly reduced to near zero.

The third method is a combination of the two above, but is performed solely in memory and does not require any stored index files. It also returns a list of p pivot keys, and not just a single pivot. This is useful for finding a list of pivots that can later be used, and is for instance used before performing our experiments. The initial pivot p_1 is first found using the first method. The distance from pivot p_1 to all other objects is then calculated and stored in an array, the object furthest from p_1 is now picked as p_2. The distance from p_2 to all objects is calculated, and the array updated with the minimum distance to any pivot. The array now contains for any object o_i, $d_i = min(d(p_1, o_i), d(p_2, o_i))$, and the next pivot is selected with the maximum distance value in the array. This continues until p pivots are found. This method requires $n * p$ distance calculations and has a runtime complexity of $O(n * p)$.

5.3.4 Indexing

Indexing contains two main parts that can either be done sequentially or interwoven. First one or more pivots must be selected, a distance object initialized with each pivot must be created, and an appropriate index file created. The index file must implement `LaesaIndex`, and usually take the distance object(s) as parameter. Second all data objects must be inserted into the index files.

Insertions are performed by each implementation, and varies slightly with the type of access method used. For the single dimensional access methods the distance between the pivot and the data object is calculated, and the distance and object key is inserted into the index file. For multi-dimensional access methods, the distance between the data object and all pivots are calculated, and a record containing the list of distances is inserted with the data key. `LaesaIndexList` implements insertions by delegating the insert to all the child indexes sequentially.

5.4 Similarity Search

Similarity search is performed in two main phases, first the filtering step will remove all objects that are known not to be within the range limit. This results in a candidate set, together with lower bound distances from the candidate set to the query object. The candidate set must then be post-processed to give the final result set. The post processing used depends on the query type, while the filtering step is always equal.

5.4.1 Filtering

Filtering is performed as described in section 4.4 by the `LaesaIndex query()` method. This takes a single query object, as well as the range limit for excluding data objects. The results of the query on each index file is a cursor containing a set of tuples $(i, \check{d})$ where i is the object key for object o_i and $\check{d}$ the lower bound distance as seen from this pivot, or set of pivots. All implementations created with the same pivot must return an equal result set, both with regard to keys and lower bounds, albeit with different orderings allowed.

The basic index types as described earlier perform the query using either a range query or a file scan followed by a range selection. For multidimensional methods this range query is somewhat more complex, but conceptually equal. The `LaesaIndexList` allows the query to be performed using several index files. The query is performed on all child indexes, and the result set of all the queries is joined using a standard database join method. During the join only the highest lower bound is kept for each object key.

During implementation and experiments several join methods have been tested. Nested loop join was effective when the result sets was small, but quickly caused problems with increasing result set size, while hash based joins managed to grow without decreased performance. We also tested several join hierarchies, mainly variations of left-deep and bushy join trees. The conclusion was that pure left-deep trees offer a substantial advantage due to the selectivity of the equality join causing the inner (left) cursor to decrease rapidly in size.

The final join method was implemented in `MaxFloatHashJoin` as specialized a hash based equality join on two input cursors. As only the highest of the lower bounds should be returned, the data objects of both cursors must be a `Float` object, and the larger of the two data objects are returned together with the object key.

5.4.2 Range Query

Range queries are performed as a post-processing step after filtering is complete, however it is often performed pipelined and including or discarding objects as they are returned from the filtering process. The `DistanceCursor` takes a input cursor as well as a query object q and range limit r. The input cursor can either be the result of a filtering step, or a full scan of the data file.

All data objects are fetched using the object key, and the exact distance $d_i = d(q, o_i)$ between the query object and the data object calculated. If $d_i \leq r$ then the object is included in the result set, together with the exact distance, otherwise it is discarded.

5.4.3 Nearest Neighbour Query

Nearest neighbour queries are performed using the `KnnCursor`. This takes as input a cursor returned from the filtering process with lower bounds, a query object q, the number of nearest neighbours k and the range limit r. It returns as output a cursor with an ordered set of the k nearest objects within the range limit r.

All objects are read from the input cursor before further processing is performed, and pipelining is therefore not possible. The candidates are then sorted by lower bound cursor and put in an input queue. Objects are removed from the head of the queue, and the exact distance between the query object and data object calculated. If the object is within the distance to the Kth nearest object found until now, it is added to the sorted list of nearest objects and the K+1 object removed from the list. This continues until the next candidate has a lower bound higher then the Kth nearest object, at this point the entire candidate queue is discarded and the result set can be returned.

It is possible to return parts of the result set as soon as it is known that no object on the candidate queue can be nearer the query object. This is done when the first object on the result list has a distance below the lower bound of the next object on the candidate queue. Thus allowing incremental nearest neighbour queries to be performed.

5.5 Parallelization

Modern database systems often rely heavily on parallelization to increase the throughput of operations, especially related to handling large amounts of data. Two methods that can be used to perform this is by either partitioning the data itself, or by pipelining the relational operators that handle the data. We have primarily relied on data portioning due to the join methods

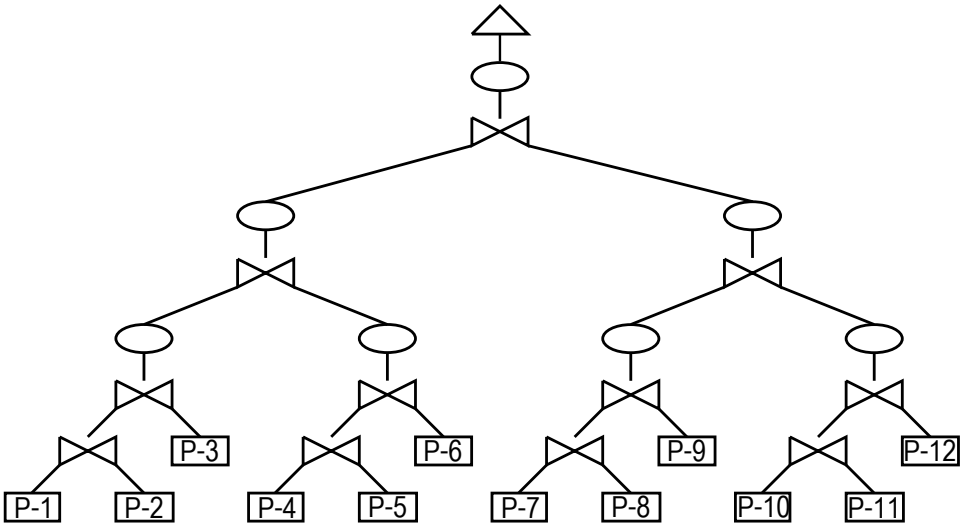


Figure 5.1: Parallel query join tree example using 12 pivots and 4 execution threads. The pivots (squares marked with P-K for pivot K) are divided in equal size groups of pivots according to the number of threads. Each pivot is joined by a hash join, and each group is read by a BufferedCursor running in each thread. The join groups are then joined in a bushy join tree, and finally written out as the result/candidate set.

used, while pipelining would be more important in a distributed environment.

For indexing each index file is handled separately. For single-pivot index files this equals one job per pivot, and for multi-index files (namely R-trees) multiple pivots are contained inside one index file. A job is created for each pivot reading in the entire data set one object at a time, calculating the distance to the pivot, and adding the distance and object key to the index structure. A parallel executor is instantiated with P threads and given the job queue. The number of threads depending on the level of parallelization wanted. As all pivots are independent all the indexing processes will also be independent and the job queue will be emptied in a FIFO fashion.

As the filtering method inherently consists of several range scan operations combined by hash joins, it is not well suited for pipelining. This is because the hash joins require the entire inner relation to be fully read and hashed before it can proceed with the outer relation. Using a different join algorithm would alleviate this issue, however that would take increased cost to perform the join itself, and would give little benefit compared to partitioning the pivots.

By partitioning the pivots into as many groups as there are executable threads, each group of pivot scans can be joined together independently of the other groups. And only when one or more of the groups are done will

the results of these be joined together using a bushy join tree. As the result set from the filtering is reduced drastically when combining several pivots, this will result in most of the computation being performed in the main join groups in parallel, and only minor parts of the data set being joined in the final join of these groups. See Figure 5.1 for a visualization of this process.

This was implemented mainly using a new cursor called the `BufferedCursor`, which both contains a buffer of data objects as well as being `Runnable` itself. One `BufferedCursor` is used at the top of each join group, thus reading in the entire result set from each of these join groups in parallel. When the first group finishes execution, the data is read as soon as it is available from the buffer of both this and the next cursor, even if this has not finished execution yet.

5.6 Testing and Experimentation

During development several test classes were used, specifically several JUnit [1, 14] test suites. These are located in the `test/` folder and contain the same package and class name as the class under test. These tests were used to ensure that the implementation was working correctly, especially when modifying parameters or testing different implementations.

The experiments were performed using three classes located in the `test` package. The class `TestRange` contains methods related to testing the effect of different range limits, independent of access method used. This class does not use the standard indexing and query method, but instead relies on a large available main memory to perform range tests. Specifically the method `printDistanceDistribution()` is used to create a range distribution histogram including information about result set size. For the performance cost experiments the class `TestDistanceCost` is used in a similar fashion.

The remaining experiments related to pivot count, indexing, range and nearest neighbour queries were performed using the `TestPivotCount` class. This contains methods for building an index, performing queries, and printing statistics based on data, pivot and query sets located in a data file. It also provides most of the configuration settings that are detailed in the experimentation section. As such this also functions a sample application of how to use the indexing and query implementation described earlier.

Chapter 6

Data Sets and Metrics

We have described a general system that can be applied to any data set providing it has a meaningful metric distance function. However, the properties and functioning of the system, especially related to its performance characteristics, are highly dependent on the characteristics of the data set and distance function.

For this reason we will show detailed information about the data sets used during the rest of the experiments. Specifically we will show the data distribution, intrinsic dimensionality, expected result set size. These properties are solely dependent on the data set and metric used, and not related to the methods or system developed during this report (however, we have used the system to calculate parts of the data. In addition we will show the response times of various metrics, something which is dependent on both the data set, metric function complexity, implementation and system properties.

6.1 Data Sets

For the experiments we will use three distinct data sets with different properties, two vector sets and one document collection. The vector data sets, *nasa* and *colors*, are both a part of the Metric Space Library [17] from the International Workshop on Similarity Search and Applications (SISAP). nasa is *a set of 40,150 20-dimensional feature vectors, generated from images downloaded from NASA and with duplicate vectors eliminated*, and colors *a set of 112,682 color histograms (112-dimensional vectors) from an image database*. These two vector sets should represent real world data better than synthetic data sets, especially uniformly generated data sets.

The document collection is a subset of the GOV2 Test Collection [34] from the Text REtrieval Conference (TREC) Terabyte Track. This contains a large proportion of the available pages under the .gov top level domain in early 2004. The subset used are the folders GX000, GX001 and GX002.

All documents are stripped of all html-related tags and whitespace, and truncating to a maximum of 2048 characters. All collections are shuffled to reduce unnatural clustering effects due to the method of data collection.

For the two vector sets we will use Minkowski distance [26] metrics, including Manhattan distance (L_1) [18], Euclidean distance (L_2) and Chebyshev distance (L_∞). For the TREC document collection we will use two variations of Levenshtein distance with either a single character or single word alphabet. Summary information about the data sets are given in Figure 6.1.

Name	Type	Count	Object size	Total size
nasa	vector (20 dim.)	40,150	80 b	3.06 MB
colors	vector (112 dim.)	112,682	448 b	48.1 MB
TREC	documents	25,205,179	17.7 kb	426 GB
TREC (subset)	documents	272,390	1.50 kb	399 MB

Figure 6.1: Summary information of the data sets nasa, colors and TREC

6.2 Data Distribution

To calculate the histograms 100,000 object pairs (o_1, o_2) is randomly sampled for each database and metric, using two independent random variables for each of o_1 and o_2. The distance $d(o_1, o_2)$ is calculated and recorded for each pair. A relative distribution histogram is then constructed using the stored data, with the mean centered for all single distribution histograms to allow for easier direct comparison.

The distance distribution for the nasa data set is shown in figure 6.2 for each metric separately, and in 6.3 they are combined in a single graph. From the histograms we can clearly see that while they all resemble a Gaussian distribution, the L_5 and L_∞ distributions are skewed heavily towards the lower range. This causes serious problems for filtering applications, as it relies heavily on discarding elements that are above the range limit, and much less on those that are below the limit. In the combined histogram, it is also clear that L_5 and L_∞ are nearly identical.

The distance distribution for the colors data set is shown in figure 6.4 for each metric separately, and in 6.5 they are combined in a single graph. We here see much the same results as with the nasa data, except for a much thicker right tail.

