NTNU
Norwegian University of
Science and Technology

Ptolemaic Indexing
An Evaluation

Eirik Benum Reksten

Master of Science in Computer Science
Submission date: June 2010
Supervisor: Magnus Lie Hetland, IDI

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

Problem Description

Ptolemaic indexing is indexing for similarity search based on a subset of metric distance measures (ptolemaic distance measures). This thesis should study the performance of ptolemaic indexing on several data sets, and compare this to the performance of other indexing techniques, such as spatial indexing.

Assignment given: 22. January 2010
Supervisor: Magnus Lie Hetland, IDI

Contents

List of Figures

List of Tables

Acknowledgements

I would like to thank my supervisor, *Magnus Lie Hetland*, for his guidance and patience. I very much appreciate his effort to help me, despite me being in a different part of the country for most of the process.

Second, I would like to thank my fiancee *Kathrine Høybakk*. She has been very supportive throughout my work, and given me space when I needed to focus.

My thanks also goes out to my family, for always being there when I need them, and for motivating me; *Inger Benum*, *Leiv Reksten*, *Siv Helen Reksten*, *Markus Reksten*, *Johanna Reksten*, *Magnhild Reksten*, *Aud Otilie Benum* and late *Olaf Benum*. I would especially like to mention my grandfather *Kjartan Reksten* who, in an age of almost 80, is learning to use a computer. He is a true inspiration, and make me very proud!

Finally, I would like to thank *Eamonn Keogh*, for providing time series testdata. This certainly made the work slightly easier.

Chapter 1

Introduction

Similarity Search is finding more and more uses in computational data processing. Examples include searching for images or sound, recognizing speech, and much more. Even Google are doing similarity searches, as it is possible to search for similar images there. Similarity in such searches are usually defined as functions returning a lower number for similar items and a higher number for dissimilar ones.

Within similarity search, there are primarily two paradigms in use to speed up the search. The first is spatial indexing, where objects are viewed as point vectors in an n-dimensional space. Sections of the space that lie far from the query object (the one we want to find objects similar to) can be discarded without further processing, thus speeding up the search.

The other paradigm is distance based indexing, often referred to as metric indexing. A metric index stores only distance data, and uses lower bounds on distances to discard parts of the index during search. This requires the distance function to have certain properties. More specifically, it needs it to be triangular.

While metric indexing is the only choice when the data cannot be represented in a vector space, these are competing when it can. In this thesis, spatial and metric indexing will be empirically tested and compared on similar data sets, seeking answer to the following:

> *How does spatial indexing compare to distance based on quadratic form distances as the dimensionality of the data increases?*

Higher dimensionality on the data means more exact searches, due to more object information being stored in the index. This involves various indexing challenges, though. According to [19], metric indexes might be more suited for dealing with these challenges. In [9], however, they've observed a different case, in that a spatial index (the R-tree) outperformed the metric one in certain cases. Other studies again have shown that at a high enough dimensionality, indexing becomes inferior to even a sequential scan [5].

Recently, another possibility for distance-based indexing have been presented in [13]. It shows that for quadratic form distances, including the euclidean distance,

Ptolemy's Inequality can be used to discard a potentially larger part of the search space than metric indexing without risking false negatives. This *Ptolemaic Indexing* was only tested on a standard pivot based indexing method (LAESA), meaning that all objects still would need to be considered. This can be helpful to decrease the amount of distance comparisons, but can not reach the potential a tree-based method has.

As this could be an important factor when comparing spatial to distance-based indexing methods, we will also take a closer look at the following.

> *Can tree structured distance based indexes benefit from making use of the ptolemaic inequality as a means of pruning the search space?*

For the testing in this thesis, three indexes has been implemented; A spatial index (X-tree), a metric index (the PM-tree) and a ptolemaic index (a modified PM-tree). In addition, a sequential list able to perform the same similarity search operations was implemented for comparison. These are all tested on quadratic form distances with varying dimensionality.

Chapter 2

Background

2.1 Similarity Search

Similarity searching is searching based on some similarity measure between data objects. This has become widely used over the last years, and has applications among other things within genetics, image recognition, voice recognition, speech processing, video compression and data mining [7].

The basis for similarity search is a distance function able to measure how different two of the data objects are. These are often based on a feature vector describing the object. Examples include feature vectors for images or time series. There are, however, other approaches as well. The Levenshtein distance, for instance, can work directly on character strings.

2.1.1 High-Dimensional Data

When the dimensionality for the data increases, so does also the complexity involved in searching said data. Often, distance computations become expensive [19] and the need for efficient pruning of the search space arises. The difficulty of indexing these data also increases. Studies [5] have shown that the relative differences between distances decrease. Combined with an increased amount of noise from the extra dimensions, efficiently indexing the data becomes a challenge.

2.2 Spatial Indexing

Whenever the data resides, or can be represented, in a vector space, there is the possibility that you can utilize some spatial properties of the data and index directly on the hyperspace defined by them. This enables you to search directly in the region of the hyperspace where your candidate objects may be located, thus potentially speeding up the search substantially over a plain linear scan.

2.2.1 R-Tree

One relatively simple spatial index is the R-tree, presented in [12]. The R-Tree indexes multidimensional data by recursively partitioning it into smaller hypercubes. Each internal node in the tree represents one such hypercube. The hypercube is the smallest possible cube that contains all the data objects located in that subtree. It references several, possibly overlapping, subtrees. Leaf nodes contain pointers to the actual data objects being indexed. An example of this hierarchical structure is shown in Figure 2.1.

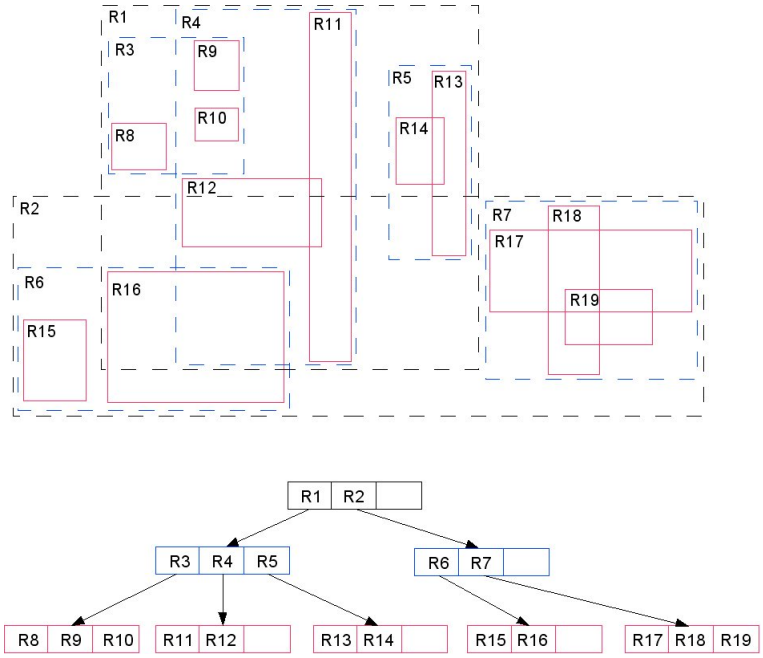


Figure 2.1: Example of an R-tree decomposition in two dimensions

R-tree Building/Insertion

One way of building an R-tree is by starting with an empty tree, and inserting objects sequentially into the tree. Inserting a new object into the R-tree is done recursively, starting from the root. At each level, the insertion algorithm evaluates all the child nodes of the current node, and finds the one that will grow the least from inserting this object there. This process is then repeated recursively, until a leaf node is reached. The new object is inserted into that leaf node.

One R-tree node corresponds to a single disk page. The fanout of the tree is therefore directly related to the disk size. If the above insertion procedure leads to the object being inserted into a node that would need more space than a single disk page, this would have to be dealt with. The solution is to split the node in

two, distributing the objects between the two new nodes. These two new nodes replaces the overflowed node in it's parent.

The splitting of a leaf node like this could of course lead to overflow in its parent as well. If so happens, this node would need to be split as well. This splitting continues upwards towards the root until a node that does not overflow is reached. If the root overflows a new root is created, referencing the two subtrees created when splitting the root.

This has the obvious advantage of the whole tree structure remaining balanced. That is, all paths from the root to a leaf node is of equal length. If, on the other hand, split nodes had extended the tree downwards (away from the root), this would not necessarily be the case. Depending on the order of insertion, that could potentially lead to very bad worst case performance.

Node Splitting

When it comes to how exactly the node splitting should be done, there have been several suggestions [12, 2, 11]. In the original R-tree, Guttman suggests the following procedure [12]. Start with choosing the two elements of the node that would be most inefficient when grouped together. The measure used is simply the area of their union with the area of each of the two elements. Each of these constitute a seed for it's own subtree (the two parts of the split). Then iterate through the rest of the elements, adding each one to the side that would need the least enlargement by adding it there. Alternatively, alternate between adding the closest object to each side to it. The latter would lead to a more balanced tree, while the first might give a better grouping of the objects (for search optimization).

In [11, 2], they suggest using the geometric properties of the hypercube to determine an axis along which to do the split. Once determined, one sorts the subelements on their position along that axis. Then one evaluates the different splits obtained by adding the first m elements to the first side of the split and $N - m$ elements to the second (N is the amount of elements total). The split that results in the least overlap between the two resulting subtrees is selected. Empirical testing have shown that this variant, used in the R*-tree, outperforms Gutmann's original R-tree by up to two orders of magnitude when it comes to search efficiency [2].

Searching the Structure

Searching for similarity in an R-tree as naturally also done in a recursive fashion [12]. Starting at the root, the search function calls itself recursively on every subtree that might contain potential candidates. When reaching a leaf node, it evaluates the distances between the data objects and the query object, adding to the result set each one that's within the search boundaries. Far a range search, this boundary is preset, while a nearest neighbor search has a dynamic bound based on the results obtained so far.

The most important part of the searching of an R-tree is determining which subtrees to search or not. In order to improve search efficiency over a linear data

scan, it is essential that parts of the tree is not searched. The R-tree, and any structure storing minimum bounding hypercubes, can evaluate each dimension independently in this case. A lower bound on the distance between the query object and all elements in the subtree can be found by looking at the distance between the query object and the minimum bounding hypercube in that dimension. If the distance is too high in any dimension, it will be possible to prune away the referenced subtree from the search space.

2.2.2 The Curse of Dimensionality

Though spatial search can be quite simple and intuitive, it is not without drawbacks. There are several reports that the efficiency of R-trees rapidly decrease as the dimensionality of the data increases [5, 3]. In [19] for instance, the curse of dimensionality is mentioned as a problem with this type of index. As the dimensionality increases, spatial indexes tend to become slower than even linear search or they'll use too much space. Moreover, while some dimensions might not contain very much information (the data may essentially lie in a lower-dimensional hyperplane), this isn't recognized by these structures. That leaves you with a lot of overhead in the index. Matters could get even worse when each dimension potentially contains some noise.

Some previous work have been done on mitigating the problems related to high-dimensional data in spatial searches. FastMap [10], for instance, is based on the observation that real data often are highly correlated and clustered. It therefore tries to create a transformation of the data into a lower-dimensional space. Another observation is that most of the information resides in a smaller subsets of the dimensions. The TV-tree [15] is an example of such an approach. It tries to store only the information needed to distinguish between the objects, leading to a more efficiently stored tree with larger fanout. Yet another approach is taken by the makers of the X-tree [3], which will be examined next.

2.2.3 eXtended Node Tree

The eXtended Node Tree (X-tree) is an effort to address the problems the R-tree runs into when the dimensionality of the data increases. Instead of utilizing the observations made in the previous subsection, the creators of the X-tree made some observations on why the R-trees performance declined. They noticed that the overlap between nodes increased proportionally with the decline in search efficiency.

Intuitively, the overlap can be defined as follows [2]. Let $E_1 ... E_P$ be the entries of the current node. The overlap of entry k is defined as

$$overlap(E_k) = \sum_{i=1, i \neq k}^{P} area(E_k Hypercube \cap E_i Hypercube) \tag{2.1}$$

One measure of the overlap in the entire node can then be found by adding these overlap values for the entire tree. Another measure is to count the amount of data objects that are covered by more than one subtree.

When overlap in an index increases, so does the amount of subtrees that need to be searched. The X-tree addresses this problem by avoiding splitting of nodes that would lead to a high amount of overlap between the result nodes. Lets have a closer look at how this index is structured.