The distance distribution for the colors TREC set is shown in figure 6.6 for each metric separately, and in 6.7 they are combined in a single graph. This shows quite a different story from the nasa and colors data. While there are still resemblances of a Gaussian distribution, the data is far more clustered with just a small range holding virtually all the data. This will

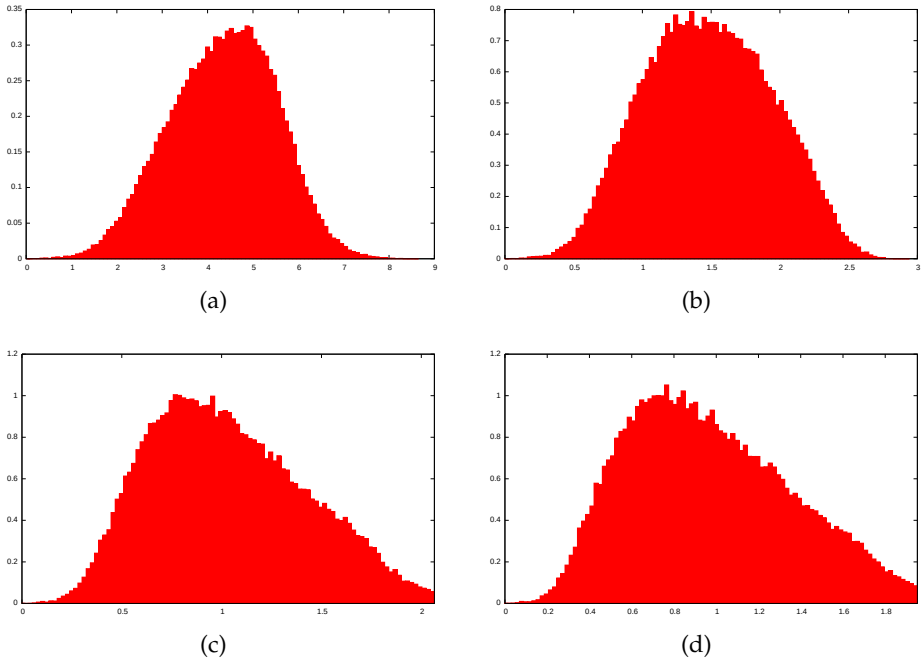


Figure 6.2: Data distribution for nasa using various Minkowski distance metrics: (a) Manhattan distance (L_1); (b) Euclidean distance (L_2); (c) L_5; (d) Chebyshev distance (L_∞).

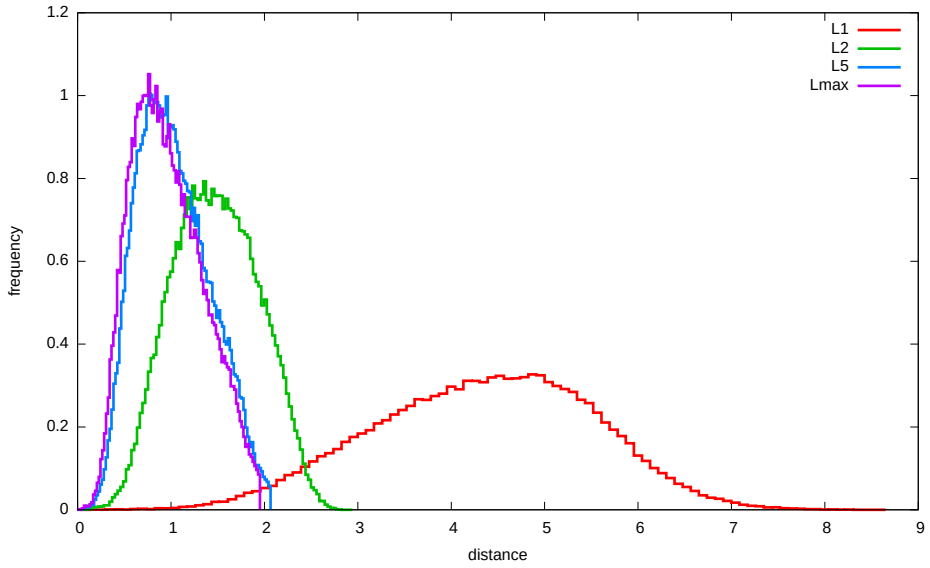


Figure 6.3: Combined non-normalized data distribution chart for the nasa database with all metrics

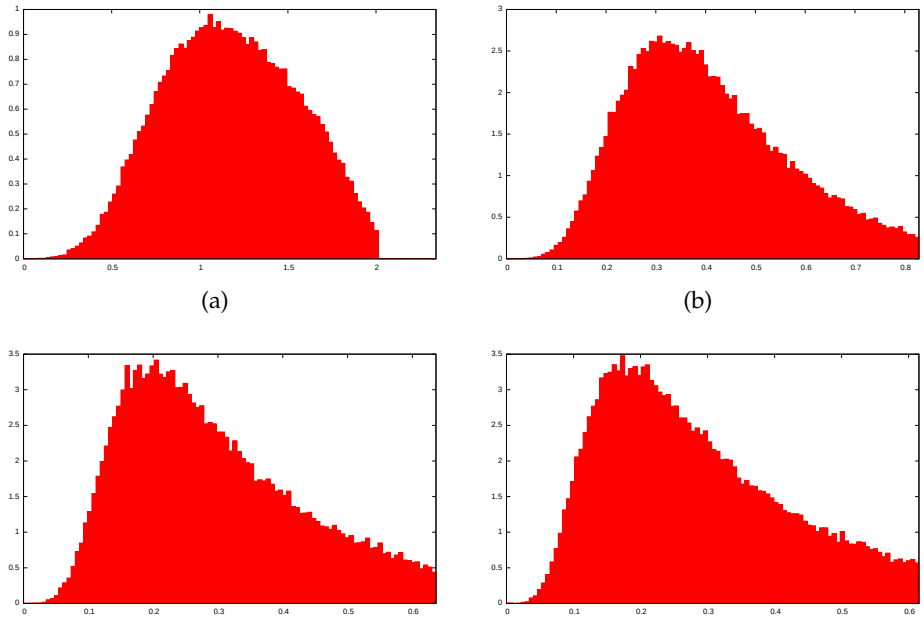


Figure 6.4: Data distribution for colors using various Minkowski distance metrics: (a) Manhattan distance (L_1); (b) Euclidean distance (L_2); (c) L_5; (d) Chebyshev distance (L_∞).

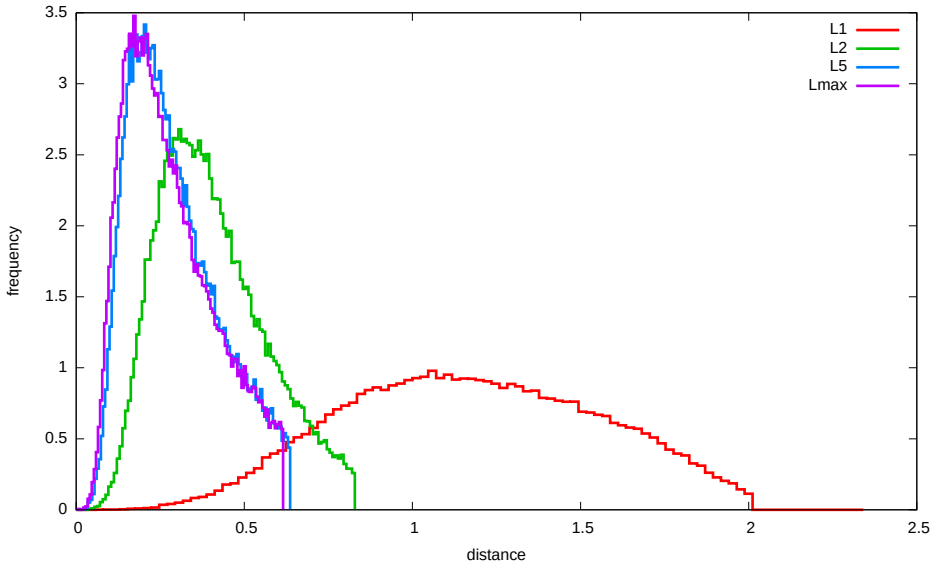
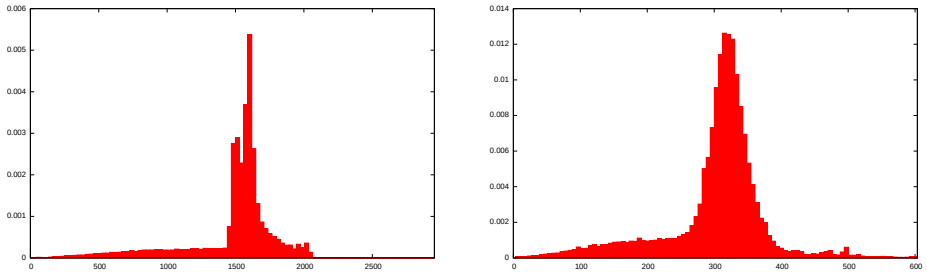


Figure 6.5: Combined non-normalized data distribution chart for the colors database with all metrics



(a) (b)

Figure 6.6: Data distribution for TREC GOV2 (subset) using two Levenshtein distance metrics with: (a) a single-character alphabet; (b) a single-word alphabet.

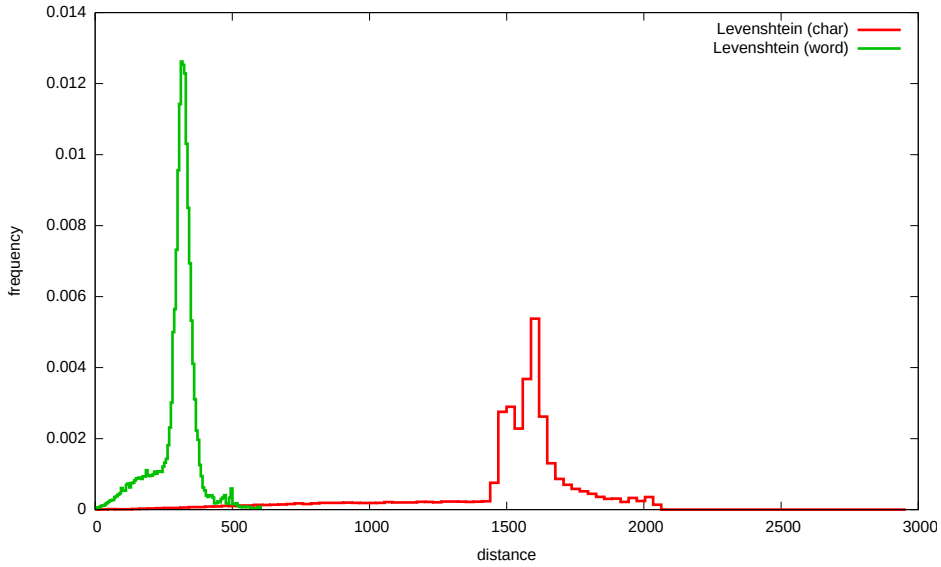


Figure 6.7: Combined non-normalized data distribution chart for the TREC GOV2 subset with both metrics

make it far harder to perform filtering on this type of data. It is also a sign that Levenshtein could be a bad metric of document similarity, as most documents are rated similarity when compared to each other.

6.3 Dimensionality

The dimensionality of a data set largely defines the hardness of the indexing and filtering operation. The more dimensions a data set contains, the more likely it is to cluster towards the center, with a lower variance in average distance. Thus more objects are returned within the same range interval. This also influences the number of false positives returned in the candidate set, and thus the number of pivots that are needed to filter out enough candidates. This is often referred to as the *curse of dimensionality*.

The dimensionality of a vector data set is given by the number of coordinate points. However, often there is a strong correlation between two or more of the coordinates. In traditional spatial indexing it is not possible to eliminate these correlated coordinates from the index, and such a lot of space and time is wasted on storing unneeded information. With pivot filtering correlated dimensions are automatically merged as they will give the same results (or some kind of mathematical relationship) in a distance computation.

When removing these correlated dimensions we are left with the data sets intrinsic dimensionality. The intrinsic dimensionality can be calculated using the data distribution histogram from the previous chapter combined with the formula $p = \frac{\mu^2}{2\sigma^2}$ [6]. This uses both the mean and variance of a histogram to calculate its intrinsic dimensionality. The *flatter* or more spread out a digram is, the lower the dimensionality, while a more centered distribution gives a higher dimensionality.

Metric	Mean	Variance	Dimensionality
Manhattan (L_1)	4.347	1.360	6.946
Euclidean (L_2)	1.478	0.210	5.202
L_5	1.038	0.159	3.394
Chebyshev (L_∞)	0.981	0.163	2.956

Figure 6.8: Statistics and dimensionality for *nasa* data set using various metrics

In Figures 6.8 and 6.9 we show the mean, variance and dimensionality for the two vector data sets and all metrics used. As we can see the dimensionality is hugest for the L_1 metric and steadily reducing for the higher L-metrics. This is also evident in the distribution graphs, with L_1 having a much more centered around the mean distribution.

Metric	Mean	Variance	Dimensionality
Manhattan (L_1)	1.1763	0.1412	4.901
Euclidean (L_2)	0.4164	0.0313	2.773
L_5	0.3198	0.0293	1.744
Chebyshev (L_∞)	0.3093	0.0301	1.588

Figure 6.9: Statistics and dimensionality for *colors* data set using various metrics

Metric	Mean	Variance	Dimensionality
Levenshtein (char)	1484	106166	10.38
Levenshtein (word)	303.2	5875	7.825

Figure 6.10: Statistics and dimensionality for *TREC GOV2* data set with both metrics

Similarly from the Figure 6.10 for the TREC data set, we can see that using the word alphabet results in a lower dimensionality, and this metric is thus more suitable for filtering than its character-based variant. We can also see that the dimensionality (and thus hardness) for the TREC data set using Levenshtein is much higher for both metrics than using any of the Minkowski distances.