Structure of the X-tree

An X-tree consists of two types of nodes, data nodes and directory nodes. Like in the R-tree, these usually map to a single disk page each. All directory nodes are made up of a series of entries referencing subtrees/hypercubes. Each entry is made up of the minimum bounding rectangle of the subtree, a pointer to said subtree, as well as a split history for that node. This structure is depicted in Figure 2.2. A leaf node, on the other hand, is simply a list of data objects or pointers to these.

Figure 2.2: Disk Page Structure for an X-tree Directory Node (*Source: [3]*)

Supernodes

In order to cope with situations where you're not able to split a node without introducing too much overlap, the X-tree introduces the concept of supernodes. Instead of splitting that node, the index allows it to extend across several disk pages. If both nodes would usually need to be accessed anyway, it will be more efficient to simply scan all of their child nodes directly [3]. This way the X-tree saves disk accesses and thus CPU time in cases where the R-tree would usually become inefficient. An example of a tree with supernodes extending across several disk pages is shown in Figure 2.3.

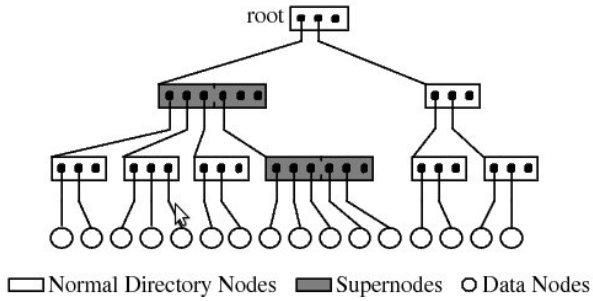


Figure 2.3: Example Structure for an X-tree (*Source: [3]*)

Note that for point data, it will always be possible to find a non-overlapping split at the leaf node level. For this kind of data, supernodes like this will only be located at directory node levels.

Node Splitting

Obviously, the node splitting process for the X-tree has to reliably determine when to split a node and when to create a supernode. The approach taken by the X-tree is to compare the actual overlap and fanout returned by a regular split to index wide parameters [3].

The splitting procedure starts by performing a regular, topological, split of the node, similar to the split procedures of other R-tree variants. It then calculates the overlap value of the two resulting nodes. If this value is below a threshold given by global parameters of the index, the split is successful and it returns the split. If not, however, it continues processing the node.

The next step is trying to determine an overlap-minimal split. An overlap-minimal split is defined as a split where the overlap between the two nodes is as small as possible. To find a split like this, it uses the split history recorded for every directory node in the tree. If all nodes that are to be distributed between the two sides have been split according to the same dimension, that dimension can be used to distribute said nodes.

An example can be viewed in Figure 2.4. Here you see that node A was first split into A' and B using dimension 2. Then node B was split into B' and C using dimension 5, and so on. In the end, each of the nodes will be split by all numbers on the path from that node to the root.

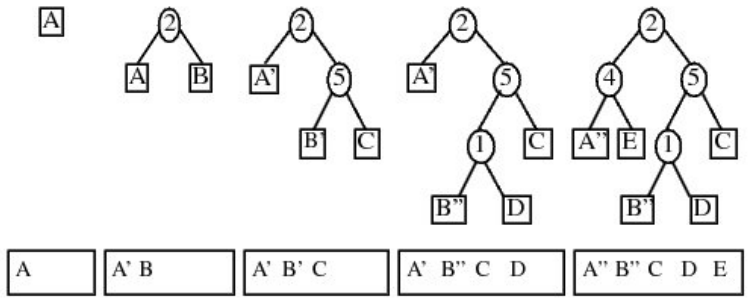


Figure 2.4: Split History for a node (*Source: [3]*)

The important consequence is that, at some point, all nodes in this subtree was split according to a value in the dimension indexed by the root of the split history. For all other dimensions there is likely to be a node that's never been split in that dimension. As a consequence, it will span the entire length of the hypercube along that dimension. Therefore, this split procedure only evaluates dimensions in which all nodes have been split. To distribute the nodes, it suffices to use the same procedure as described in [2]. That is, sort the nodes according to this dimension and evaluate the different splits.

Now, though this split will be overlap-minimal, there is no guarantee about it being balanced. The X-tree operates with another global parameter (in addition

to the max allowable overlap). This is the minimum fanout allowable from a split. Each of the sides needs to contain at least a set percentage of the nodes after the overlap-minimal split. If this does not happen, it abandons the split and creates a supernode instead.

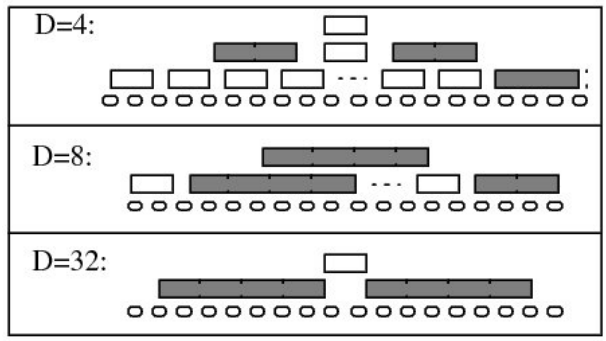


Figure 2.5: The X-tree adapts to the data it indexes (*Source: [3]*)

As an end result, the X-tree will be more linearly organized in areas of the tree where there would be more overlap. In areas where it is possible to achieve a clearer division of the data, the X-tree will be more hierarchical. Consider Figure 2.5. When the dimensionality increases, the tree will have a flatter structure, so that the overhead from processing the directory nodes will be less likely to need more resources than what is saved from pruning the search space.

2.3 Metric Indexing

Another widespread means of indexing objects for similarity searches is called Metric Indexing [19]. A metric index assumes that the distance function adheres to the metric postulates:

$$\forall x, y \in O, d(x, y) \geq 0 \tag{2.2}$$

$$\forall x, y \in O, d(x, y) = d(y, x) \tag{2.3}$$

$$\forall x, y \in O, x = y \Leftrightarrow d(x, y) = 0 \tag{2.4}$$

$$\forall x, y, z \in O, d(x, z) \leq d(x, y) + d(y, z) \tag{2.5}$$

O is here the set of objects on which the distance function operates.

The strength of metric indexing lies in its ability to utilize the metric postulates, especially equation 2.5, to prune the search space. Specifically, knowing two of the three distances referenced will be enough to provide an upper and lower bound on the last one. (2.5) gives an upper bound as it stands, while a lower bound is found by rewriting it into the following:

$$\forall x, y, z \in O, d(x, y) \geq d(x, z) - d(y, z) \tag{2.6}$$

Especially when the distance measure is expensive to calculate, this is very valuable. There is also quite a bit to gain when it comes to I/O costs, as only parts of the data needs to leave secondary memory during search. A third benefit is that since this is a generic and widely studied search technique, you'll have access to a lot of different indexes already. For these reasons it could often be a good idea to research your data to see if it might conform to a metric space.

2.3.1 Distance-based Data Indexing

In [19], three general ways of indexing data for metric searches are described. Similar to all of them is that they utilize precomputed distances to calculate distance bounds like the ones from (2.5) and (2.6).

- **Ball Partitioning** recursively divides the data into subsets based on the distances from chosen pivots. Knowledge of the distances between objects and pivots, as well as between the pivots and the query object, gives us bounds on the remaining data. This allows us to prune away whole subtrees from the hierarchical structure. An example of such a method is the *Vantage Point Tree* (as described in [19])
- In **Generalized Hyperplane Partitioning**, each level contains a set of pivots (often a pair), as opposed to one from the above mentioned ball partitioning. An object is placed into the partition corresponding to the pivot that lies closest to it. When we've calculated the distances between the query object and the pivots, we could (hopefully) disregard some of the partitions from the continued search. An example of such a technique is the *Generalized Hyperplane Tree*.
- In addition to these two, we've got the direct use of **Pivot Filtering**. This bases itself on knowing the distances to more than one pivot, and therefore hopefully getting tighter bounds than by just using one. In *LAESA*, for instance, a grid of distances between m pivots and all the objects is maintained. During search, we iterate the pivots and prune away all objects where the lower bound is higher than the highest interesting distance. Thus saving quite a few distance computations.

More advanced metric indexes often utilize more than one of these techniques, as well as others (some of which are also described in [19]).

2.3.2 The Curse of Dimensionality

Though not directly using the dimensions of the data other than for computing distances, metric indexes does not really avoid the curse of dimensionality. In [5] some results related to this is presented. Specifically, it observes the behavior of the distance function as the dimensionality increases.

When more features are added, you also add more noise to the feature vectors. Assuming new dimensions generally affect the distance as much as the previous, and also that they likely increase total distance, [5] presents the following result:

$$\lim_{m\to\infty} |\frac{D_{max}}{D_{min}} - 1| = 0 \tag{2.7}$$

where m is the amount of dimensions. That is, all distances converges towards the same number as dimensionality increase. Due to this, dividing the hyperspace based on inter object distances might become hard with more dimensions.

Though metric indexing won't avoid the curse of dimensionality altogether, it is not without its advantages compared to a spatial approach. First of all distance based methods do not suffer as much when the true structure of the indexed data is intrinsically much simpler than the amount of dimensions would suggest. The same thing applies if the amount of values along one dimension is a small discrete number [19]. The complexity of a spatial approach could easily be unnecessary for that dimension.

In addition to these things, one difference between metric and spatial indexes lies in the fact that the former will use sphere-shaped division of the data, while spatial searches like the R-tree uses hypercubes. This leads to potentially smaller volume for a subtree defined by a distance from a pivot. Lets evaluate the volumes of the hypercube relative to the hypersphere:

$$V_{cube} = (2r)^d \tag{2.8}$$

$$V_{sphere} = \frac{2r^d\pi^{d/2}}{d\Gamma(d/2)} \tag{2.9}$$

$$R_{sphere/cube} = \frac{\pi^{d/2}}{d2^{d-1}\Gamma(d/2)} \tag{2.10}$$

$R_{sphere/cube}$ represent the ratio between the sphere volume and the cube volume for a sphere inscribed in that cube (see Figure 2.6). Obviously, if this was the case, the ratio would approach zero as the dimensionality approached infinity. Thus, most of the volume of the hypercubes are located in the corners.

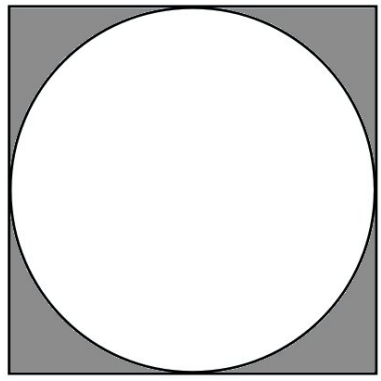


Figure 2.6: Sphere versus hypercube in 2D.

This case will only appear if none of the data objects are actually located in those corners. For uniformly distributed data this is very unlikely, so you can not draw any clear conclusions from this. For instance, if an object is located in the farmost corner, the sphere could potentially be a lot larger than the cube.

2.3.3 Metric-Tree

The Metric-tree (M-tree) [8] is the metric search equivalent of the R-tree. It is based on the ball partitioning approach to metric indexing, recursively dividing the data into subtrees based on their distance from a pivot connected to that specific node. Each internal node stores a pivot object along with the maximal distance to an object that resides in that subtree. It also contains the distance from the pivot object to its parent. An example structure of an M-tree is shown in Figure 2.7.

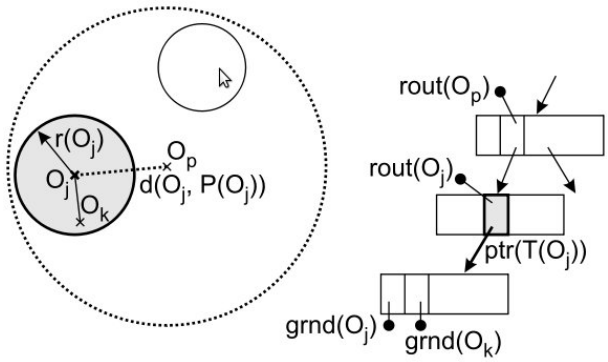


Figure 2.7: Example M-tree structure (*Source: [18]*)

When searching the tree, these distances can be used with 2.6 to create lower bounds for the distances to the subtree data objects. If you compute the distance from your query object to the pivot, you can use the lower bound shown in 2.11 for this pruning. Due to the fact that distances to parent pivots is precomputed, it might be possible to avoid distance computations altogether, however. Assuming you've already got the distance between the query object and the parent pivot (from the previous recursive step), the lower bound from equation 2.12 can be applied.

$$d(Q, O_i) \geq d(O_r, Q) - r(O_r) \tag{2.11}$$

$$d(Q, O_i) \geq |d(O_p, Q) - d(O_r, O_p)| - r(O_r) \tag{2.12}$$

In the above equations, O_p is the parent pivot, O_r is the current pivot and $r(O_i)$ is the corresponding ranges.