6.4 Result Set Size and Range Limit

The most important factor for an effective filtering technique to work is the range limit, especially so for our disk based method. The range limit defines the tightness of the lower bound, and as such the pruning power of each pivot. This is very important as all our performance factors are related to the pruning power. A high pruning power results in fewer file pages that must be read from each index, less CPU time spent joining candidate sets, and most importantly fewer false positive matches (candidates) that must be discarded after a costly exact distance calculation. Contrary, a low pruning power has the opposite effect on all of these performance factors.

The average expected result set size for a given range limit can be analysed using the distance distribution diagrams from the previous sections. This results in a cummulative distribution graph. We have shown only the lower range limits, as these are the most interesting in relation to range limits. From the figures we clearly see that the range limits used are highly dependent on the metric.

Common for all metrics is an exponential growth in result set size for the low range. Thus it is important to use the lowest range limit possible to get sufficient pruning power. Even a difference of $1 - 5\%$ in the range limit

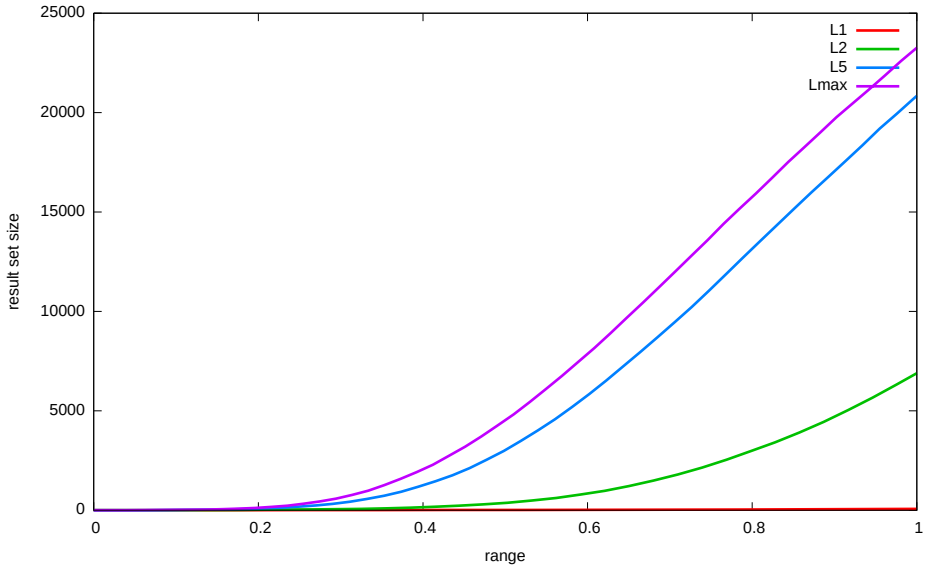


Figure 6.11: Expected result set size for the *nasa* data set using all applicable metrics for range limits $r \leq 1.0$

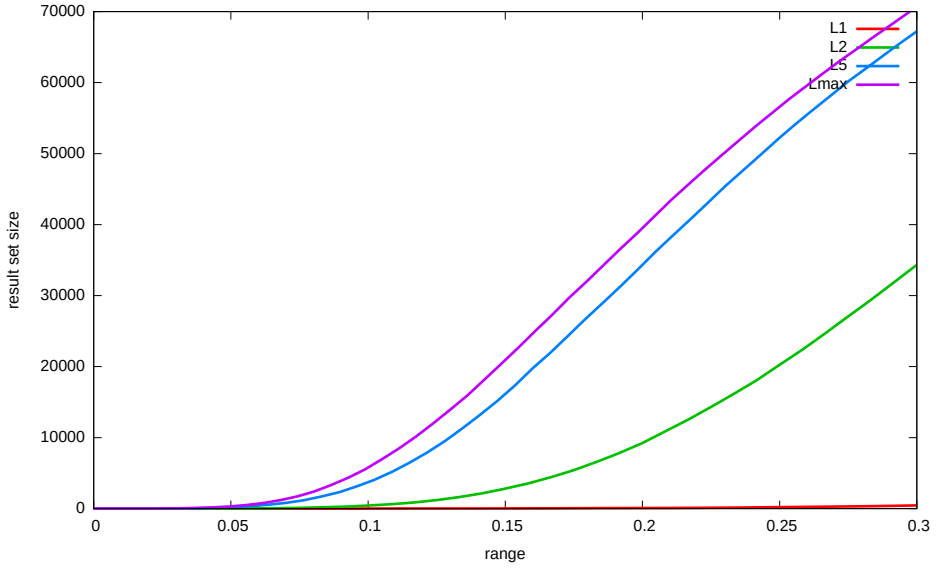


Figure 6.12: Expected result set size for the *colors* data set using all applicable metrics for range limits $r \leq 0.3$

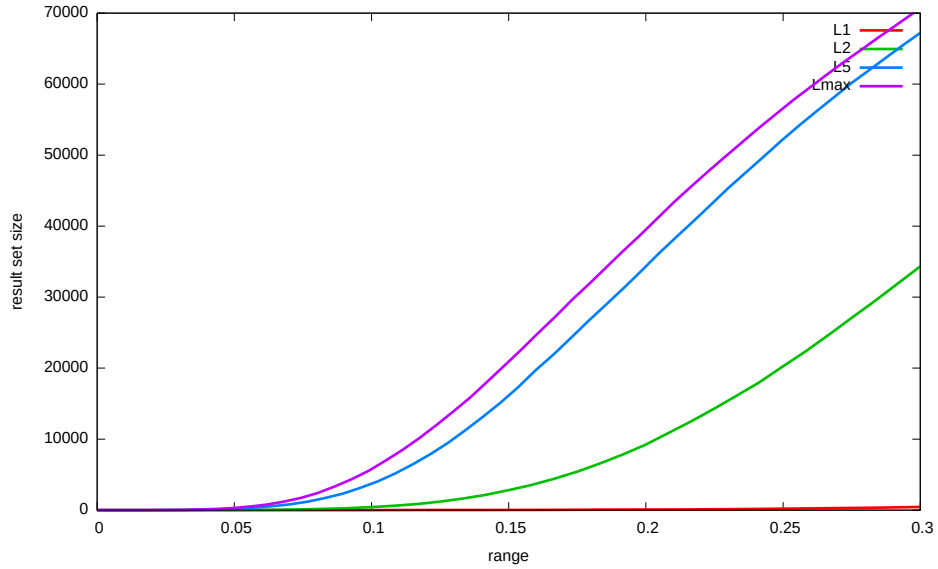


Figure 6.13: Expected result set size for the *TREC GOV2 (subset)* data set using all applicable metrics for range limits $r \leq 100$

r	$N(R)$	$N(R)/N(D)$
0.2	17.8	0.04 %
0.4	135.7	0.33 %
0.6	771.1	1.92 %

Figure 6.14: Range limits with average result set size and fraction of total data set size, using the nasa data set and L_2 metric

can have a severe impact on the number of results returned. To ensure our experiments are representative for different query requirements, we will perform all filtering and query experiments using several range limit values.

As we will concentrate especially on the *nasa* data set and the L_2 metric (Euclidean distance) we have shown the average result set size and fraction of total data in Figure 6.14. These are also the three range limits we will concentrate on during our experiments, allowing us to show how widely different range limits will have an impact on the experiments.

6.5 Metric Performance

Another factor to consider when choosing a metric is its computational performance, that is how long it takes to perform a distance computation. For some applications several metrics will give equivalent distance measures

and thus equivalent recall and precision, then the computational time will come into play. A faster algorithm will allow more distance calculations to be performed, and require fewer pivots (and thus lower memory and disk cost) to achieve a similar response time.

Figure 6.15 shows the average response time of performing a single distance calculation between two objects. For each of our three data sets 1,000,000 object pairs were picked at random, and the distance between each was calculated. The calculations are of course dependent on the test machine used (see section sec:system-configuration for details), so we will only compare the relative response times here and ignore the absolute response times.

Metric	nasa	colors	TREC
Manhattan (L_1)	2.93 μs	7.42 μs	-
Euclidean (L_2)	1.92 μs	2.59 μs	-
L_5	10.4 μs	28.4 μs	-
Chebyshev (L_∞)	2.17 μs	3.84 μs	-
Levenshtein (char)	-	-	12900 μs
Levenshtein (word)	-	-	557 μs

Figure 6.15: Average response time for a single distance calculation between two random objects in a given data set for all applicable metrics

The two quickest metrics are the L_2 and L_∞ metrics, which is not surprising considering their both fairly straight-forward algorithms and are implemented using a specialized class. The following metrics, L_1 and L_5, are both implemented using the same parameterized class, and shows how performing more complex exponentiation and nth root calculation has a much higher cost. We can also clearly see that the colors 112-dimensions takes much longer to compute than the 20-dimensional nasa data, however it is by no means a 5-fold increase as one would expect with five times as much data.

More importantly we can see that compared to the L_2 metric, Levenshtein distance is 5 orders of a magnitude slower when using the most common character based alphabet. This shows the high complexity of comparing documents, even at only 2048 characters long, compared to easier data like vectors. For images or video data, this complexity is of course even higher, with comparison times measured in seconds and minutes instead of micro- and milliseconds.

When using the word-based alphabet this drops to only 3 orders of a magnitude slower. This result is not surprising when looking at the complexity function of Levenshtein, $O(n_1, n_2)$, where n_1 and n_2 are the lengths of either string. Thus it has a large advantage when reducing the length (in number of alphabet characters) at the cost of increased alphabet size.

Considering the enormous speed advantage of the word-based approach, it would be hard to ignore it as an option for a metric even if it had a lower recall or precision (it is also quite likely that word-based Levenshtein is better in these aspects as well).

While we have now looked at pairwise distance computations, our methods have just a few objects (pivots or query objects) that are used in all computations. This could give advantages to algorithms that benefit from cached results, locality of reference or has a high initialization cost for each source object used for comparison. We have therefor also performed a measurement with 10,000 random objects compared against 100 random objects each. That causes each object to be *re-used* in a distance calculation 100 times with only a single initialization.

Metric	nasa	colors	TREC
Manhattan (L_1)	1.39 μs	6.01 μs	-
Euclidean (L_2)	0.39 μs	0.93 μs	-
L_5	8.89 μs	26.5 μs	-
Chebyshev (L_∞)	0.68 μs	2.12 μs	-
Levenshtein (char)	-	-	12800 μs
Levenshtein (word)	-	-	554 μs

Figure 6.16: Average response time per single distance calculation between a random source object and 100 random target objects.

As can be seen in Figure 6.16 the effect on the various Minkowski algorithms was greatest, especially were the original response time was lowest. While there is little difference for the metrics with higher response time, with the Levenshtein metrics seeing virtually no difference. This result is easily explained by the fairly constant absolute initialization cost for all the metrics. This leads to a much higher relative initialization cost for the faster metrics compared to their low computation cost.

6.6 Discussion

As we have seen the intrinsic dimensionality, and thus the complexity of indexing and querying, of a data set is both related to the data set itself, as well as the metric used for comparison. From this we could assume that choosing the metric with the lowest dimensionality would be optimal, and for a pure time-performance measure that would be true. However, in real world applications the comparison performance of the metric compared to what a human would consider similar objects is often more important than memory usage and response time. For metrics that give similar recall and precision performance, one should however also consider the dimensionality as well as the response time performance of similar algorithms.

The dimensionality is also an important factor when considering the number of pivots that are needed to give an adequate filtering power, and should be used together with the distribution and other statistics when the optimizer determines the number of pivots used when answering a query. We will perform, and show the results of, more detailed experiments related to the number of pivots in the next chapter.

Chapter 7

Indexing and Query Processing

In order to verify the performance of the different indexing methods proposed and implemented, we have run several experiments. First an overview of the system configuration will be presented, followed by a few initial experiments to determine some final parameters for the experiments.

For all the experiments performed, the *nasa vector database* from [17] is used as the experimental data set. This is *a set of 40,150 20-dimensional feature vectors, generated from images downloaded from NASA and with duplicate vectors eliminated*. And should represent real world data better than generated data sets, especially uniform data sets. We use Euclidean distance (L_2) for distance measurements.

50 randomly selected objects from the data set compromise the query set. All query experiments are performed by issuing 50 independent queries from this set and averaging the measured results.

7.1 System Configuration

The specifications of the system used for experiments is given in Figure 7.1. In addition the implementation described in the previous chapter has a series of configuration parameters for controlling the indexing method used. These parameters are described below, in addition to their setting during the experiments.

CPU	$2 *$ AMD Opteron 2218 2.6GHz Dual Core
Memory	8.0 GB main memory
Disk	$2 *$ 146GB 10K SAS 1.5 Gbps, SmartArray E200i RAID-1
Operating system	Microsoft Windows 2003 Enterprise Edition SP2
Virtual machine	SUN Java SE 1.6.0_10

Figure 7.1: Experimental system specification

Page size the size (in bytes) of each buffer page. Set to 8192.

Buffer size sets the number of pages the buffer will have room for. Set to 10000.

Index type allows different indexing methods to be used for each test. For the tests where the indexing method is relevant, one of the following values will be used.

Heap file for using the heap file indexing method.

BTree for using the B-tree indexing method.

RTree d for using the R-tree indexing method, with a maximum dimensionality of d. Values used for d will be 16, 32, 64 and 128.

Java Virtual machine experiments are run with `-Xms1536m -Xmx1536m` to ensure enough memory is available and the garbage collector will not interfere with the performance measurements.