M-tree Building/Insertion

The original M-tree is built as the R-tree, by sequentially inserting objects into the structure. During insertion, the M-tree recursively inserts the new object into a subtree that requires no increase of covering radius to include that object. If there are several, it chooses the one with the closest pivot. If there are none, it inserts into the one requiring the least increase.

Naturally, node overflows can occur in an M-tree as well. The M-tree deals with this in the exact same way as an R-tree. It splits the node in two and replaces it in the parent tree. Due to this the M-tree, just as the R-tree, is balanced.

Node Splitting

M-tree node splitting is pretty straight forward. It starts by choosing two seed objects, one for each side of the split. In the article, they suggest several ways of doing this [8]. The common goal for each of these is two minimize the covering radii of nodes resulting from the split. Methods include trying all pairs (variants minimizing the sum or maximum of the radii), trying a random subset of pairs or simply choosing the two objects that lie furthest from eachother.

Distribution of the nodes is suggested done in one of two ways. The first is matching each of the remaining elements with the closest pivot. The other is to balance the nodes, alternating between matching the closest elements to one or the other. Experimental results indicate that the former method works best in practice.

2.3.4 Pivoting Metric-Tree

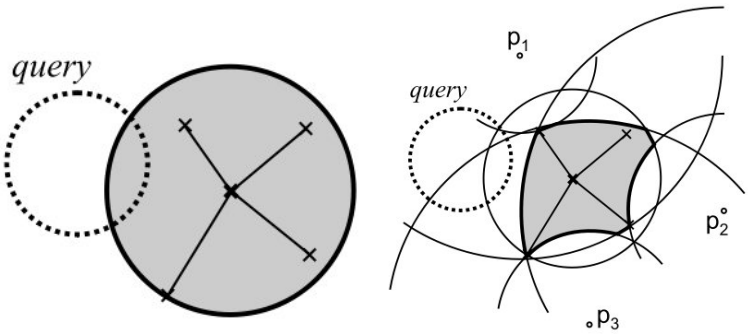
The Pivoting Metric-Tree (PM-tree) is an extension of the M-tree, combining it with a pivot filtering approach [18]. The idea is to use pivot filtering to create tighter bounds than with just a single pivot, pruning away a larger part of the search space during search. Experimental results indicate that this leads to significant improvements in CPU costs, as well as a slight increase in disk access costs.

Global Pivot Table

The main addition in the PM-tree compared to an M-tree is a global pivot table. A number of data objects are chosen as global pivots, visible to all nodes of the structure. All leaf nodes store the distance from its corresponding to each of the global pivots. In effect, you get a feature vector of size equaling the number of pivots used. Directory nodes, meanwhile store the minimum and maximum distances between the global pivots and each of the objects in that subtree. That is, they store hyperrings centered at the pivot objects in which all data objects are located.

The main difference between feature objects in the PM-tree and those in a spatial index like the R-tree (or X-tree), is that the features stored in the PM-tree inherits the properties of the distance function. The distance function is as known assumed to adhere to the metric postulates, and can therefore be used in

conjunction with the triangular inequality 2.5 to produce lower bounds. This can give tighter bounds than what is achievable with a regular M-tree, as seen in Figure 2.8.



(a) Knowing pivot distances allow us to eliminate parts of the search space in the M-tree

(b) Combining several pivots potentially lets us make tighter bounds in the PM-tree.

Figure 2.8: Pivoting in the M-tree and the PM-tree. (*Source: [18]*)

In order to keep track of this data in the PM-tree, the insertion method needs to be adjusted. Whenever a new object is inserted, it's distances to each of the global pivots is calculated once for storage in the leaf. These distances can therefore be reused in all the internal nodes in the path from the root, thus rendering any additional distance calculations (related to these pivots) unnecessary [18]. One must also take extra care of correctly setting the hyperrings of the new entries resulting from a split node.

2.4 Ptolemaic Indexing

In [13], Hetland suggests using Ptolemy's inequality as a basis for indexing. That would of course require the distance function to adhere to it. The inequality is as follows:

$$d(O_A, O_C) * d(O_B, O_D) \leq d(O_A, O_D) * d(O_B, O_C) + d(O_A, O_B) * d(O_C, O_D) \quad (2.13)$$

Here, O_i is any data object on which the distance functions operates. From now on, we'll call a space in which the distance functions adheres to Ptolemy's Inequalty (2.13) a ptolemaic space. Rewriting this equation, you can find upper and lower bounds for distances that has not yet been computed:

$$d(O_A, O_C) \leq \frac{d(O_A, O_D) * d(O_B, O_C) + d(O_A, O_B) * d(O_C, O_D)}{d(O_B, O_D)} \quad (2.14)$$

$$d(O_A, O_C) \geq \frac{d(O_A, O_D) * d(O_B, O_C) - d(O_A, O_B) * d(O_C, O_D)}{d(O_B, O_D)} \quad (2.15)$$

Due to the similar natures of ptolemaic and metric indexing, it is quite possible that already developed techniques for indexing a distance space can be enhanced by ptolemaic indexing. As the equations above require several precomputed distances to be efficient, Hetland [13] suggests using them in indexes utilizing pivot filtering. Like we see in the PM-tree, they will contain several precomputed distances.

2.4.1 Quadratic Form Distance

On family of distance functions that adheres to the ptolemaic inequality is Quadratic Form Distances [4, 13]. A quadratic form distance is a distance of the form

$$d_{QF}(\mathbf{x}, \mathbf{y}) = \sqrt{\sum_{i=1}^{N} \sum_{j=1}^{N} a_{ij}(x_i - y_i) * (x_j - y_j)} \quad (2.16)$$

Where $\mathbf{x}$ and $\mathbf{y}$ are the feature vectors on which the distance function operates, while a_{ij} are entries in a parameter matrix $\mathbf{A}$, specific to this instance of the quadratic form distance [4].

One advantage of the quadratic form distance is that its highly modifiable. The matrix A can be customized to model correlations between the different features of the vector. This makes it very applicable to similarity searches because of being able model many different similarity measures [17].

Note that the regular euclidean distance is in fact a special case of the quadratic form distance, using the identity matrix as the matrix $\mathbf{A}$ in the formula.

Chapter 3

Methodology

In this chapter, there will be given a description of the system that's been implemented for the testing. It will begin with a short overview of the whole system, and then move on to the individual components. A short justification for each part of the system will be given where relevant.

The purpose of the system is to compare the performance of a spatial index against a metric index, as well as a metric index modified for ptolemaic indexing (as mentioned in the introduction chapter). The systems are to be tested on different data dimensionalities, in order to try get an indication on how the performances changes as dimensionality increases. Also, to get a clearer view of how the ptolemaic indexing paradigm compares, the system should be able to run tests on distance functions suited for such indexing.

3.1 System Overview

The whole system has been implemented in the Java programming language. The choice of Java was heavily based on the availability of an indexing framework, NEUStore [1], described later in this chapter. This framework facilitates disk reads/writes, and thus allows testing for amount of disk accesses made by each of the indexes.

Four different similarity search indexes have been implemented, a sequential list, an X-tree, a PM-tree, and lastly a PM-tree modified for supporting ptolemaic pruning of the search space. All four indexes should support quadratic form distances, as well as the euclidean distance. The latter is a special case of the former, however, so designing based on the former suffices. In addition, a small module for generating valid tests was implemented. This generates the test data, ensures that the tests are run with valid constraints, and also records the results of the tests.

3.1.1 JUnit Testing Framework

Throughout the implementation process, the JUnit testing framework has been used to ensure the correctness of the code. JUnit testing is a means for setting up automated tests in Java. A method for setting up the system is given for each testing class. Implementing this method means that the function will be called before running each of the separate tests in that specific class.

Among the more complicated tests are those testing insertion and searching in the indexes. These generate a set amount of randomized data before building the index. After that, a series of searches are run on those indexes to verify the search results. This result is compared to the result of a linear search through the data.

In addition to these tests, there are correctness tests for the two distance measures; the euclidean and the quadratic form. These mainly verify that the methods return the correct distances.

3.2 Index Implementations

The NEUStore framework [1] forms the basis for all the implemented indexes. All the most important classes in each index are inheriting classes from NEUStore. This facilitates a disk page based approach, and thus allows us to keep track of how pages are swapped in and out of memory. Though it puts some constraints on the actual implementation, at the same time it forces a more realistic implementation. After all, most practical indexes will need to store so much data that use of secondary memory is required. The main classes of the NEUStore framework are depicted in Figure 3.1.

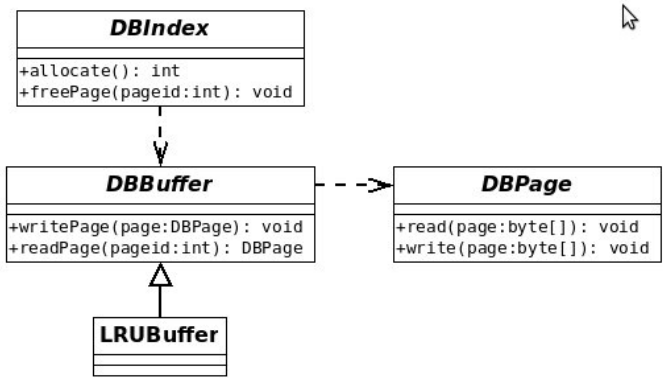


Figure 3.1: Main classes of the NEUStore framework.

The **DBIndex** class must be inherited by the actual index class. It represents the interface the index gives to the outside. The DBIndex class provide methods for allocating and freeing disk pages. Implementing classes does not need to keep track of what pages have been used or not. This is all done by the methods provided by

this class.

DBPage is the class representing a single disk page for the index. In order to be able to use the buffer and disk reading framework, it is necessary to inherit this class. That involves implementing two essential methods. The first one is `write`, which will write all the information contained in the page to disk. The other one is the corresponding `read` method. This one is supposed to read the information stored in the write method into an otherwise empty page.

In order to represent an interface for memory management, the **DBBuffer** class defines buffer methods for accessing the different pages stored. Only through the DBBuffer can the index actually access the disk pages. This is done though the `readPage` and `writePage` methods. Internally, the buffer keeps track of which pages are stored in primary memory (buffer) and which are stored on disk. Whenever a new page needs to be accessed on the disk, it also chooses which one to discard from the buffer.

The actual implementation of the DBBuffer class uses a RandomAccessFile to model the disk, so during disk accesses actual writes and reads are being done, leading to a slower performance when lots of disk accesses are being done. For the purposes of this system, the provided `LRUBuffer` class was used as a disk buffer. This buffer always discards the least recently used disk page when loading a new page into memory. It is parameterized on page size and buffer size.

In addition to these classes, the framework provides some secondary support classes. Most of these were not used directly in these implementations, however.

3.2.1 Reference Index - Linear Scan

Storing the data sequentially on disk and scanning through all during searching is called a linear index. Unless indexes outperforms a regular linear scan, the point of indexing disappears. This is the most memory efficient index there is, due to only needing to store the actual data, and other indexes need to show speed improvements in order to be worthwhile. These can of course from various factors, more on that later.

Due to this, a sequential list index was implemented on this system as reference. Each of the tests run on the other indexes will be ran on the linear index as well in order to see how their performances compare. This is especially interesting as the performance of both the spatial and metric index is expected to drop as dimensionality increases. At what number of dimensions are these surpassed by the linear scan (if ever)?

The linear scan implementation in this system is done as a linked list. Every disk page contain only a memory address pointer in addition to the actual data. The pointer points to the next page in the list. The index head only stores the address to the first page. This setup supports insertion in $O(1)$ time, always inserting into the first page in the index. If this page is full, we simply create a new page and add it to the front of the list.