7.2 Number of Pivots

Different studies have come to different conclusions regarding the number of pivots that are optimal with a given data set. While some favor that only the intrinsic dimensionality is a factor [11], others have found that increasing the number of pivots with growing data size is preferable [6]. As all the experiments in this report are performing using a fixed data set, the volume and dimensionality is also fixed, and all that is needed is to find the optimal pivot count for the data set given various queries.

The number of pivots give a trade-off between filtering and post processing. While adding more pivots will indeed reduce the number of candidates returned from filtering, it also increases the cost of both indexing and filtering. Three factors are important when deciding on the number of pivots. The number of distance calculations that must be performed, the number of disk accesses, and the total response time. Disk accesses are needed both during filtering for accessing the index files, and during post processing for fetching candidates, but depends greatly on the indexing method and query type and will be discussed later.

The number of candidates returned from the filtering process, with varying number of pivots, are shown in Figure 7.2. The number of candidates is reduced drastically by the first pivots, however the effect for each added pivot diminishes until there is virtually no advantage of more pivots.

Distance calculations are performed both during filtering and post processing, and are independent of the index structure used. During filtering the number of distance calculations are equal to the number of pivots, while during post processing they depend on the query type. Range queries will

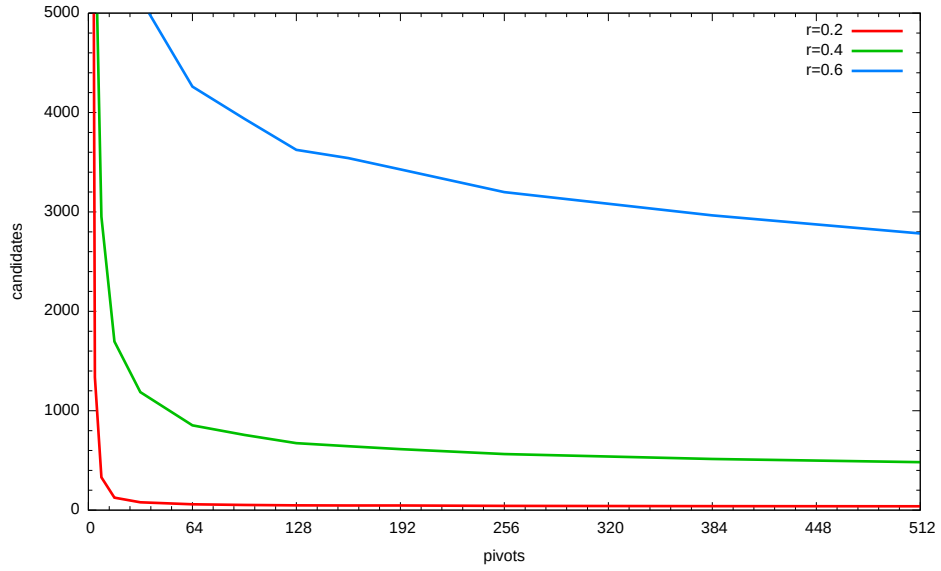


Figure 7.2: Size of candidate set returned by filtering process with varying number of pivots

need to calculate the distance of all candidates, while KNN queries usually only need to perform distance calculations until the remaining candidate set can be discarded by using the lower bounds.

Figure 7.3 shows the total number of distance calculations that must be performed for range queries, including those performed during filtering. We see that for all the range limits the distance count falls sharply at first, coinciding with the known drop in candidates. After the initial drop they remain fairly steady until finally rising slowly as the number of candidates remain equal. It is clear that increasing the number of pivots above about 256 give very little, if any, further benefit. We can also conclude that the number of pivots used for filtering depends heavily on the range limit, as a small range limit requires far less pivots for eliminating a majority of false candidates.

For 10-NN queries we see a similar trend in Figure 7.4, albeit more prominent due to the AESA algorithm causing far fewer distance calculations for KNN queries. Note especially the scale difference between the two query types. The minimum distance calculations needed for a range query at $r = 0.6$ is about 3300 against only 400 for 10-NN. And this also requires far fewer pivots, 128 versus 512, in fact the filtering alone took more distance calculations during query.

Once again the optimal number of pivots depend on the range limit, with lowest values at 32 pivots for $r = 0.2$, and 128 pivots for both $r = 0.4$ and $r = 0.6$. As such both the query type, and the range limit must be taken

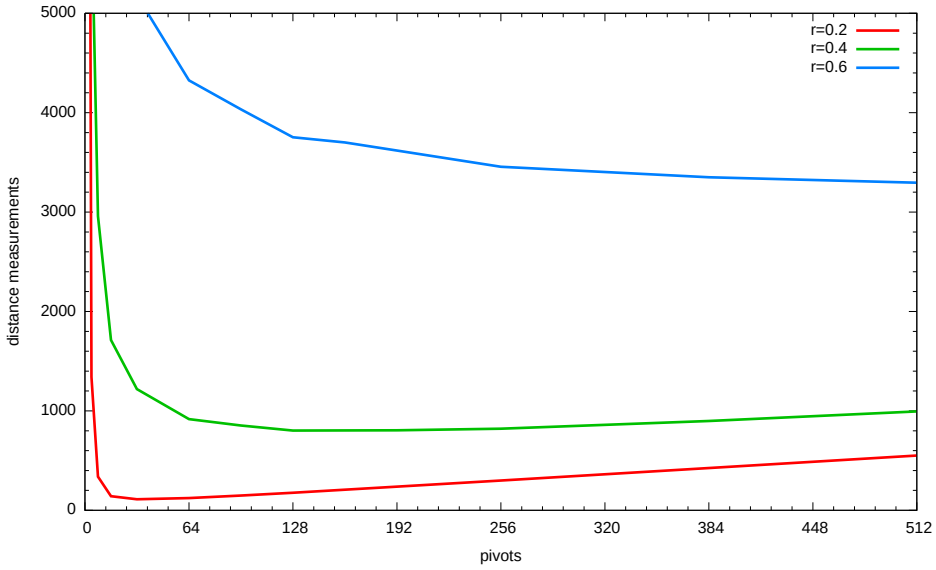


Figure 7.3: Distance calculations for range query

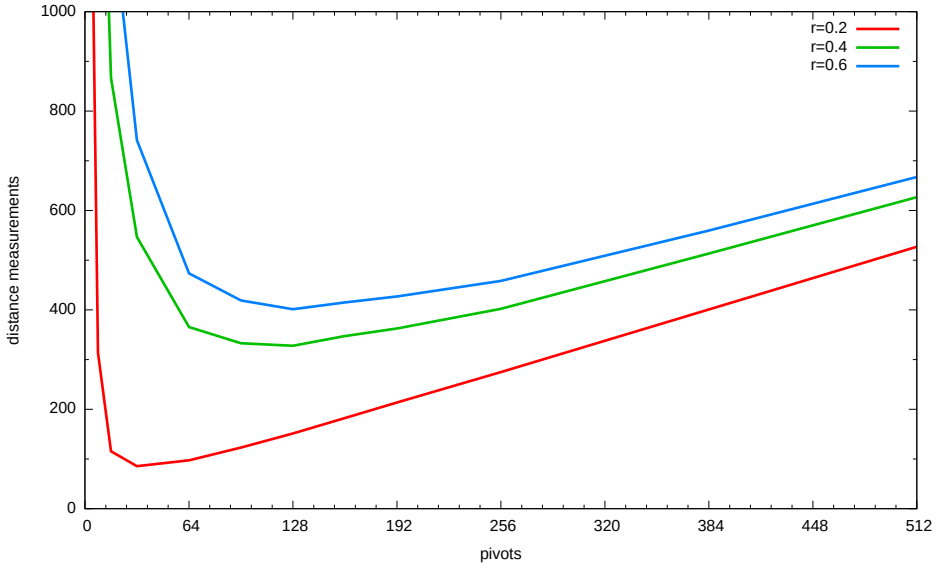


Figure 7.4: Distance calculations for 10 nearest neighbours query

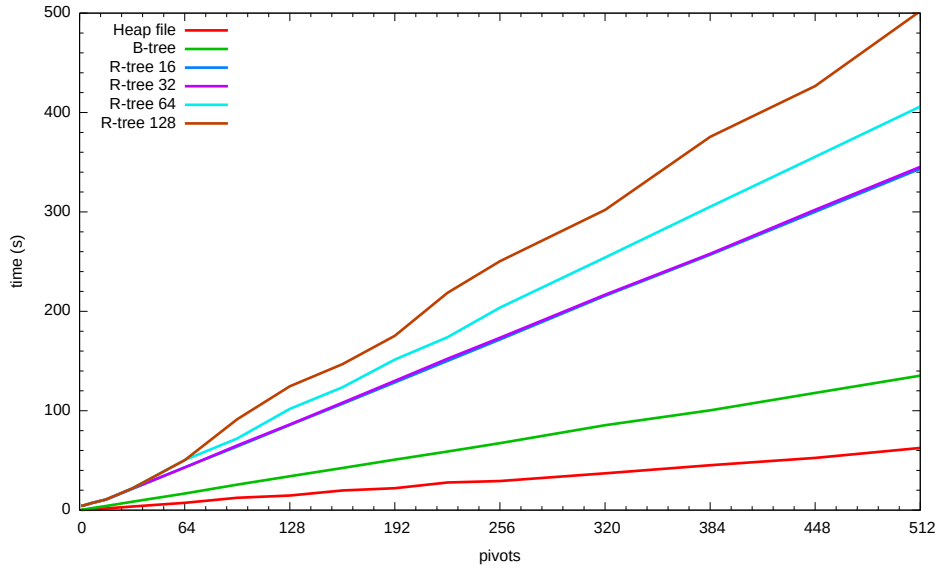


Figure 7.5: Time spent indexing for different indexing methods.

into consideration when selecting the number of pivots. However, what is clear is that adding more pivots is only useful up until a limit, when further pivots have a negative effect. For this reason we have concentrated on the results of experiments at 512 pivots and below.

7.3 Indexing

Indexing is performed by adding one index file at a time, scanning the datafile and inserting the distances into the index. For single pivot indexing methods (Heap file and B-tree) this is done once for each pivot, while for multidimensional indexing methods (R-tree) this is done once for each group of pivots. With a larger data set, adding several pivots at once would give an advantage as the data set would only be read from disk once. However, as the data set used in the experiment is much smaller than the available buffer space, there is no advantage to this method.

As indexing is independent of the query type it easy to compare the different indexing methods in Figure 7.5. Heap files are clearly fastest, with B-trees taking about twice as long. R-trees take a considerably longer time, and seem to be dependent on the dimensionality of the R-trees. Specifically the 16 and 32 dimensional R-trees are about equal, while the 64 and 128 are slower. It is not surprising that the simpler the index structure, the faster indexing is performed.

Looking at the number of file pages written in Figure 7.6 the reason for B-trees taking longer is obvious, in addition to the extra work of performing

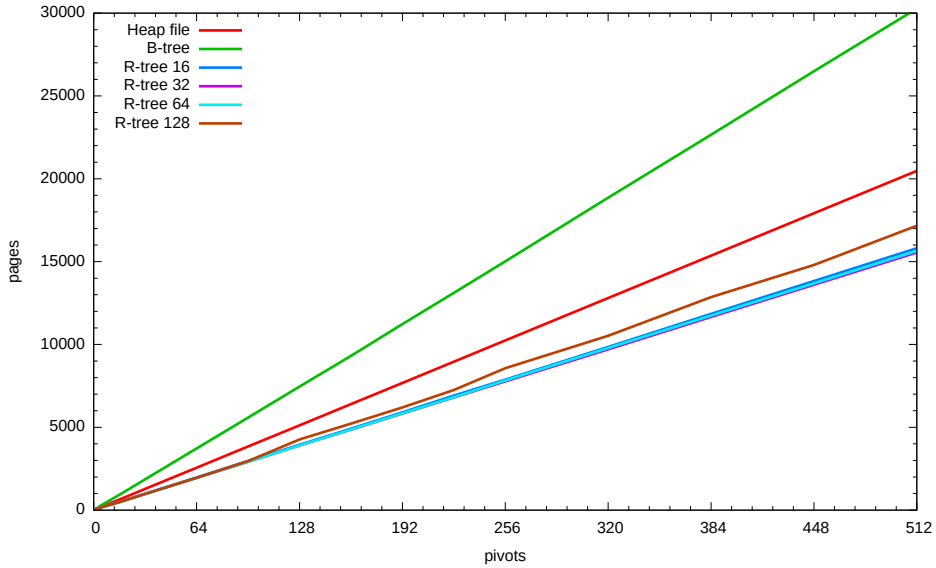


Figure 7.6: File pages written during indexing for different indexing methods.

comparisons it also has to write many more pages to disk. The reason for this is both the additional index nodes and underfilled data nodes. As we know the heap files are I/O bound, it is safe to assume that B-trees are also mainly I/O bound due to the high number of disk pages.

For R-trees however, the picture is a bit less clear, writing fewer pages then both heap files and B-trees. The dimensionality is less important here, with only the 128 dimensional sticking out with a few more pages, mostly because of lower fill-degree due to fewer records per node. This is in fact the key to why the 64 and 128 dimensional R-trees are so slow. As the number of records per node is bounded by the dimensionality, R-tree 128 has only 7 records per index node, and 15 per data node. This leads to a very high amount of splits, a very costly operation in R-trees compared to B-trees. As such R-trees seem to be mainly CPU bound.