Searching, both the K-Nearest Neighbor search and the range search, is done by sequentially processing all the pages in the linked list. All data objects are tested against the search criteria. In the range search, they are simply added directly to

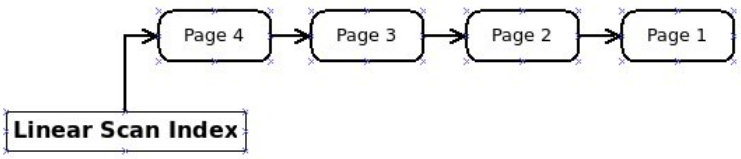


Figure 3.2: The linear scan index is implemented as a linked list.

the result set. In the k-nearest neighbor search, on the other hand, you don't know which objects satisfy the criteria until you've checked all of them. Therefore the procedure keeps a priority queue (heap) with the k best matches found so far, with the worst match on top. When a better match is found, the worst match is popped off the heap and the new one is added. When all objects are checked, the objects that are on the heap are added to the result. The performance of the range search is therefore $O(N)$, while for the k-nearest neighbor search it is $O(N * log(k))$.

Note that although the first page might not be full, this has no impact on performance. All pages need to be accessed anyway.

3.2.2 Spatial Index - X-Tree

The X-tree was chosen for the spatial index in this system. In [16], they describe it as one of the strongest spatial indexing structures for high-dimensional data. In contrast to other spatial structures for high-dimensional data, the X-tree does not make any assumptions about the data set. FastMap [10] assumes that the n-dimensional data is representable in k-dimensional space without a too large loss of data. The TV-tree, meanwhile, assumes that most of the information is on the objects are stored in a smaller subset of the dimensions. This allows it to index based only on these dimensions, leading to a larger fanout and better query performance.

The X-tree makes no such assumptions, and is a clear spatial index based on the R-tree. Other indexes mentioned in [16] (like the SSS-tree) are influenced by distance-based indexing.

Disk Page Structure

One node in the X-tree is in general represented on a single disk page, with the exception of supernodes. Luckily, with a little planning, it isn't necessary to handle supernodes all that different from regular nodes. Leaf nodes are either way relatively simple. They only store a sequence of data objects, in our case vectors of floating-point numbers. All information required in the page header is the number of objects referenced.

Internal nodes, however, are a little more complex, due to more complex references and the possibility of supernodes. To deal with supernodes, each page header stores a reference to a disk page. In all access methods for the page, this reference is checked. If it is set to a negative number, it is simply ignored. If it is set to a

positive number, however, we are dealing with a supernode. The access method then needs to access the referenced page to fetch the information stored there. That new page can in turn also refer to another page, and so on. You can thus view these supernodes as being structured as a linked list similarly to the Linear Scan index described earlier.

References to subtrees are implemented exactly as described in Chapter 2, with a minimum bounding rectangle, a memory address and a split history. The one things that perhaps need a little clarification is the implementation of the split history. As explained, the split history of a node can be viewed as a tree structure, with all performed splits in internal nodes and the actual subentries as leaf nodes. You can then notice that every node has been split by all splits in it's path to the root. An important observation to make here is that the actual tree is irrelevant. what really matters is to know along which dimensions each of the subentries have been split.

Due to this it is enough to store split dimensions along with each of the entries. Here, this is done with a bitmask. Each bit in the mask maps to a single dimension in the indexed space. If the bit is set to 1, the entry have been split along that dimension. If it is set to 0, it is not. As we will see, this simplifies finding common splits.

Algorithms

The main algorithms in the X-tree index are related to node splitting and actual searching. Both these are essential to the performance of the index. Without a decent node splitting procedure, the structure of the tree can easily become inefficient. A searching procedure is essential to traverse the tree efficiently while at the same time ensuring that all candidate objects are found.

When a node overflows during insertion, the splitting procedure is initiated. This one first tries a topological split. The split implemented here is a combination of Greene's split [11] and the split used by Beckmann and Kriegel in the R*-tree [2]. It first chooses a split axis based on the two most distance subentries. Then it measures the overlap from all splits within the minimal fanout, and chooses the best one. This procedure does however have the potential of returning a split with high overlap. If this value is higher than a global value for the maximal overlap, the splitting procedure instead calls another procedure for finding an overlap-minimal split.

The procedure for finding an overlap-minimal split is shown in Table 3.1. It is basically a brute force method, trying all splits along dimensions common to all subentry split history. Retrieving all valid split dimensions is done by AND'ing together all split histories of the subentries. The resulting bit vector will contain have bits set for all candidate dimensions (usually only one). All potential splits are evaluated in each of the candidate dimensions, and the best one is returned.

The overlap-minimal split will usually return a split that lies within the accepted limits of overlap. The one condition that might cause it to fail, however, is it being unbalanced. If the fanout of one of the suggested subtrees lies below a global limit, a supernode is created instead.

Algorithm Overlap Minimal Split
1. splits := AND all subentry split history
2. **for** i = 1 **to** number of dimensions:
3. **if** the i'th bit of splits is set:
4. sort the subentries along the i'th dimension
5. **for** j = 2 **to** number of subentries:
6. Measure overlap for split where entries 1 to j-1 are in the first part.
7. **endif**
8. **endif**
9. **endfor**
10. Return the best split found

Table 3.1: Finding the split with the least overlap based on the split histories

When it comes to searching, the most important part is to add decent pruning conditions. The searching is as simple as a recursion through the whole tree, and without pruning of subtrees it would quickly become much slower than a linear scan. We therefore avoid visiting subtrees where we can guarantee that the closest object is farther away than the search range. For euclidean distance, this is as simple as measuring the distance between the query objects and the closest possible point in the minimum bounding rectangle. The procedure for quadratic forms is more complicated, and explained in a bit. When the search reaches a leaf node, distance to each of the data objects are evaluated, and those within the search range are added to the result set.

Note also, that it is important to traverse the tree in a depth-first fashion. For k-nearest neighbor searches, the search range is updated only when visiting leaves. It is therefore important to visit leaves before considering all internal nodes, in order to prune as much as possible of the search space.

X-tree Quadratic Form Distance

In order to support the Quadratic Form distance in a spatial index, some adjustments have to be made to the searching functions. Specifically, the calculations of a lower bound on the distance from a point to a mininum bounding rectangle is no longer straightforward. This is of course essential to an index like this, as it would otherwise be far outperformed by a linear scan. It was therefore necessary to implement a way of finding this value. Recall that the quadratic form distance function is the following quadratic expression:

$$d_{QF}(\mathbf{x}, \mathbf{y}) = \sqrt{\sum_{i=1}^{N}\sum_{j=1}^{N} a_{ij}(x_i - y_i) * (x_j - y_j)} \tag{3.1}$$

Now, we first translate the coordinate system so that the query point is at origo. We can then attack finding the quadratic form distance from an MBR to a point by formulating a quadratic program:

Algorithm Quadratic Form Box Distance
1. translate coordinate system;
2. point := *closest*(MBR, origo)
3. distance := d_{QF}(point,origo)
4. **do**
5. g := $gradient_{QF}$(point)
6. g := *truncate*(MBR, point, g)
7. s := *linearMinimization*(point, g)
8. point := *closest*(MBR, point + s*g)
9. ndist := d_{QF}(point,origo)
10. **if**(*equals*(distance,ndist))**break**
11. distance := ndist
12. **until** distance < *EPSILON*
13. **return** distance

Table 3.2: Finding the minimal quadratic form distance between a minimum bounding rectangle and a query point

$$\begin{array}{lccl} \text{Minimize} & & d_{QF}(\mathbf{x}, 0) & (3.1) \\ \text{subject to} & x_i & \leq & MBR_{i,high} \\ & x_i & \geq & MBR_{i,low} \end{array}$$

Here, $MBR_{i,high}$ and $MBR_{i,low}$ are the lower and upper limit of the minimum bounding rectangle of teh box, respectively. The solution to this program will be the minimum distance between the query object and the minimum bounding rectangle.

To actually solve this program, ideas from [17] was used. The general algorithm is to use gradient descent towards the optimal solution. This involves selecting a starting point, and then iteratively moving along the descending gradient of the quadratic form function.For every new point calculated, we have to make sure not to leave the minimum bounding rectangle. Due to the rectilinearity of the MBR, this becomes slightly easier. Pseudo-code for the algorithm is given in Table 3.2.

In the algorithm, 'point' and 'g' and 'origo' are vectors (origo is the zero vector). 'distance', 'ndist' and 's' are scalars. The function *closest* returns the point in the MBR that lies closest to the point. The function *truncate* alters the gradient function to not point out of the box, while *linearMinimization* returns a scale value to make the algorithm move faster towards the minimum. Closer descriptions of these can be found in [17]. The last procedure, *gradient*, returns the gradient vector of the quadratic form function in the point. To find the gradient in a single point, we have to find the derivative of the function in that point. By writing the quadratic form function as a matrix multiplication, this becomes easier:

$$d_{QF}(\mathbf{x}, \mathbf{0}) = \mathbf{x}^T \mathbf{A} \mathbf{x} \tag{3.2}$$

$$gradient(\mathbf{x}) = d'_{QF}(\mathbf{x}, \mathbf{0}) = 2\mathbf{x}^T \mathbf{A} \tag{3.3}$$

So finding the gradient becomes as simple as multiplying the point vector with the quadratic form matrix ($\mathbf{A}$). This is done in $O(N^2)$ time. In total, this means that the algorithm runs in total $O(h * N^2)$ time, where h is the amount of times the loop is executed. Empirical results, also supported by [17], showed that this loop usually executes relatively few times. No results higher than 20 were witnessed. It is important to note, though, that this value might not be entirely exact, due to the necessary use of *EPSILON* in floating-point equality comparisons. To ensure 100% correctness in the result, care must be taken not to dismiss subtrees whose lower bound lie close to the limit value.

3.2.3 Metric Index - PM-Tree

As a metric index, the PM-tree was chosen. The M-tree family in general is a very common metric index, especially when hierarchical ones are concerned [19]. When comparing a metric index against an R-tree structure, M-trees are desirable for their fundamental similarities with them. The PM-tree specifically was chosen for being suitable for ptolemaic indexing, due to its pivot filtering approach. This is further explained in the next section.

Essential to the implementation of the PM-tree is a set of global pivots. In this implementation, this is required to be passed to the constructor of the index prior to building the index. That way we avoid problems related to having to update the entire structure as new pivots are added/chosen. There are no constraints on the amount of pivots, however a too large number of pivots can potentially lead to worse performance. All pivots are potentially evaluated towards a lower bound on distances in every node in the tree.

Disk Page Structure

The structure of a node/page in a PM-tree is slightly more complex than that of of an X-tree. The header only contains the number of subentries contained in that specific node. Then follow that amount of subtree references. Each of these contain the following elements:

$$[P_i, d(P_i, parent(P_i)), r(P_i), ptr(T(P_i)), HR_{low}, HR_{high}]$$

- P_i - A pivot object.
- $d(P_i, parent(P_i))$ - The distance from the pivot object to its parent pivot.
- $r(P_i)$ - The radius of the subtree. All objects in the entire subtree will lie within this distance of the pivot.
- $ptr(T(P_i))$ - Disk address to the subtree.

- HR_{low} - A list of lower limits on distances to the global pivots. Each object in this subtree are at least this far from the global pivot.
- HR_{high} - A similar list of upper limits on distances between global pivots and the subtree objects.

The radius and the set of distances stored in this entry constitutes the basis for which pruning the search tree will need. As for leaf nodes, they have a similar structure. The only differences are due to the subentries in leaf nodes obviously only containing a single data object. This renders the radius value useless, and the pivot object can be considered only a regular data object. In addition, the lower and upper limits on distances will be equal, so it doesn't make sense to store both of them explicitly. They are therefore combined into one list of numbers.

Algorithms

As with the other indexes, insertion and searching algorithms are essential. Insertion in a PM-tree is done, as with the X-tree, recursively. At each node, it chooses the subentry that requires the least radius enlargement, and recursively inserts the new object in the referenced subtree. If there are more than one option, it chooses the subtree with pivot closest to the inserted object. Of course this could lead to the node overflowing here as well, and node splitting is in order. This is done exactly as described in the Background chapter (Chapter 2). The insertion is a little more interesting. Due to all the distance information, there are several possibilities of pruning the tree.

Prior to starting the search through the tree, distances between the query object and all global pivots are computed and stored in a table. The search procedure then initiates the search at the root page. For each page a series of checks are done for every subentry to see if a search in that subtree is necessary. If any of the below checks fail, that specific subtree does not need to be searched. The first check is using only precomputed distances, as mentioned in chapter 2:

$$range \geq |d(pivot, parentpivot) - d(query, parentpivot)| - radius \tag{3.4}$$

For the root page this will always pass, due to parent distances being set to zero. Also note that for leaf nodes, the radius will be zero, making the bound a lot stronger in these nodes. Next the same check is performed using each of the global pivots in place of the parent pivot. All the necessary distances for this have also been calculated, due to the precomputation of the pivot-query distances. If all checks pass, it is time for calculating the distance between the query object and the pivot. In leaf nodes this equates to calculating the actual query object distances, and it can thus be compared directly with the search range and possibly added to the result set. In internal nodes, the following check is then performed:

$$range \geq d(pivot, query) - radius \tag{3.5}$$

If this check also passes, the search procedure recurses into the subtree with the last computed distance as the new parent distance.

3.2.4 Ptolemaic Index - Ptolemaic PM-Tree

One of the main reasons for choosing the Pivoting Metric-tree as the metric index is it's applicability for ptolemaic indexing. As mentioned in [13], pivot filtering is especially suitable for ptolemaic indexing. Due to there being four different objects referenced in Ptolemy's inequality, a lot more distances are required to apply this kind of pruning power compared to regular metric indexing.

In addition, the similarities between the M-tree and the R-tree families make these suitable for comparing the indexing paradigms in general. The PM-tree is therefore a natural choice. In this section we will take a little closer look at how one can apply ptolemaic indexing principles to improve the performance of the PM-tree.

Ptolemaic Search Pruning

The main addition from applying ptolemy's inequality is increased pruning power. In [13] empirical results are reported that this can provide tighter lower bounds on the distances, and thus improve search performance. The tests have only been run on leaf objects, with actual distances computed, though, using a LAESA-like index. When searching a tree structure, however, one does not always have access to all needed distances. Due to there being multiple data objects in a subtree, the best one can get is lower and upper bounds on distances between pivots and these data objects.

Luckily the PM-tree stores such bounds for all pivots. So the problem is reduced to deriving lower bounds on Ptolemy's Inequality for an entire subtree using these distances. First, lets reiterate the actual lower bound described in chapter 2.

$$d(Q,O) \geq \frac{d(Q,P_1) * d(O,P_2) - d(Q,P_2) * d(O,P_1)}{d(P_1,P_2)} \tag{3.6}$$

Here, Q represents the query object, O is a data object and P_1 and P_2 are pivots. As mentioned, for leaf nodes in the tree this can be applied directly. For internal nodes, we don't have the distances $d(O,P_1)$ and $d(O,P_2)$. Let $LB(d(A,B))$ $UB(d(A,B))$ represent a precomputed lower and upper bound on the distance between objects A and B, respectively. We can derive a new formula for bounding the distance to objects in a subtree:

$$\begin{aligned}
d(Q,O) &\geq \frac{d(Q,P_1) * d(O,P_2) - d(Q,P_2) * d(O,P_1)}{d(P_1,P_2)} \\
d(Q,O) &\geq \frac{d(Q,P_1) * LB(d(O,P_2)) - d(Q,P_2) * d(O,P_1)}{d(P_1,P_2)} \\
d(Q,O) &\geq \frac{d(Q,P_1) * LB(d(O,P_2)) - d(Q,P_2) * UB(d(O,P_1))}{d(P_1,P_2)}
\end{aligned} \tag{3.7}$$

Note that as all distances are assumed to be positive, the transitions in this derivation will be correct. With equation 3.7, you no longer need distances to the

unknown data objects in order to apply Ptolemy's inequality on calculating a lower bound on distances. A visual representation of this new bound can be viewed in Figure 3.3.

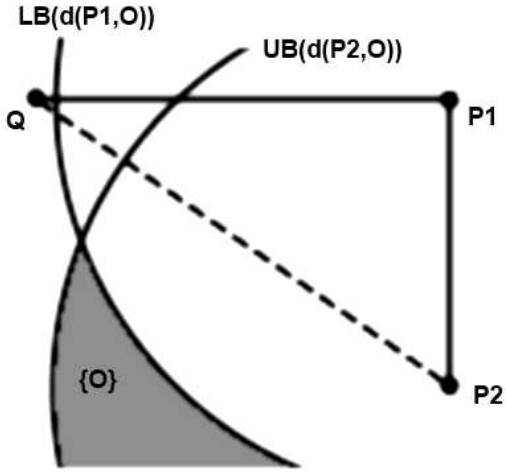


Figure 3.3: Using Ptolemy's Inequality for lower bounding distances on subtrees.

PM-tree Modifications

Incorporating these pruning conditions into the PM-tree does not require much modifications on the original PM-tree. So little, in fact, that the implementations here use the exact same classes and implementation. The constructor of the index class takes an additional boolean parameter indicating whether ptolemaic filtering is to be used or not.

The insertion procedures are identical in the ptolemaic and non-ptolemaic versions. The main change lies in the searching procedure. During check for pruning of subtrees, the simple algorithm described in Table 3.3 is added. In short, it simply iterates through all pairs of objects in the global pivot table, and applies the pruning condition using them. Due to being $O(N^2)$, this might lead to worse results when the numbers of global pivots increase. A simple modification to avoid this could be to constrain the amount of checks to a number, and choose this amount of random pairs from the table.

One more addition to the index is needed. Notice that the denominator in the equations conisists of the distance between the two pivots used. These distances are not originally stored in a PM-tree. We therefore add computation of a $N \times N$ matrix (where N is the number of global pivots) of pivot-pivot distances when creating the index. This operation is only done once, and will not affect the actual running time of index operations after this.

Regarding choosing what pivots to use in the construction of the index, [13] discusses pros and cons. While you would want to maximize the numerator in

Algorithm Ptolemaic Search Pruning
1. prune := false
2. **for** each pivota in global pivots:
3. **for** each pivotb in global pivots:
4. **if** equation 3.6 or 3.7 passes:
5. prune := true
6. break out of all for loops
7. **endif**
8. **endfor**
9. **endfor**

Table 3.3: Checking for the ptolemaic pruning condition. If prune is set to true, it is unnecessary to recurse the search on this subtree.

equations 3.6 and 3.7 by having one pivot close to the query and the other close to the object, this is somewhat mitigated by the fact that the result of the subtraction is divided by the distance between the pivots. For these tests, we simply choose a random set of pivots.

3.3 Testing

An important part of testing is to set up decent tests. A separate module was implemented to support quickly setting up and running tests. This module also records data from every test run in a separate file.

3.3.1 Generating Test Data

The main purpose of indexing data is to speed up search. Unless there is quite some amount of data, the actual difference between indexed and non-indexed search won't be so big. Due to this, indexes are mostly aimed at working well on vast amounts of data. It is therefore necessary to generate quite some amount of data for testing this system. The four indexes in this thesis were tested on two different kinds of data, both vectors of floating-point numbers, with varying dimensionality.

The first data set was merely a randomly generated set. Parameterized on minimum and maximum value for the actual numbers, amount of dimensions and number of objects, the data generation procedure was very simple. It creates a somewhat uniformly distributed data, unlike how much real life data probably would behave. Still, it gives an important perspective on how the indexes behave under these circumstances.

The second data set, on the other hand, is based on real data. It uses data from [14], a site managed by Eamonn Keogh, where different time series data collections are gathered to be tested on classification problems. The data collection used here is the Yoga set, consisting of 3000 different time series of length 426. The Yoga set contains two different classes.

When generating index data from these time series, a sliding window approach is used. For each of the time series, a sliding window of length equal to the amount of dimensions the data set should have is passed over it. At each index, the numbers in the sliding window is used as a data vector and added to the actual data set. This is repeated for the following time series until the desired amount of data objects is reached.

This method is likely to generate a slightly more clustered data set, due to the original data being meant for classification problems. It is likely that this data more closely represent a real-life situation than the entirely synthetic data does. Due to there being a natural correlation between adjacent numbers, quadratic form distances are naturally more interesting on such data than on purely synthetic data.

3.3.2 Quadratic Form Matrix

It is important that the function represented by the quadratic form matrix adheres to the metric postulates mentioned in Chapter 2. If not, that could potentially lead to some unexpected results, as the indexing would no longer be valid. Luckily, this problem reduces to making sure the matrix is semipositive definite (nonnegative definite) [17]. That is, independently of the vectors passed to the function, the result is nonnegative.

In order to make sure that the matrix is indeed semipositive definite, we look at a property of such a matrix. According to [6], a matrix being semi-positive definite is equivalent with the determinant of all the following matrices being nonnegative:

- The upper left 1×1 corner submatrix
- The upper left 2×2 corner submatrix
- ...
- The upper left $(N-1) \times (N-1)$ corner submatrix
- The entire matrix itself

In short, all the submatrices starting in the upper left corner of the original matrix has to have a non-negative determinant.

The determinant of a matrix can be found by Gauss Jordan Elimination. When the matrix has been diagonalized, it is simply a matter of multiplying all the numbers along the diagonal. A simple function for verifying that the quadratic form matrix is therefore to iterate through all submatrices, compute their determinants, and return true if all of them are greater than or equal to zero.

For the tests of this system, finding valid matrices was done by generating different matrices and simply verifying that they were semipositive definite using this procedure. More on this in Appendix A.

3.3.3 Performance Testing

Each index is tested for performance on insertions, range searches and nearest-neighbor searches. For each specific test, the four indexes are tested on the exact same set of data objects, to avoid any discrepancies from 'easier' data sets. The performance testing module was designed to record the following results for the different operations:

- CPU Time - Actual time used
- Disk accesses - Amount times a page was loaded into buffer from disk.
- Distance Calculations - Amount of times the distance calculation was called.

Disk Accesses are kept track of by the buffer class from NEUStore. Whenever it needs to write/load pages from disk, the counter is updated. Distance calculations, meanwhile, are kept track of by the distance classes. Both actual distance computations and the computations of lower bounds for minimum bounding rectangles count towards this goal. For the quadratic form distance in the X-tree, every call to the $O(N^2)$ methods (distance and gradient) increases the counter. The last result, CPU time, is simply recorded from amount of system time used by the whole process. Before the start of the test, system time reported from `System.currentTimeMillis()` is stored. When the test is finished, the difference between the start time and the current time is recorded.

When the tests have been run, the results are written to a text file.

Chapter 4

Results and Discussion

4.1 Test Setup

When testing the search operations, it was observed that the range search and the k-nearest neighbor searches gave very comparable searches. The main difference was that depending on the data set, distance and range used, the range search would be very variable in amount of results returned. It therefore also had more variable performances.

In a real life index the range is more likely to fit the data set and distance, and thus give more reliable performance. When testing searching here, it was therefore used a k-nearest neighbor search instead. K was set to 40.

All tests were ran several times (50), and the average was used in the result.

Testing Parameters

Both the data sets were generated to contain 500000 (five hundred thousand) elements. There was a trade-off between having the tests run in reasonable time and having a large enough set to be indexed.

In order to get reasonable results from the tests, buffer and page sizes were chosen with several factors in mind. First of all, the buffer shouldn't in general be able to contain the whole index. At the same time, it is important to allow enough disk pages in the buffer for the indexes to keep important pages in the buffer even when searching other pages. For the page size, this was also constrained by not having the whole index in the buffer. To avoid getting a too low fanout, it was important to keep it at some size. In the end, after some testing, the page size was set to 8192 bytes (8kB), while the buffer size could contain a total of 2048 pages. This models a main memory size of 16MB. Though most indexes will have more to play with in real life, this is offset here by a smaller sized data set.

Quadratic form matrices were generated as described in the previous chapter. More on this can be found in Appendix A.

X-tree Parameters

According to [3], you can find good parameters for the maximum allowable overlap between two nodes from inspecting the times required for disk page access (T_{IO}), time to transfer a page into main memory (T_{Tr}) and time needed for processing a block (T_{CPU}). A good estimation for a value of the limit can be found by using the formula below.

$$MaxO = \frac{T_{Tr} + T_{CPU}}{T_{IO} + T_{Tr} + T_{CPU}} \tag{4.1}$$

From their results, they recommend using a value around 0.20. That is, if the nodes overlap by more than 20 percent, the topological split is rejected. This value is being used in these tests as well.