7.4 Filtering

Filtering is the most important operation in a metric indexing system, and is where the different indexing methods we propose are most important. Because the candidate size is identical for all our methods at the same number of pivots, the additional time to perform post processing will also be identical. As have been seen B-trees and heap files are clearly faster during indexing, however while databases are only indexed during creation and insertion of new objects, filtering will be performed for every query.

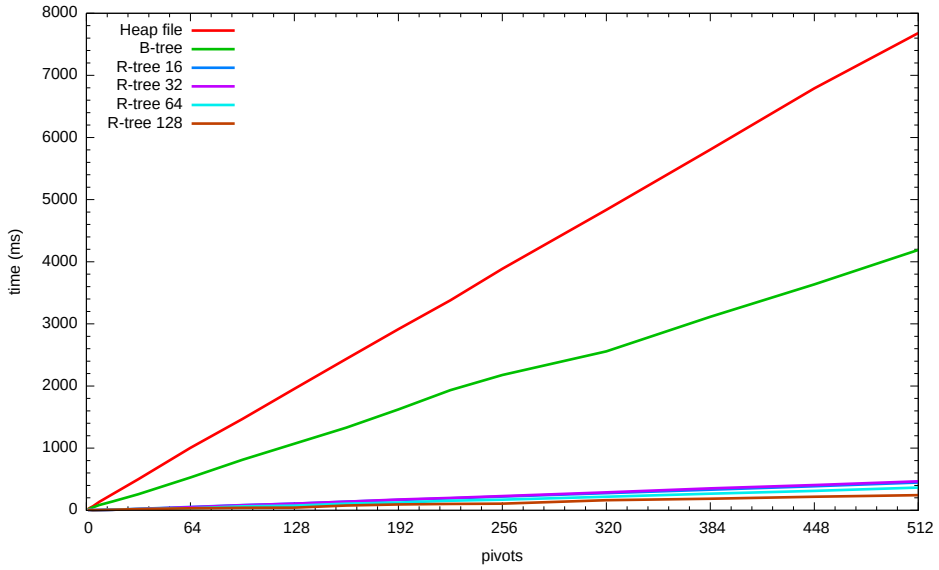


Figure 7.7: Response time for filtering process with $r = 0.2$

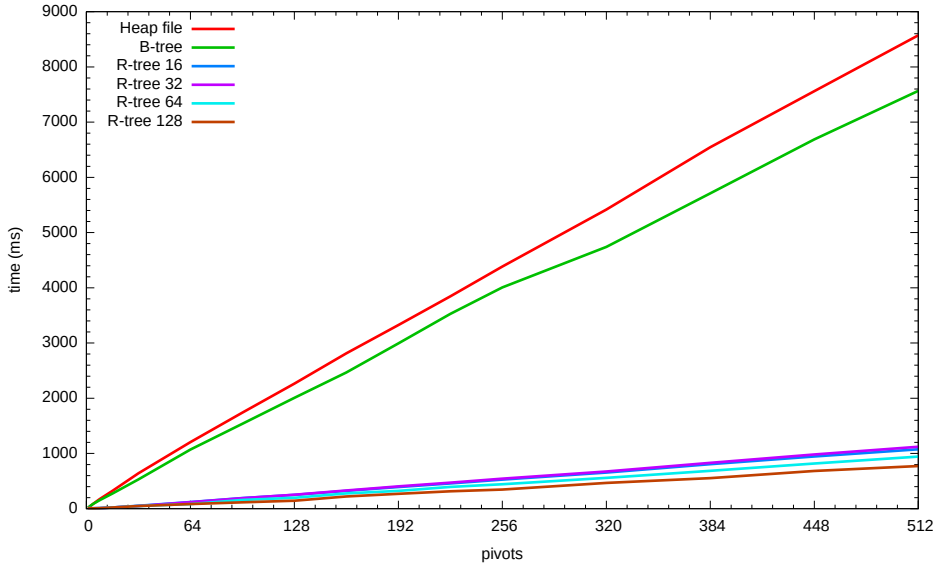


Figure 7.8: Response time for filtering process with $r = 0.4$

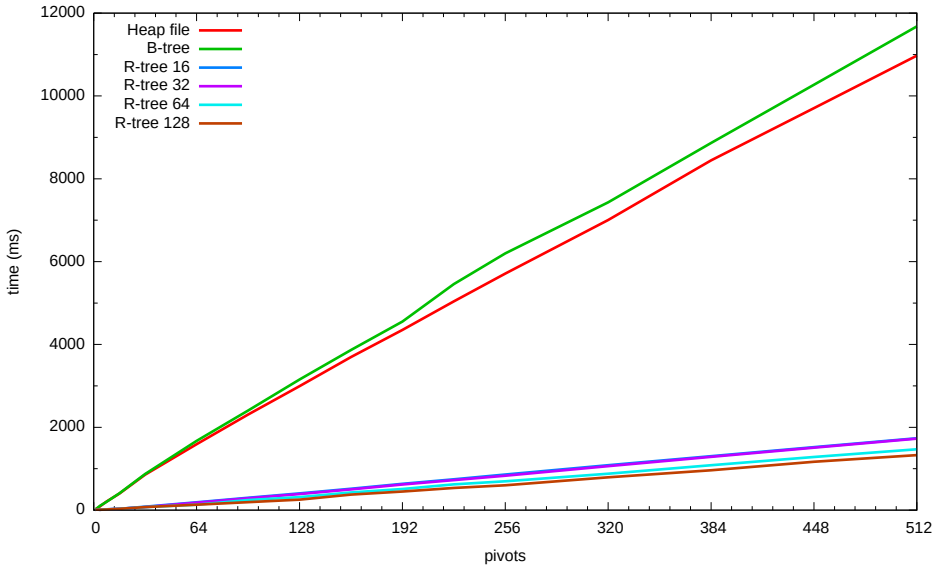


Figure 7.9: Response time for filtering process with $r = 0.6$

As the filtering depends heavily on the range limit used, filtering time is shown for $r = 0.2$ in Figure 7.7, $r = 0.4$ in Figure 7.8 and $r = 0.6$ in Figure 7.9. What is immediately obvious is that while B-trees work well at $r = 0.2$, they do not have any advantage over heap files at $r = 0.4$ and actually perform worse at $r = 0.6$. This can be explained by looking at the distribution graph in 6.3 and expected result set size in 6.11, and remembering that each pivot can only discard the objects known to be more than r from the query object. This gives an interval of $2r$ that must be included for each pivot, for $r = 0.6$ this is an average of 80%-95% of each index file. As such B-trees only give an advantage at low range limits.

R-trees however has the advantage of combining the pruning power of several pivots, and thereby reading far smaller portions of the index file. This leads to an enourmous improvement in filtering time, almost an order of a magnitude faster than B-trees. It is also evident that the highest dimensional R-tree, combining 128 pivots in a single index structure, is the fastest. However the lower dimensional trees are not much slower, and in fact 16 is virtually identical with 32.

The number of file pages read is shown in Figures 7.10, 7.11 and 7.12. For the heap file the page count is identical and independent of the range limit, as this is a pure file scan. For B-trees we see a clear advantage at $r = 0.2$ reading almost half as many pages, however for $r = 0.4$ it is identical to heap files, and for $r = 0.6$ it is much worse. For the latter the B-tree method is almost as bad as a full scan of the entire data level due to the low pruning power of a single pivot.

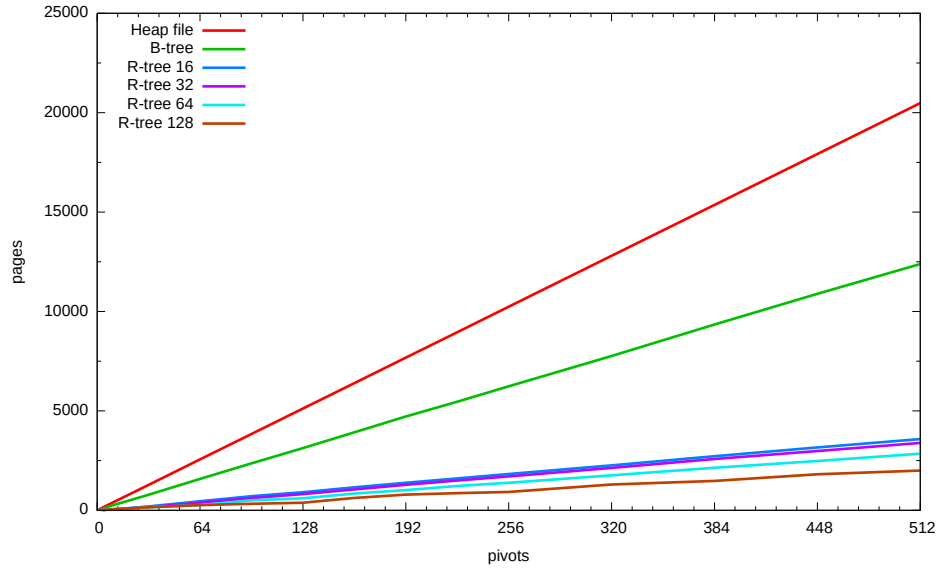


Figure 7.10: File pages read for filtering process with $r = 0.2$

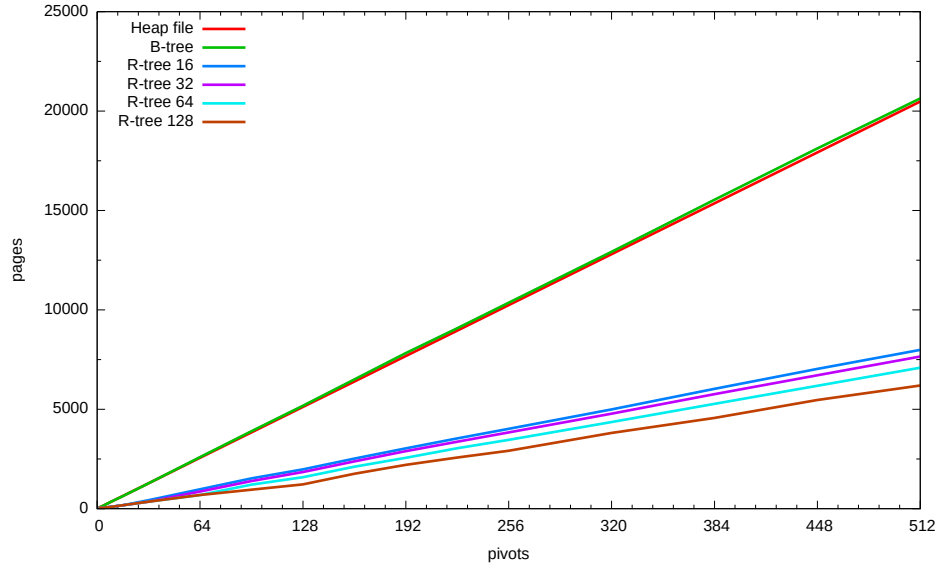


Figure 7.11: File pages read for filtering process with $r = 0.4$

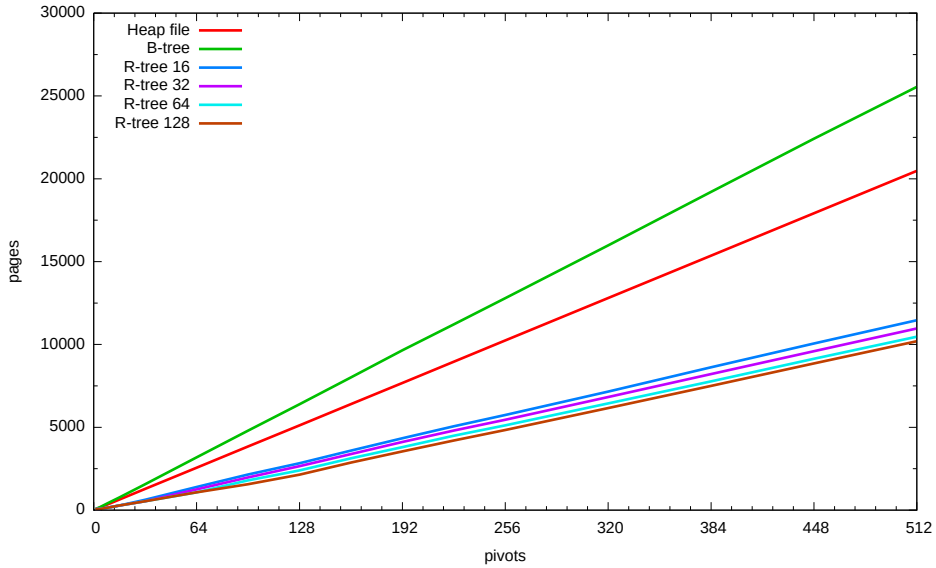


Figure 7.12: File pages read for filtering process with $r = 0.6$

R-trees are once again far better, reading only one fifth the number of pages for $r = 0.2$. For the larger range limits more pages are needed here as well, but still below half of that needed for the B-tree based filtering. Once again the dimensionality of the R-trees separate them slightly, with the higher dimensionality leading to fewer page reads.

7.5 Range and Nearest Neighbour Query

Range queries are one of the fundamental query types in a metric space, and also a fundamental part of our filtering process. Nearest neighbour queries are however more intuitive for end users, and is probably the most used query type. The results from the previous section could make it seem that fewer pivots are better, because of the linearly rising cost per pivot added. However the filtering process only returns candidate keys, and each of these must be fetched from disk and compared to the query object before inclusion in the result set. For nearest neighbour queries, the LAESA algorithm enables a large fraction of the candidates to be eliminated without any fetch or comparison, and as such saves considerable time compared to range queries.