The same article suggests a value around 0.35 for minimum fanout. That is, for a split to be accepted, each part of the split needs to contain at least 35 percent of the subentries.

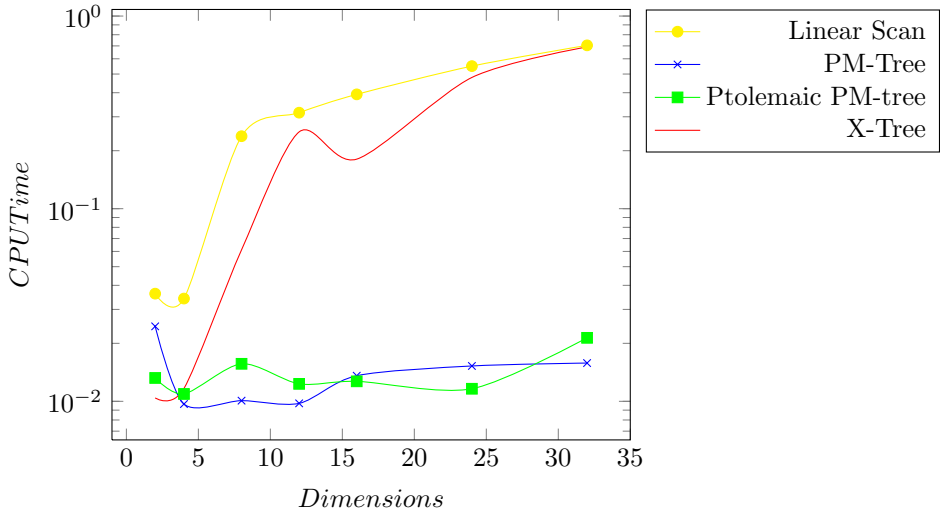
PM-tree Parameters

The one crucial parameter to the PM-tree is the list of global pivots. For these tests, 10 pivots were used. These were randomly generated with a value between -100 and +100 for each dimension.

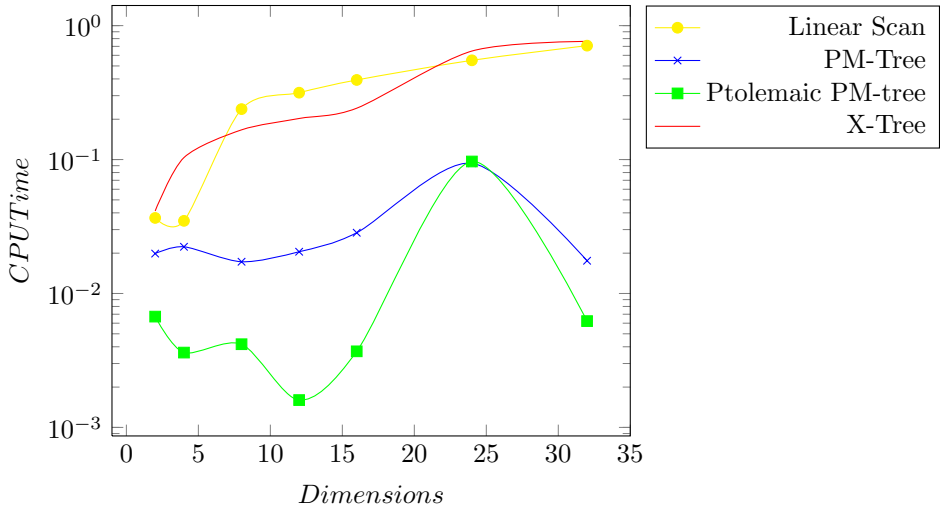
4.2 Searching Performance

4.2.1 CPU Time

Synthetic Dataset

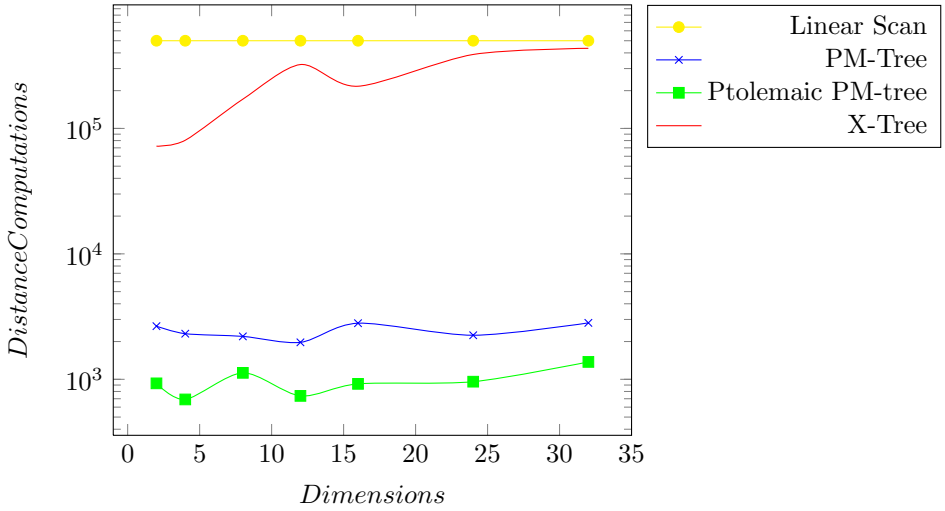


Yoga Dataset

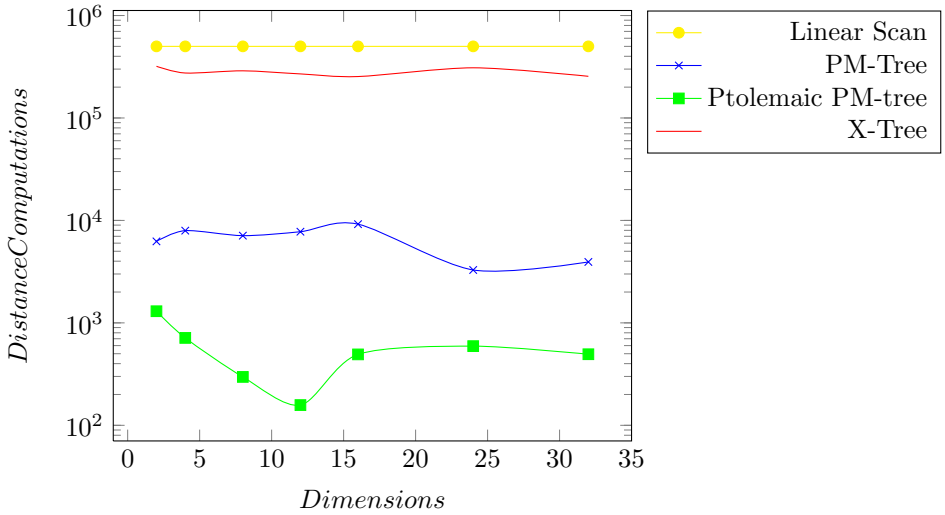


4.2.2 Distance Computations

Synthetic Dataset

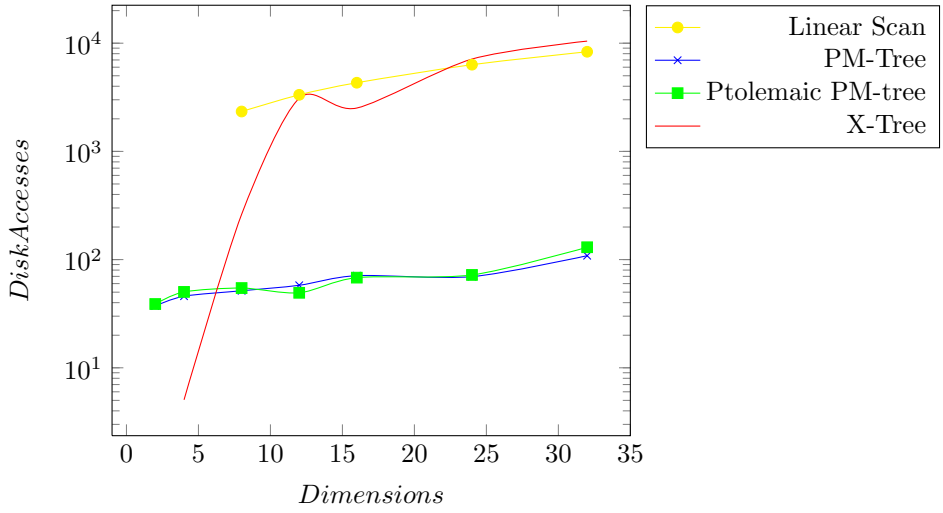


Yoga Dataset

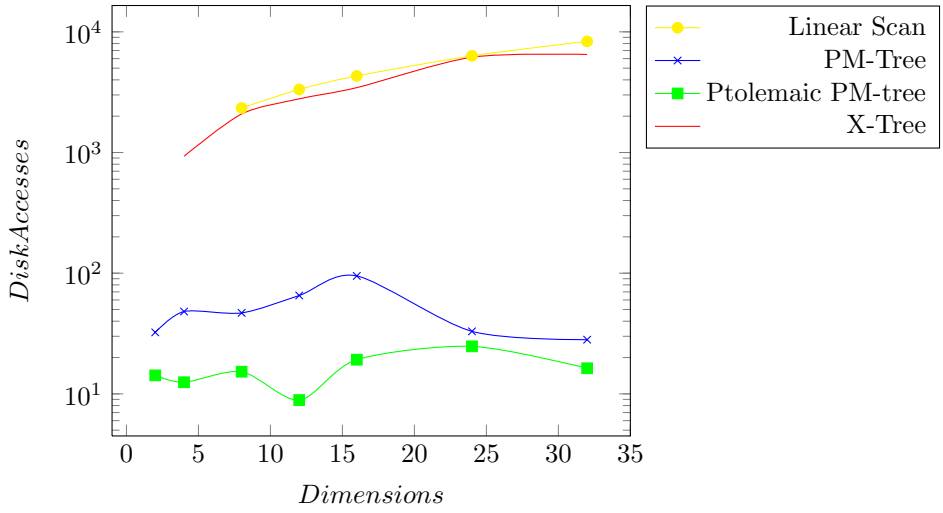


4.2.3 Disk Accesses

Synthetic Dataset



Yoga Dataset



4.2.4 Observations

The most obvious observation to make on this data is that the X-tree performs surprisingly poor. For these tests, the X-tree does not show any significant improvement over a sequential/linear scan for dimensionalities 12 and above. For lower dimensionalities on the synthetic data set, it shows results of a similar order of magnitude as the distance-based indexes.

One explanation of this could be related to the computation of lower bound distances between query objects and a minimum bounding rectangle requires multiple actual distance computations when dealing with quadratic form distances. If few subtrees are actually pruned based on this, it is really a waste of resources. Judging from the amount of disk accesses, this is the case here.

Another possibility is that the amount of data is too small for the X-tree to really shine. It could well be that with a larger data set, the fragmentation of the search space would increase, and a larger scale pruning of the search tree becomes possible. If, for each distance call, more disk accesses would be saved, the total performance would of course improve.

To the X-trees defense, the actual index structure is unaffected by the quadratic form matrix. It is therefore possible to search using different distance measures in one and the same structure. This can be exploited by allowing for custom distances, for example through user adaptable searches [17]. In a distance-based index, this is not possible. All distances could potentially change, and the index structure will have to be rebuilt to support a new quadratic form distance.

On the other end, the distance based indexes show surprisingly strong and stable performance. While the X-tree slightly outperforms them on very low dimensionalities, they don't seem very affected by the added complexity of more dimensions. This is in contradiction with some of the information studied previously, showing that a linear scan will overtake other indexes for high-dimensional data. According to [5], this effect could happen as early as with 10-15 dimensions. Unless it strikes very suddenly, these results would be an indication that the effect does not appear until the dimensionality is a bit larger than this.

Now, for the Yoga dataset, this can probably be explained. As the data has been generated using a sliding window on time series, there will likely be a strong correlation between 'adjacent' dimensions. This can probably have made the data intrinsically simpler than what the dimensionality should indicate. As a distance-based index is only indirectly affected by the dimensionalities, this is an advantage for these.

For the Synthetic data set, an explanation is harder to see. The data is likely somewhat uniformly distributed, and it is possible that this somewhat eases the indexing for a distance-based index. The results are still at least a strong indication that the PM-tree and Ptolemaic PM-tree can handle increasing dimensionality quite well.