The main cost in this experiment, in addition to filtering, is related to reading the candidate objects from disk and performing distance calculations. As our experiments are performed using Euclidean distance in a low-dimensionality vector space, the time to perform distance calculations is almost negligible. For this reason we have weighted the distance calcu-

lations that are performed with an extra 1ms to get a result closer to what a real world application would see. Note that as the number of candidates is equal for all the indexing methods, this only affects the number of optimal pivots.

r	$N(R)$	$N(R_{10-NN})$
0.2	17.68	3.58
0.4	129.84	8.64
0.6	753.30	9.92

Figure 7.13: Number of results for random query set

Figure 7.13 shows the average number of results for our random query set for both range and 10-NN queries. We will concentrate on the $r = 0.4$ limit as this gives a fair amount of results for range queries, while returning nearly 10 nearest neighbours for all query objects except for a few outliers. The results of the range query are shown in Figure 7.14 and the 10-NN query in Figure 7.15, both with varying number of pivots.

From this we see that the optimal number of pivots for Heap files and B-trees are about 32 for both query types, while the R-trees optimal number of pivots is 128. All the R-tree methods also perform in under half the time with the optimal number of pivots. It is clear that the much lower overhead for filtering with a large number of pivots using R-trees allows for better pruning, and thus less time for fetching and comparing data objects. Performing the given queries using sequential scan with the given presumptions would take slightly over 40 seconds. This is mainly due to the cost of distance calculations set at 1ms, and will vary greatly depending on the data set size and distance calculation cost.

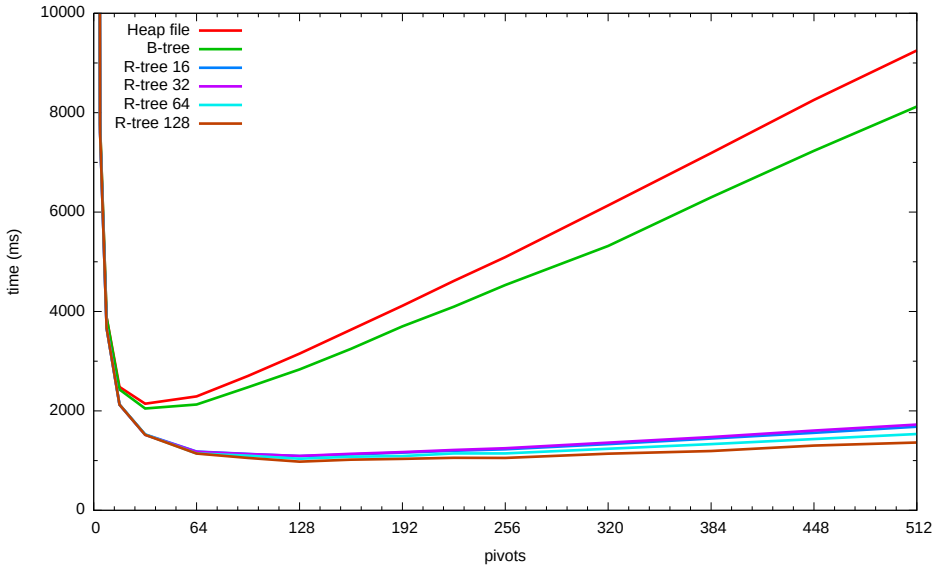


Figure 7.14: Response time for range query with $r = 0.4$

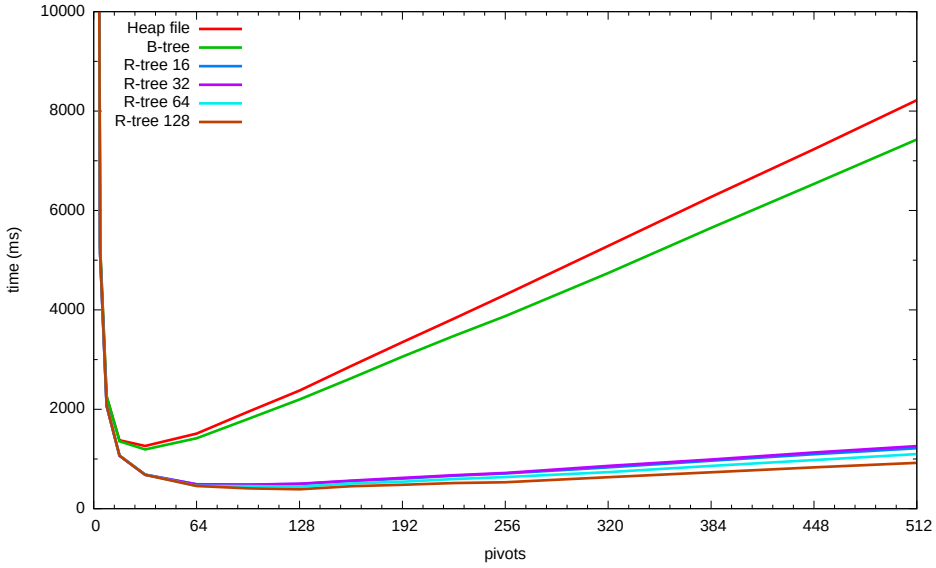


Figure 7.15: Response time for 10-NN query with $r = 0.4$

Chapter 8

Parallel Processing

We have now looked at how well our method fares while using several different access methods. In our experiments so far, only a single process with a single thread was used for the processing, leading to a strictly serial execution. We will now look at how the method fares in relation to both raw performance figures and the relative speedup achieved compared to the serial tests in the previous chapter. The speedup is the most important factor here, and should optimally be linear with the number of processors. We will use the same system for testing as in the previous chapters (see Figure 7.1), except we will now allow the system to use 4 threads during execution, thus utilizing all four processor cores.

Optimally one should see speedups of up to the number of threads, depending on how saturated the disk and CPU is respectively. However, as we use only a single disk (actually two disks in RAID-1) for the experiments, this is probably unlikely. This will clearly give the indexing and filtering methods that were CPU bound during the last experiment will have a better chance at a higher speedup. The usage of four CPU cores against a single disk is consistent with where modern computer architecture is moving, where continued processor speed gains are now achieved using multi-processor and multi-core systems.

8.1 Indexing

Figure 8.1 shows the indexing time when using this setup. What is clearly visible is that the disadvantage of R-trees compared to B-trees is now completely gone. While Heap files still remain the fastest indexing method, the difference is now minimal compared to all the R-trees except the 128-dimensional. It is not surprising that the higher-dimensional R-trees are not as fast, as they do not have enough pivots to divide them among all the executors. If more pivots were added to allow a higher number of trees, these high-dimensional trees would most likely catch up to the performance of

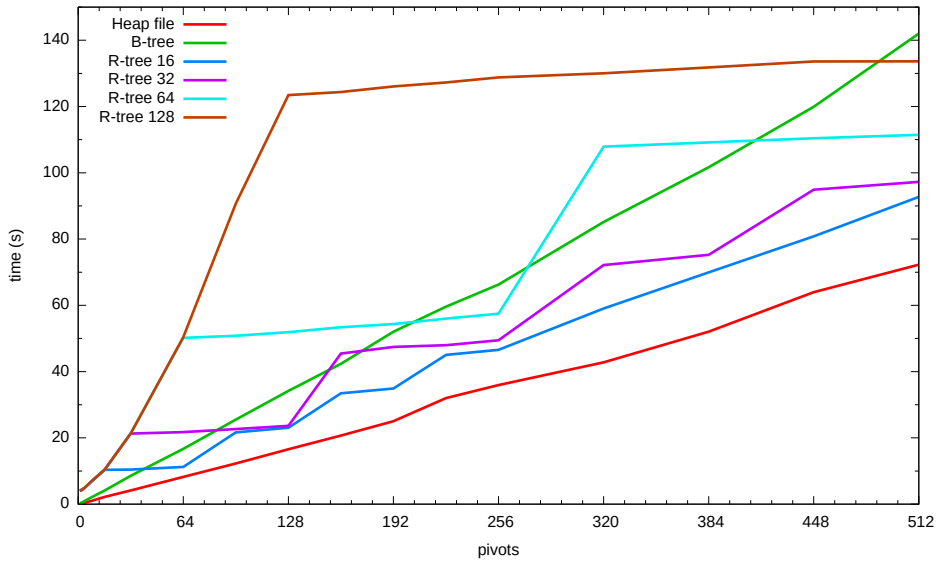


Figure 8.1: Time spent indexing for different access structures using 4 threads in parallel.

the lower dimensional trees.

The main reason for the increased performance for R-trees is because they were not IO bound in the previous experiments. In fact R-trees write far fewer pages to disk compared to B-trees, and even fewer than Heap files (see Figure fig:exp-indexing-file). Thus when it is allowed to use more CPU resources, it is also able to utilize more of the available disk resources and thus gain a significant speed advantage.

In figure 8.2 the speedup, given by $S_p = \frac{T_1}{T_p}$ is shown for the indexing response time using 4 threads compared to the single thread run in the previous chapter. This clearly shows that adding more threads makes virtually no difference to either B-trees or Heap files and both stay at a speedup of about 1.0 (that is no speedup). In fact Heap files are slightly lower than this, indicating a slowdown due to the multi-threading. This is not surprising, as what was previously a more or less continuous disk write is now interrupted by three other threads doing the same. B-trees similarly have a fairly easy structure, and as such require little computation and thus are also limited by disk IO (however, B-trees use more random IO as writes are spread across the various pages).

R-trees however have a very complex structure and especially splitting pages is an extremely costly operation. Because of this they have a huge speedup advantage in indexing, and in fact come very close to a linear speedup of $S_4 = 4$. What is evident is that R-trees have a disadvantage when there are not enough pivots to fill 4 full trees. 16-dimensional R-trees

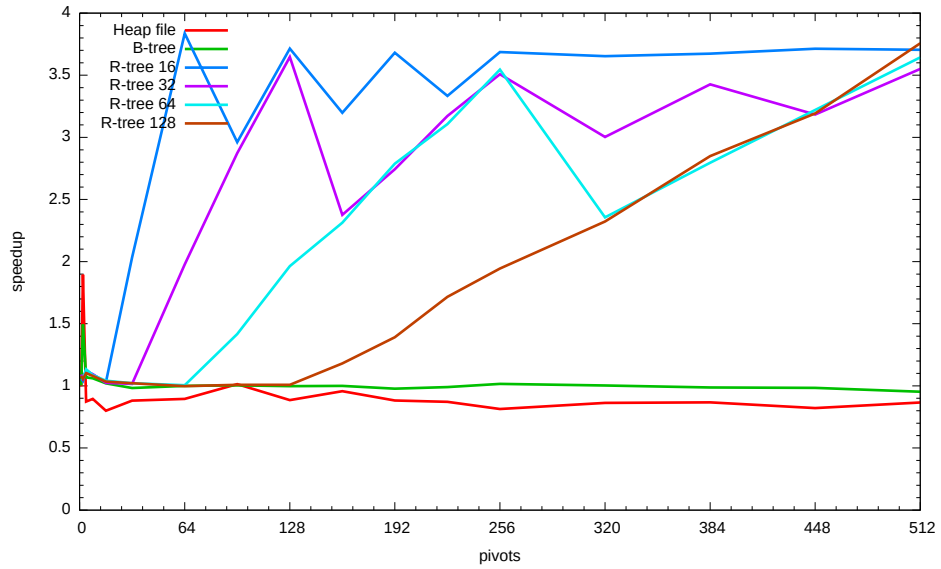


Figure 8.2: Speedup, S_4, of indexing for different indexing methods using 4 threads in parallel.

do not reach their potential for speedup until at least 64 pivots are used ($16*4 = 64$), similarly 32-dimensional require $32*4 = 128$ pivots, up to 128-dimensional that requires $128 * 4 = 512$ pivots. As such lower-dimensional R-trees are probably more suitable in situations where a lower number of pivots, or a large number of execution threads, are used.

8.2 Filtering

The response times for filtering is given in Figures 8.3, 8.4 and 8.5 for $r = 0.2$, $r = 0.4$ and $r = 0.6$ respectively. There is not much relative change in the figures compared to the single-threaded experiments, there are decreases in the absolute response times but overall there is no major variation as was seen with indexing. This is mainly because of the enormous advantage to all the R-trees.

The speedup comparison is shown Figure 8.6, 8.7 and 8.8 for $r = 0.2$, $r = 0.4$ and $r = 0.6$ respectively. This shows that there was a speedup of between 1.5 and 2.5 for all access structures, however nowhere near the linear speedup we saw for indexing. This is not surprising considering we made no increases in the number of disks, and filtering is mainly a large number of semi-sequential reads from disk. Even for R-trees the computational complexity of queries is far less and as such there is not as much room for improvement.