Regarding the second research question stated in the introduction, there are also some interesting observations to be made. How does the ptolemaic indexing technique affect the performances of the distance based index?

We can see that the ptolemaic version consistently use less distance computations than the regular PM-tree. This is not surprising, as [13] has previously reported such results. However, in terms of CPU time and disk accesses, this had little effect on the synthetic data set. In fact, the results were almost identical on these runs.

As the amount of disk accesses on this set are just as large for the ptolemaic version as for the non-ptolemaic version, it is very likely that most of the savings in distance accesses was saved on leaf nodes. Pruning at leaf level will only save one distance calculation, so this can affect the performance. If a design goal for the index had been to optimize CPU time, one should take a closer look at the amount of ptolemaic checks done at leaf level.

Assuming the marginal increase in probability of pruning using the ptolemaic inequality is strictly decreasing, one can evaluate how many checks it is viable to make without letting it affect the cpu time. We derive the following formula:

$$|ptolemaicchecks| \leq \frac{cost(distance) * P_{prune}}{cost(ptolemaiccheck)} \tag{4.2}$$

Here, $cost(distance)$ and $cost(ptolemaiccheck)$ is the cost of performing a distance calculation and doing a check for ptolemaic filtering with two pivots. P_prune is the probability of pruning an element using $|ptolemaicchecks|$ checks for pruning. If this equation does not hold for the index setup, one will actually lose time because of the ptolemaic pruning, despite being able to avoid distance computations.

Sadly, measuring P_{prune} for a given amount of checks is not an easy task. One can still use this as a guideline for how the index will act, though. In these tests, for instance, it is not unlikely that with up to 100 ptolemaic checks, the gain from extra pruning is lost when leaf nodes are concerned. At least for low dimensionalities, this is intuitively true.

On the other data set, Yoga, the case is different, however. Here a reduced amount of disk accesses is observed. This means that the search function is able to use the ptolemaic inequality to prune away more subtrees than would otherwise have been pruned. This naturally also leads to an improvement in terms of CPU time, as the procedure processes less data in total.

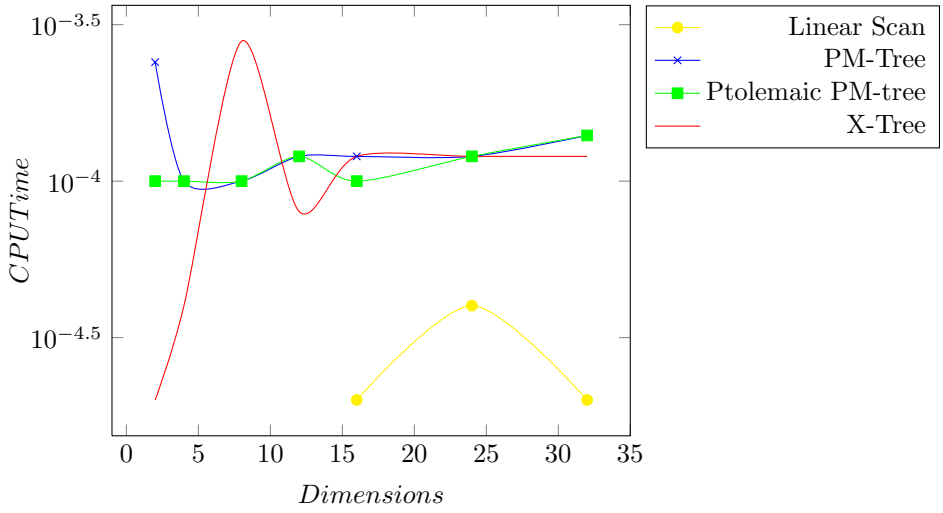
A similar line of thought as the one above can also be followed for internal tree nodes, but the calculations become slightly more complicated here. The amount of operations saved will be larger, so it could quite well be that more checks are beneficial. At the same time the probability of successfully pruning will decrease, due to lower bounds in general.

As a final note, the performance of the linear scan for the lowest dimensionalities is affected by the fact that it's compact memory representation allowed it to fit the whole index in the buffer (each data object requires very few bytes to store).

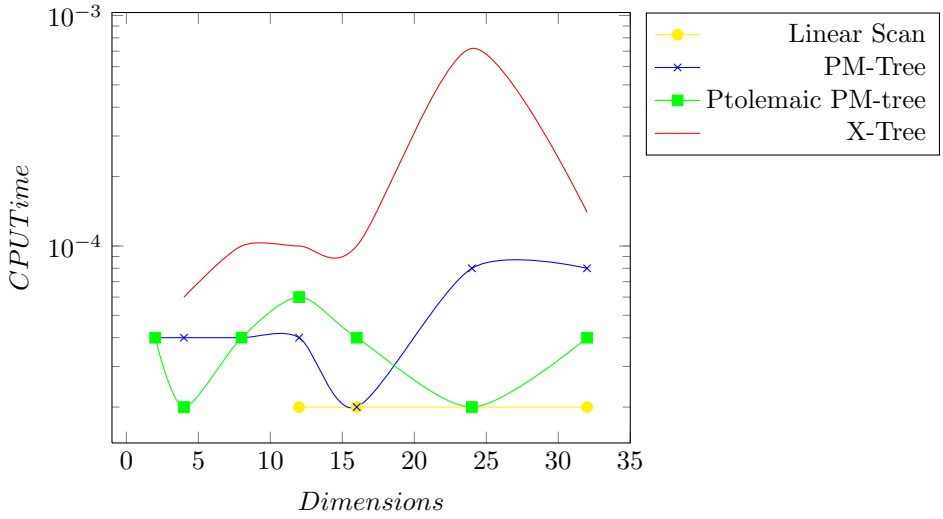
4.3 Insertion Performance

4.3.1 CPU Time

Synthetic Dataset

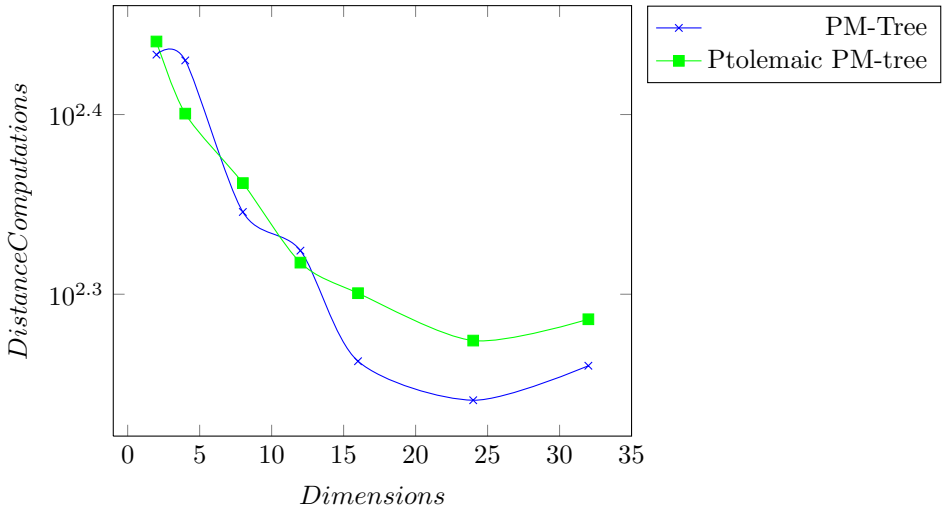


Yoga Dataset

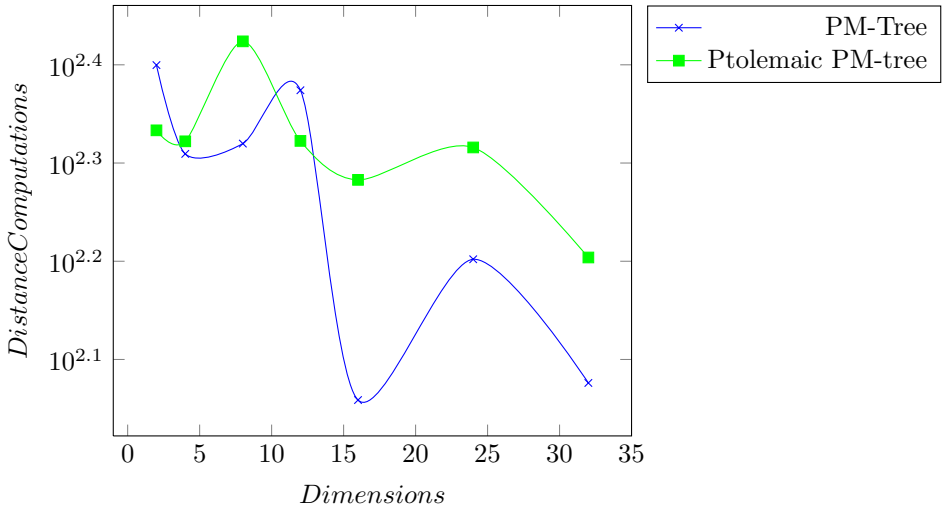


4.3.2 Distance Computations

Synthetic Dataset

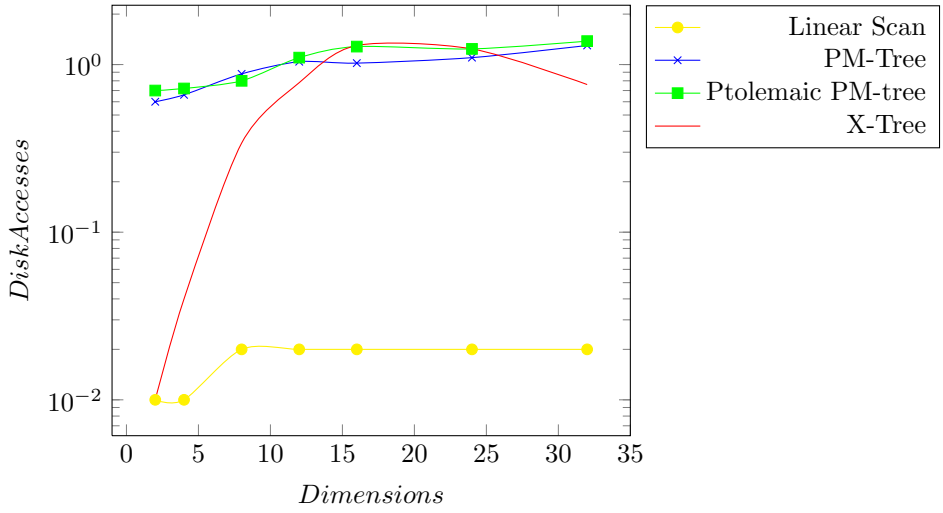


Yoga Dataset

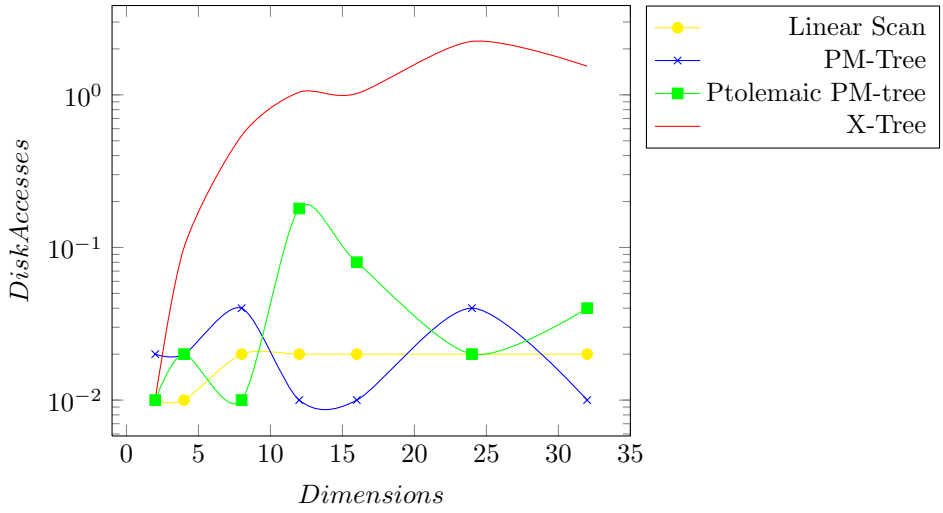


4.3.3 Disk Accesses

Synthetic Dataset



Yoga Dataset



4.3.4 Observations

The insertion results are in general not much different between the indexes here. The exception is of course the linear scan, which as we know will insert a new object in $O(1)$ time with at most 1 disk access. the other indexes are very similar in their insertion procedures, where mostly only the splitting procedures will differ. Either way, the insertion performance shown here is pretty strong. The indexes had on average one disk access per insertion. The CPU time performance mirrors this effect nicely.