It is however still clear that the lower-dimensional R-trees have the

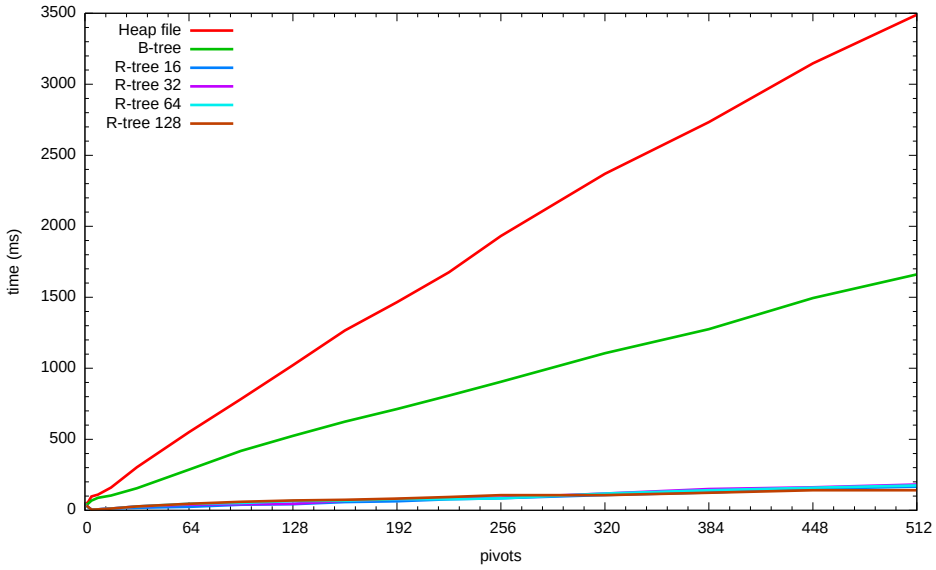


Figure 8.3: Response time for filtering process with $r = 0.2$ using 4 threads

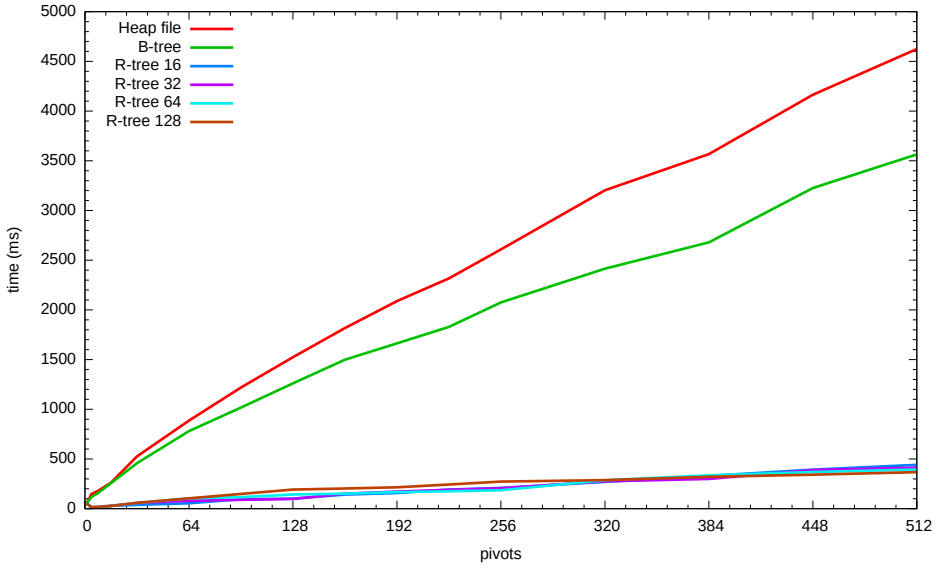


Figure 8.4: Response time for filtering process with $r = 0.4$ using 4 threads

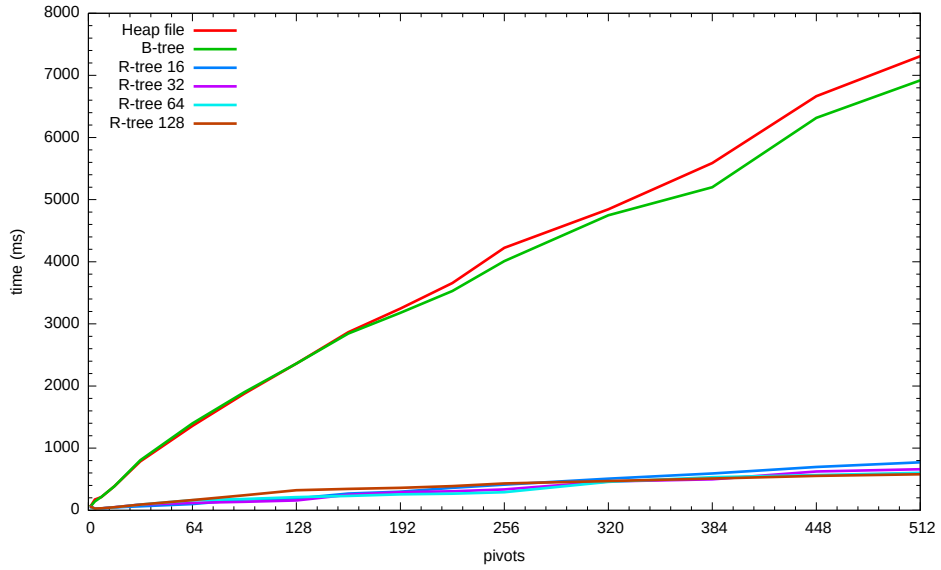


Figure 8.5: Response time for filtering process with $r = 0.6$ using 4 threads

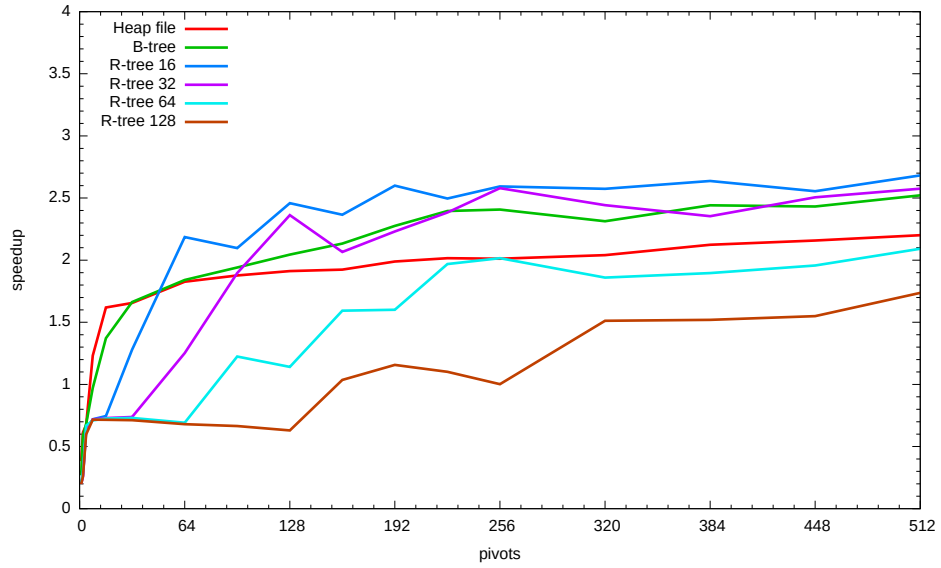


Figure 8.6: Filtering process speedup with $r = 0.2$ using 4 threads

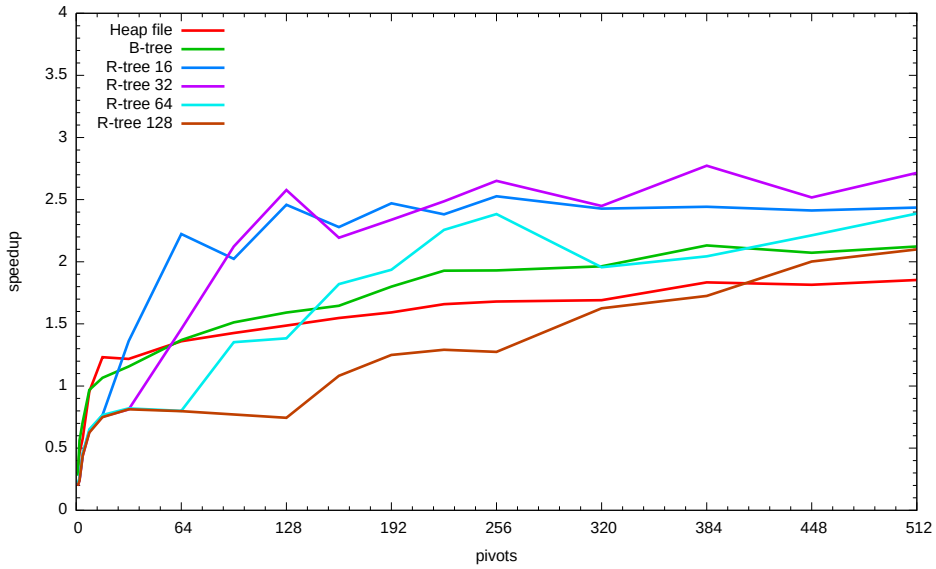


Figure 8.7: Filtering process speedup with $r = 0.4$ using 4 threads

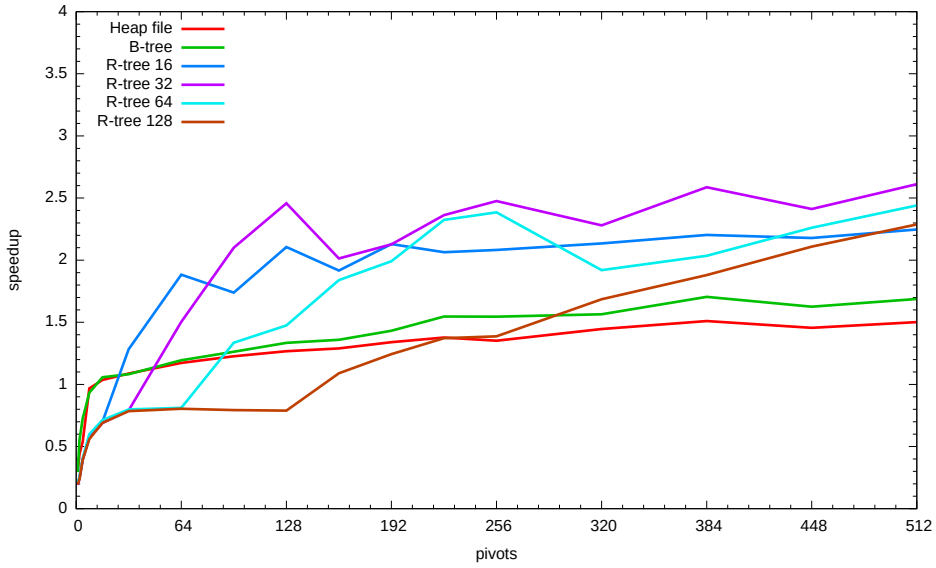


Figure 8.8: Filtering process speedup with $r = 0.6$ using 4 threads

greatest advantage, this is of course for the same reason as before. The larger dimensional R-trees simply need far more pivots to be able to create enough trees to utilize all the threads completely.

8.3 Discussion

As we have seen, R-trees have a higher advantage to parallelization than both B-trees and Heap files. This is mainly because of the much higher CPU cost in R-trees compared to B-trees, especially related to indexing where a huge amount of splits occur and the area and overlap needed for this are fairly complex calculations. It does however require there to be enough R-trees to actually divide them among the available processors. As such using lower dimensional R-trees would probably be advantageous. This would also simplify adding more pivots after the initial indexing.

Overall it is clear that R-trees have an even greater advantage over B-trees when used on multi-processor or multi-core machines. For indexing we very nearly got a linear speedup, which is remarkable considering the number of disks was not increased. We did not see as large an increase in speedup during filtering, however this is not surprising considering the lack of increased disk resources.

Chapter 9

Conclusion

Our report shows that pivot based filtering can be implemented using several standard database access structures and join methods. Heap files, B-trees and R-trees are all available in modern database management systems, and can all be used to implementing pivot based search. In fact, any database access method that supports a range selection can be used for pivot filtering.

We also studied and tested several different types R-trees, settling on a R*-tree with some of its optimizations turned off as they had a negative effect on CPU usage and response time. This was especially important in very high-dimensional trees, where we found that CPU time is the major bottleneck. Our implementation of R*-trees work well up to 200 dimensions, and this is currently only limited by the small block size and a minimum requirement of five records per node. We did see increased response time and CPU usage during insertions on trees with more than 150 dimensions, however query processing performance was not impacted. The reason for increased response time for inserts is the enormous amount of splits that are caused by the small number of records per node, and as such an increased block size should allow even higher-dimensional trees with our implementation.

Using pivot based filtering can greatly reduce the cost of similarity search in large databases, both for range and nearest neighbour queries. This is especially true when comparing complex objects, like images or documents, with an expensive and time consuming distance measure. Pivot based filtering can reduce the number of distance calculations needed greatly compared to a full scan and comparison with all objects. This is done by filtering out objects that are guaranteed to be outside the query region, using the lower and upper bounds as seen from the pivot.

The candidates returned from the filtering process must be processed and, depending on query type, either the distance from query object to all or parts of the candidate set will need to be calculated exactly. We have ex-

perimented with varying number of pivots and varying range limits during filtering. While adding pivots will continually decrease the number of candidates, it also increases both indexing and processing time. In addition, as the distance between the query object and all pivots used for filtering must be calculated, adding more than the optimal number of pivots will increase the number of distance calculations.

We found that the optimal number of pivots for a query depends both on the query type and, most importantly, on the range limit. This is important when designing databases for use with varying range limits, and suggests using a variable number of pivots for filtering depending on the query specification. The range limit also depends on the data distribution as we have shown with several range distribution histogram for various data sets and metrics. This also shows that different metrics have widely different results in performance and data distribution.