It's not gold all that glimmers, however. During testing, both the X-tree and PM-tree showed very varying performance on insertions across the entire index construction procedure. While most insertions where executed in a flash, there were periods of significant slow-down as well. Especially for the PM-tree (and thus for the Ptolemaic PM-tree as well, since these share insertion procedure), this was very clear.

This can likely be attributed to the way these insertion procedures work. In faster periods, the object is recursively inserted into a leaf node, and that's that. When the tree starts filling up, however, node splits start occurring. If it gets necessary to split far up the tree, for instance all the way to the root, performance should be expected to decrease dramatically. Amortized, the times are probably still pretty good, but one should be aware of poor worst case performances for indexes like this.

The X-tree did not seem as slow as the PM-tree in this procedure. This can probably be attributed to the supernodes existing in an X-tree. Supernodes are never split. When a subnode of a supernode is split, the supernode instead grows by another entry. If that would overflow the last page of the node, another page is added to the supernode. This effectively stops the splitting procedure from travelling the longest paths in the tree, and can potentially 'save' us from the worst case performance witnessed in the PM-tree.

Regarding distance computations, we keep in mind that neither the sequential scan nor the X-tree uses any during insertion. As the two versions of the PM-tree use the same insertion procedure, we really only have one index to evaluate here.

The one interesting thing to note when looking at the amount of distance computations made here, is that it is actually a tad larger for the lower dimensionalities. Due to the page entries requiring less space when there are fewer dimensions, the fanout of the nodes is larger. Unless the height of the tree also is lower, this leads to more distances being evaluated in each level. Therefore, a slightly larger amount of computations is to be expected (or at least not come as a surprise).

Chapter 5

Conclusions

This report have had a closer look at similarity search for high-dimensional data. The search paradigms spatial and distance based indexing have been studied. The goal has been to see how they compare when used on high-dimensional data subject to quadratic form distances.

Using NEUStore, as a framework, one of each index was implemented in the Java programming language. The eXtended Node tree (X-tree) was chosen as a spatial index, as it's been specifically designed with high-dimensional data in mind. As a distance based index, the Pivoting Metric tree (PM-tree) was chosen. The PM-tree was shown to be especially suited for ptolemaic indexing, a new form of search space pruning introduced in [13].

The implementation of the PM-tree was then used as a basis for a ptolemaic index. This is basically a metric (distance based) index which also incorporates ptolemaic search filtering. Only a few small changes was necessary in order to create this ptolemaic index. Finally, a sequential list was implemented as a reference index.

These indexes was then tested for distance computations, disk accesses and CPU time needed in order to try finding answers to a couple of research questions stated in the introduction of this report. Lets have a closer look at these now.

Spatial versus Distance Based Indexing

> *How does spatial indexing compare to distance based on quadratic form distances as the dimensionality of the data increases?*

Though some results pointing towards one end of the scale was seen, the question is too broad to be able to make any clear statements about it.

The results seen through the testing in this report clearly show that the spatial index struggles as the dimensionality increase. More so than the distance based index does, which in fact continues to perform relatively well even as the dimensionality passes 30. This despite some previous research reporting that sequential scans can start performing better than indexes like this already at 10-15 dimensions.

While there are many factors in play, such as test data, buffer size, page size, implementation specifics and so on, this at least does not write off distance based methods for vector data. Despite spatial methods being more specialized in terms of this type of data, the metric and ptolemaic indexes were superior in these tests when the dimensionality passed 10.

When indexing high-dimensional data, it could thus be a very good idea to at least consider using a distance based method instead of a spatial one.

Ptolemaic Indexing

> *Can tree structured distance based indexes benefit from making use of the ptolemaic inequality as a means of pruning the search space?*

On this question, I feel confident in stating that this report have provided a little more insight. Ptolemaic Indexing is most definitely a promising development in terms of increased pruning power. A large decrease in total distance computations was observed, as well as improvements when it came to disk accesses and CPU time.

This was already known for distances directly associated with an object. Through these tests, we've seen that this also holds true when considering larger sets of objects, as long as we have lower and upper bounds on the distance between these objects and at least two pivot objects. The potential is definitely there. Further studies can perhaps give more insight into exactly how powerful it is.

Another advantage of ptolemaic indexing is that it does not need a fundamentally different structure than other distance based methods. It therefore has the possibility of being used and tested in a series of different index structures.

Future Work

As the answer to the first research question is far from settled, this is still an area where more work is needed to be able to make better judgment on whether or not a distance based index is better than a spatial one in a specific setting. One research area left open is to look at what criteria specifically makes a distance based better. If one could find better decision factors, that could potentially make life a lot easier for those who are about to choose an indexing method for their data.

Another interesting area of research will be on designing a distance based index structure for high-dimensional data, just as the X-tree have been in the spatial indexing family. Also for distance based indexing, one will in some cases have to expect increased overlap between nodes. It could be interesting to try and combine the strength of the (ptolemaic) PM-tree with the use of the X-tree supernodes. Perhaps the overall performance could increase even more.

Other than this, the whole field of ptolemaic indexing lies open to further research. Examples include choosing a better set of pivots, finding a better way of choosing which pivots to use for filtering the search space, adjusting the amount of ptolemaic checks against the expected gain from pruning that specific node, and so on. All in all, there are still a lot of things that need closer study. Ptolemaic indexing is an interesting field that is likely to see more progress in time to come.

Appendix A

QF Matrices

Here are the quadratic form matrices used for the lower dimensionalities. Due to size considerabilities, the rest of the matrices were omitted here. The numbers in each matrix have been rounded to three decimals.

Generating Procedure

For small N, where N is the amount of dimensions, it is quite possible to simply generate entirely random matrices and then test whether they are non-negative definite. As N grows larger, the probability of finding such a matrix converges towards zero, however, and some tweaks are in order.

First of all, we can use the property that for all non-negative definite matrices, the following equation holds [6]:

$$|m_{ij}| \geq \sqrt{m_{ii} * m_{jj}} \tag{A.1}$$

Here m_{ij} are the numbers in the non-negative definite matrix M. By first choosing the numbers along the diagonal, we can add this constraint to the other numbers generated. This significantly improves the amount of successful generations.

One can then note that because of the submatrices in the upper-left corner needs to have a non-negative determinant, one can iteratively increase the matrix with one row and column. Randomize the last row and column until the determinant of the whole matrix becomes non-negative. This speeds up the generation significantly, but you can still walk into a corner where it will be near impossible to increase the size of the matrix with non-zero numbers. It is therefore a good idea to start over after trying a large amount of times (10000 in the implementation used here).

Still, this doesn't work when the matrix grows very large (24 dimensions and above). For this many dimensions, an additional probability of setting the number to zero was added to all positions except for along the diagonal. As the diagonal numbers (assumed to be positive) will always contribute positively towards the determinants (and the distance), this increases the chance of success dramatically, and generating large matrices was no longer a problem.

2D

$$\begin{vmatrix} 4.869 & 0.442 \\ 0.442 & 0.484 \end{vmatrix}$$

4D

$$\begin{vmatrix} 0.056 & -0.210 & -0.197 & -0.193 \\ -0.210 & 0.954 & -1.292 & -1.718 \\ -0.197 & -1.292 & 3.114 & 1.957 \\ -0.193 & -1.718 & 1.957 & 4.169 \end{vmatrix}$$

6D

$$\begin{vmatrix} 4.956 & -0.955 & 0.469 & -0.381 & 1.030 & -0.442 \\ -0.955 & 1.878 & -1.356 & 0.514 & -1.258 & 0.032 \\ 0.469 & -1.356 & 3.712 & -1.134 & -0.683 & 1.304 \\ -0.381 & 0.514 & -1.134 & 2.063 & 0.779 & -0.477 \\ 1.030 & -1.258 & -0.683 & 0.779 & 4.605 & -3.194 \\ -0.442 & 0.032 & 1.304 & -0.477 & -3.194 & 4.887 \end{vmatrix}$$

8D

$$\begin{vmatrix} 1.154 & -1.431 & 0.202 & 0.083 & 0.675 & 0.133 & 0.066 & 0.384 \\ -1.431 & 3.641 & -0.365 & 0.271 & -1.250 & -0.263 & -0.345 & 0.084 \\ 0.202 & -0.365 & 2.890 & -0.298 & 0.096 & 0.220 & -0.042 & -0.223 \\ 0.083 & 0.271 & -0.298 & 2.248 & -0.369 & -0.080 & -0.165 & -0.398 \\ 0.675 & -1.250 & 0.096 & -0.369 & 1.098 & 0.081 & 0.245 & 0.210 \\ 0.133 & -0.263 & 0.220 & -0.080 & 0.081 & 0.633 & -0.072 & 0.200 \\ 0.066 & -0.345 & -0.042 & -0.165 & 0.245 & -0.072 & 0.101 & -0.019 \\ 0.384 & 0.084 & -0.223 & -0.398 & 0.210 & 0.200 & -0.019 & 0.136 \end{vmatrix}$$

Bibliography

[1] Neustore - manual. http://zgking.com:8080/home/donghui/research/neustore/. Last accessed June 20th 2010.

[2] Beckmann, N., Kriegel, H.-P., Schneider, R., and Seeger, B. The r*-tree: An efficient and robust access method for points and rectangles. *Praktische Informatik, Universitaet Bremen* (1990).

[3] Berchtold, S., Keim, D., and Hans-Peter, K. An index structure for high-dimensional data. In *Proceedings of the 22nd VLDB Conference* (1996).

[4] Bernas, T., Asem, E. K., Robinson, J. P., and Rajwa, B. Quadratic form: A robust metric for quantitative comparison of flow cytometric histograms. *Cytometry Part A 73A*, 8 (2008), 715–726.

[5] Beyer, K., Goldstein, J., Ramakrishnan, R., and Shaft, U. When is 'nearest neighbor' meaningful? *Computer Science Department, University of Wisconsin-Madison* (1999).

[6] Bhatia, R. *Positive definite matrices.* Princeton Series in Applied Mathematics, 2007.

[7] Brin, S. Nearest neighbor search in large metric spaces. In *Proceedings of the 21st VLDB Conference* (1995).

[8] Ciaccia, P., Patella, M., and Zezula, P. M-tree: An efficient access method for similarity search in metric spaces. In *Proceedings of the 23rd VLDB Conference* (1997).

[9] Colossi, N. G., and Nascimento, M. A. Benchmarking access structures for high-dimensional multimedia data. Tech. rep., Department of Computing Science, University of Alberta, 1999.

[10] Faloutsos, C., and King-Ip, L. Fastmap: A fast algorithm for indexing, data-mining and visualization of traditional and multimedia datasets. In *Proceedings of the 1995 ACM SIGMOD international conference on Management of data* (1995).

[11] Greene, D. An implementation and performance analysis of spatial data access methods. In *Proceedings of the Fifth International Conference on Data Engineering* (1989).

[12] Guttman, A. R-trees: A dynamic index structure for spatial searching. In *Proceedings of ACM SIGMOD Conference* (1984).

[13] Hetland, M. L. Ptolemaic indexing. Awaiting Publishing. Accessed June 18th, 2010 at http://hetland.org/research/.

[14] Keogh, E., Xi, X., Wei, L., and Ratanamahatana, C. A. Ucr time series classification/clustering homepage. http://www.cs.ucr.edu/ eamonn/time_ series_ data/.

[15] King-Ip, L., Jagadish, H., and Faloutsos, C. The tv-tree: An index structure for high dimensional data. *The International Journal on Very Large Databases* (1994).

[16] Samet, H. *Foundations of Multidimensional and Metric Data Structures.* Morgan Kaufmann Publishers Inc., 2005.

[17] Seidl, T., and Kriegel, H.-P. Efficient user-adaptable similarity search in large multimedia databases. In *Proceedings of the 23rd VLDB Conference* (1997).

[18] Skopal, T. Pivoting m-tree: A metric access method for efficient similarity search. *Department of Computer Science, VŠB, Technical University of Ostrava* (2004).

[19] Zezula, P., Amato, G., Dohnal, V., and Blatko, M. *Similarity Search - The Metric Space Approach.* Springer Science + Business Media Inc., 2006.