The reduced distance calculations when querying is of course with the added cost of doing pre-indexing. However, as each pivot added to the index costs the same as doing a single full scan and comparison with all objects, this cost is quickly amortized in a read-mostly database. Indexing can also be performed during idle time, or in periods of low activity. We have implemented pivot selection algorithms that exploit our indexing structure and already calculated distances, allowing the selection of a new pivot without performing a single distance calculation (with the exception of the initial pivot). The algorithm performs in linear time to the number of pivots when selecting a new candidate and can return a list of p pivots in $O(p * n)$ time.

Indexing is not surprisingly cheapest using heap files, with B-trees performing only slightly worse. R-trees require significantly more index time, especially with high dimensionality. As mentioned earlier this is due to the high amount of splits that occur when the number of records per node drops below 20, and the high CPU cost related to such splits in the R*-tree. During indexing the R*-tree is clearly bound by the processor speed, as the number of disk accesses are much lower than for B-trees.

When using several processors in parallel this causes R-trees to perform significantly better, and actually surpass B-trees when using low-dimensional R-trees or a large number of pivots. B-trees would probably perform far better if each processor had a separate disk. However, both the number of processors and cores and speeds are increasing far faster than the number of disks in contemporary systems, and it is unlikely that servers of the future will contain a similar number of disks and processor cores.

The biggest contribution from this report and our experiments is clearly the comparison of filtering with different access methods. We found that for query processing, R-trees perform significantly faster than B-trees. Specifically filtering can be performed more than 5 times faster than B-trees. And

as both methods scale linearly with the number of pivots, R-trees gain an enormous performance advantage with a high number of pivots. Also R-trees handles large query range limits without degradation in performance, something that is a major issue with B-trees making them near unusable for high range limits. This is due to the combined pruning power of several pivots in the same index, thus reducing the number of disk pages that must be fetched and candidate sets that must be joined. We also see a significant drop in the number of IO operations when using R-trees. R-trees are thus a significant improvement over B-trees and heap files, both in the number of disk accesses and response time.

The dimensionality of R-trees does have an impact on both indexing and query performance. While high dimensional R-trees are slower to index, they also have a lower response time for queries. What was surprising was however how little difference experiments showed from 16 to 128 dimensions during query processing. As such using several lower dimensional trees will give the advantage of increased flexibility and lower indexing time, with only a very slight performance loss. This is both because of the CPU time used to process and compare the high dimensional records, as well as the few records per node causing more wasted space and a lower fanout.

Chapter 10

Further Work

Several enhancements and optimizations are possible, and together could have a substantial impact on the performance of the implementations tested in this report. This is especially true for B-trees, but also applies to R-trees and the join-based filtering technique used. While these enhancements are outside the scope of this project and report, we will mention them here so the method developed further in the future by other teams.

Different caching techniques have a large effect on the performance. In this report a naive Least Recently Used (LRU) cache has been employed, a method that is far from ideal for use with scanning operations. Better caching of index blocks, and pre-fetching or group-fetching of page files during scan would probably reduce query times.

Sorting of index and data blocks in B-trees, or the use of bulk-loading or packing algorithms like STR for R-trees [19], would give a better locality of data in addition to increasing the density of the index structure. This would both reducing the total size of the index file, and reduce the number of file pages the must be read.

While experiments were performed with several executor threads working in parallel on a shared-everything machine, no work was performed in testing on shared-memory or shared-nothing machines as this was outside the scope of this report. It should however be trivial to implement support for a distributed environment based on our method.

While some join methods were tested briefly in this report, further work in this area is certainly possible. For parallel systems it would probably be advantageous to perform hash joins using the split and merge operators described in [8]. This would probably be especially important in a shared-nothing environment, where large amounts of data must be transferred between processes and nodes.

For the tests performed in this report only static datasets were tested. In real world applications the dataset would likely evolve over time, possibly with large parts of the data volume added, updated or deleted daily.

This is especially true when the data is gathered from the internet or other sources with many users performing simultaneous updates. In some cases it might be beneficial to add more pivots as the data volume grows and evolves, in that case B-trees or low-dimensionality R-trees would have a clear advantage as single, or small groups of, pivots can be added easily. As the method supports using a heterogeneous set of indexing methods, it would also be possible to combine the added B-trees or low-dimensionality R-trees into high-dimensionality R-trees at a later time.

Finally it would be interesting to see how standard database management systems that support both B-trees and R-trees would perform using the techniques described. For instance PostgreSQL, MySQL and IBM DB2 all support both B-trees and R-trees, and can be used to test the filtering methods described here using real world database management systems. It would also be interesting to test how well these methods would perform on search architectures such as the FAST Distributed Processing Architecture (DPA) [29].

Bibliography

[1] Kent Beck, Erich Gamma, and David Saff. JUnit. `http://junit.org`. Collected September 22, 2008.

[2] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles. In *SIGMOD '90: Proceedings of the 1990 ACM SIGMOD international conference on Management of data*, pages 322–331, New York, NY, USA, 1990. ACM.

[3] Benjamin Bustos, Gonzalo Navarro, and Edgar Chávez. Pivot Selection Techniques for Proximity Searching in Metric Spaces. *Computer Science Society, 2001. SCCC 2001. Proceedings. XXI Internatinal Conference of the Chilean*, pages 33–40, 2001.

[4] Jr. Caetano Traina, Agma J. M. Traina, Bernhard Seeger, and Christos Faloutsos. Slim-trees: High performance metric trees minimizing overlap between nodes. In *EDBT '00: Proceedings of the 7th International Conference on Extending Database Technology*, pages 51–65, London, UK, 2000. Springer-Verlag.

[5] Edgar Chávez, J. L. Marroquín, and Ricardo Baeza-Yates. Spaghettis: An array based algorithm for similarity queries in metric spaces. In *SPIRE '99: Proceedings of the String Processing and Information Retrieval Symposium & International Workshop on Groupware*, page 38, Washington, DC, USA, 1999. IEEE Computer Society.

[6] Edgar Chávez, Gonzalo Navarro, Ricardo Baeza-Yates, and José Luis Marroquín. Searching in metric spaces. *ACM Computing Surveys*, 33(3):273–321, September 2001.

[7] Paolo Ciaccia, Marco Patella, and Pavel Zezula. M-tree: An efficient access method for similarity search in metric spaces. In *VLDB '97: Proceedings of the 23rd International Conference on Very Large Data Bases*, pages 426–435, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc.

[8] David DeWitt and Jim Gray. Parallel database systems: the future of high performance database systems. *Communications of the ACM*, 35(6):85–98, 1992.

[9] Donghui Zhang. NEUStore: A Simple Java Package for the Construction of Disk-based, Paginated, and Buffered Indices. CCIS, Northeastern University, `http://www.ccs.neu.edu/home/donghui/research/neustore/`, September 2005. Collected August 21, 2008.

[10] Karina Figueroa, Edgar Chávez, Gonzalo Navarro, and Rodrigo Paredes. On the least cost for proximity searching in metric spaces. In *Proceedings of the 5th International Workshop on Efficient and Experimental Algorithms (WEA)*, volume 4007, pages 279–290. Springer, 2006.

[11] Roberto Figueira Santos Filho, Agma Traina, Caetano Traina Jr., and Christos Faloutsos. Similarity Search without Tears: The OMNI-Family of All-Purpose Access Methods. In *ICDE '01: Proceedings of the 17th International Conference on Data Engineering*, pages 623–630, Washington, DC, USA, 2001. IEEE Computer Society.

[12] Antonin Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. In *Readings in database systems*, pages 599–609. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1988.

[13] Magnus Lie Hetland. The basic principles of metric indexing. *Swarm Intelligence for Multi-objective Problems in Data Mining*, 2009. To appear.

[14] Andy Hunt and Dave Thomas. *Pragmatic Unit Testing in Java with JUnit*. The Pragmatic Programmers, September 2003.

[15] Masahiro Ishikawa, Hanxiong Chen, Kazutaka Furuse, Jeffrey Xu Yu, and Nobuo Ohbo. Mb+tree: A dynamically updatable metric index for similarity search. In *Proceedings of the First International Conference on Web-Age Information Management*, pages 356–373. Springer, 2000.

[16] H. V. Jagadish, Beng Chin Ooi, Kian-Lee Tan, Cui Yu, and Rui Zhang. iDistance: An Adaptive B+-tree Based Indexing Method for Nearest Neighbor Search. *ACM Transactions on Database Systems*, 30(2):364–397, June 2005.

[17] Karina Figueroa and Gonzalo Navarro and Edgar Chávez. Metric Spaces Library. `http://sisap.org/Metric_Space_Library.html`, November 2008. Collected November 21, 2008.

[18] Eugene F. Krause. *Taxicab Geometry: An Adventure in Non-Euclidean Geometry*. Dover Publications, New York, NY, USA, 1986.

[19] Scott T. Leutenegger, Jeffrey M. Edgington, and Mario A. Lopez. STR: A simple and efficient algorithm for r-tree packing. Technical Report 97-14, Institute for Computer Applications in Science and Engineering (ICASE), February 1997.

[20] V. I. Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady*, 10:707–710, February 1966.

[21] Yannis Manolopoulos, Alexandros Nanopoulos, Apostolos N. Papadopoulos, and Yannis Theodoridis. *R-Trees: Theory and Applications*. Springer, Vienna, Austria, third edition, September 2005.

[22] Luisa Micó, Jose Oncina, and Rafael C. Carrasco. A fast branch & bound nearest neighbour classifier in metric spaces. *Pattern Recognition Letters*, 17(7):731–739, 1996.

[23] María Luisa Micó, José Oncina, and Enrique Vidal. A new version of the nearest-neighbour approximating and eliminating search algorithm (aesa) with linear preprocessing time and memory requirements. *Pattern Recognition Letters*, 15(1):9–17, January 1994.

[24] Francisco Moreno-Seco, Luisa Micó, and José Oncina. Extending laesa fast nearest neighbour algorithm to find the k nearest neighbours. In *Proceedings of the Joint IAPR International Workshop on Structural, Syntactic, and Statistical Pattern Recognition*, pages 718–724, London, UK, 2002. Springer-Verlag.

[25] Oscar Pedreira and Nieves R. Brisaboa. Spatial Selection of Sparse Pivots for Similarity Search in Metric Spaces. In *Proceedings of the 33rd Conference on Current Trends in Theory and Practice of Computer Science, number 4362 in Lecture Notes in Computer Science*, pages 434–445, 2007.

[26] D.T. Phama, Y.I. Prostovb, and M.M. Suarez-Alvareza. Statistical approach to numerical databases: clustering using normalized Minkowski metrics. In *Proceedings of the Second Virtual International Conference on Intelligent Production Machines and Systems*, pages 356–362, Oxford, UK, 2006. Elsevier.

[27] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. McGraw-Hill, Boston, MA, USA, 3 edition, 2003.

[28] Juan Ramón Rico-Juan and Luisa Micó. Comparison of aesa and laesa search algorithms using string and tree-edit-distances. *Pattern Recognition Letters*, 24(9-10):1417–1426, 2003.

[29] Knut Magne Risvik, Børge Svingen, Tor Egge, and Arne Halaas. *The FAST Distributed Processing Architecture (DPA) and its Application for a Large-Scale Search Engine*. Dr.philos thesis, Department og Computer and Information Science at the Norwegian University of Science and Technology (NTNU), May 2004. NTNU 2004:54, ISBN 82-471-6318-7.

[30] E V Ruiz. An algorithm for finding nearest neighbours in (approximately) constant average time. *Pattern Recognition Letters*, 4(3):145–157, 1986.

[31] Sam Chapman. String Similarity Metrics for Information Integration. Department of Computer Science, University of Sheffield, `http://www.dcs.shef.ac.uk/~sam/stringmetrics.html`, 2006. Collected March 20th, 2009.

[32] Sam Chapman. SimMetrics: Similarity Metric Library. SourceForge, `http://sourceforge.net/projects/simmetrics/`, February 2007. Collected March 20th, 2009.

[33] Václav Snásel, Jaroslav Pokorný, and Karel Richta, editors. *Proceedings of the Dateso 2004 Annual International Workshop on DAtabases, TExts, Specifications and Objects, Desna, Czech Republic, April 14-16, 2004*, volume 98 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2004.

[34] TREC. Text REtrieval Conference (TREC) Terabyte Track: GOV2 Test Collection. University of Glasgow, `http://ir.dcs.gla.ac.uk/test_collections/gov2-summary.htm`, May 2004. Collected March 20th, 2009.

[35] Enrique Vidal. New formulation and improvements of the nearest-neighbour approximating and eliminating search algorithm (aesa). *Pattern Recognition Letters*, 15(1):1–7, 1994.

[36] Juan Miguel Vilar. Reducing the overhead of the aesa metric-space nearest neighbour searching algorithm. *Information Processing Letters*, 56(5):265–271, 1995.

[37] Cui Yu, Beng C. Ooi, Kian-Lee Tan, and H. V. Jagadish. Indexing the distance: An efficient method to knn processing. In *VLDB '01: Proceedings of the 27th International Conference on Very Large Data Bases*, pages 421–430, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[38] Li Yujian and Liu Bo. A Normalized Levenshtein Distance Metric. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 29(6):1091–1095, 2007.

[39] Pavel Zezula, Giuseppe Amato, Vlastislav Dohnal, and Michal Batko. *Similarity Search: The Metric Space Approach*. Springer, November 2005.