Universidad Michoacana de San Nicolás de Hidalgo

División de Estudios de Posgrado de la Facultad de Ingeniería Eléctrica

EFFICIENT ACCESS METHODS FOR METRIC DATA

TESIS

Que para obtener el grado de

DOCTORADO EN CIENCIAS EN INGENIERÍA ELÉCTRICA

presenta

Luis Guillermo Ruiz Velázquez

Dr. Edgar Leonel Chávez González

Director de Tesis

Morelia, Octubre 2015

List of Publications

"Towards Self-Indexing Relational Databases."
Guillermo Ruiz, Edgar Chávez, Eric Sadit Tellez.
ENC 2013: 20–27 (IEEE Press)

"Extreme Pivots for Faster Metric Indexes."
Guillermo Ruiz, Francisco Santoyo, Edgar Chávez, Karina Figueroa, Eric Sadit Tellez.
SISAP 2013: 115–126 (ISI Conference Proceedings)

"Finding Near Neighbors Through Local Search. "
Guillermo Ruiz, Edgar Chávez, Mario Graff and Eric S. Téllez.
SISAP 2015: To appear (ISI Conference Proceedings)

"Singleton Indexes for Nearest Neighbor Search."
E. S. Tellez, G. Ruiz and E. Chávez.
Information Systems (JCR issn: 0306-4379).

Resumen

En esta tesis se presentan varios métodos nuevos para acceder a información almacenada en una computadora. Concretamente, *búsquedas en espacios métricos*. Este aparente simple problema puede volverse muy complicado dependiendo del volumen de información almacenado, el tipo de consulta realizada y la complejidad de la representación de los objetos, entre otros. Las búsquedas básicas son, dado un objeto, encontrar los elementos más parecidos a él. Este tipo de búsquedas se llaman *búsquedas de proximidad* o *búsquedas de vecino cercano* y tienen muchas aplicaciones como búsquedas en bases de datos por contenido. Los *índices* son estructuras que nos permiten responder este tipo de consultas.

El problema de búsqueda por proximidad puede dividirse en dos clases, en la primera, el conjunto de resultados coincide con el de la búsqueda exhaustiva. Los índices que corresponden a esta clase se les llama *índices métricos exactos*. Los otros ídices corresponden a la segunda clase y se les llama *índices métricos aproximados*. Para el caso exacto se presentan los índices EPT y ANNI. Estos cuentan con simples parámetros ya que se ajustan a las circunstancias de cada base de datos. En los múltiples experimentos se muestra el excelente desempeño de ambos índices en comparación con el estado del arte. Para búsquedas aproximadas se presentan los índices APG*, APG*-R y BS. Todos ellos parten de ver al problema de encontrar al vecino más cercano como un problema de optimización. Estos índices son mejoras sobre el conocido APG.

Finalmente, se hace un estudio del grafo HSP que tiene múltiples aplicaciones tanto en búsquedas de espacios métricos como en redes de sensores entre otras. Se presentan varias propiedades del grafo y de su generalización.

Palabras clave: búsquedas en espacios métricos, búsquedas de proximidad, búsquedas de vecino cercano, índices métricos, EPT, ANNI, HSP.

Abstract

This thesis presents new methods to access the information on a computer. Concretely, *searches on metric spaces*. This apparently simple problem could be very complicated depending on the amount of the information stored, the type of query, the complexity of the representation of the data, and many others. The basic searches are, given an object, find the more similar elements. This type of searches are known as *proximity searches* or *nearest neighbor searches*, and has many applications like searching multimedia databases by content. The *indexes* are structures capable of answering these type of queries.

The proximity search problem can be divided in two classes, in the first, the set of results coincide with the exhaustive search. The indexes corresponding to this class are called *exact metric indexes*. The rest of the indexes are called *approximate metric indexes* . This document presents the indexes EPT and ANNI for the exact case. They have simple parameters and both adjust to the circumstances of each database. The extensive experimentation shows their excellent performance in comparison to the state of the art. For the case of approximate searches, the indexes APG*, APG*-R, and BS are presented. All of them are the result of treating the nearest neighbor problem as an optimization problem. They are improvements over the well known APG.

The thesis concludes with a study of the HSP graph that has multiple applications in metric space searches, ad-hoc networks, and others. Various properties of the graph and its generalization are presented.

Keywords: metric space searches, proximity searches, nearest neighbor searches, metric indexes, EPT, ANNI, HSP.

Agradecimientos

Quiero agradecer a mi asesor, Dr. Edgar Chávez, por todas las atenciones, no sólo en el ámbito académico. También al Dr. Sadit Téllez por su ayuda y compromiso. A mis asesores y revisores, Dr. Juan José Flores, Dr. Félix Calderón y al Dr. Antonio Camarena. A la Universidad Michoacana y al CICESE y su comunidad.

A mi familia, amigos y mi querida esposa Coré que siempre me apoyaron cuando los necesité.

Content

List of Figures

List of Tables

List of Algorithms

List of Symbols

μ	Mean.
σ	Standard deviation.
$\log$	Logarithm of base 2.
q	Query.
nn	Nearest neighbor.
$(q, r)_d$	Range query.
r	Radius.
U	Metric Space.
d	Distance function.
S	Database.
k-nn	k-nearest neighbor.
$\mathbb{P}$	Set of pivots.
$\delta_p(x)$	$d(p, x)$.
cov	Covering radius.
n	Size of the database.
piv(u)	The pivot covering u.
$A(p)$	Associated points with the pivot p.
α	Distance from the mean to the set A(p).
X, Y	Random variables.
ℓ	Number of PG's or ANNI's.
m	Number of pivots.
$Pr(x)$	Probability of x.
$E(X)$	Expected value of X.
Ω	Set of states.
$\mathcal{N}$	Transition function of states.
$\mathcal{A}$	Minimization algorithm.
E	Set of edges.
V	Set of vertices.
$\mathbf{PP}(u, v)$	Proximity path between u and v.
$B_r(u)$	Ball of radius r and center u.

Chapter 1

Introduction

The importance of computers and electronic devices in modern life is unquestionable. They are responsible for more than half of the global economy. Computers are used in every corner of the planet with a wide range of applications. They contribute to the development of science, they manage whole factories, are in charge of traffic control and navigation, the communications are practically in-existent without them, are the key tools to develop new medications, and the list can continue practically indefinitely.

The number of computers is huge and most of them are used to generate or recollect data. The internet, the biggest collection of information, is ever growing. The size of the information has an exponential growth and this will not stop any time soon.

Storing the information is important but also the ability to access it. In a big library where the books are not organized a revision of every book is needed to find the information we want. The appropriate management of this massive information is crucial to make it useful. On all this data we want to find what we are looking for, and, most importantly, we want to do it fast.

The search for information is used on practically every computer application. Many solutions have been proposed through the years, mostly focused on finding the exact match of the searched pattern, similar to DBMS (Appendix A) where the information is structured. With the advance of the information requirements, unstructured repositories were needed. New types of data like text, images, or audio are stored and a new type of search is required.

One way to easily find information is to classify it so we know directly where to start. The

way is organized depends on the type of things we are going to search and the type of information. It is different to organize numbers than to organize texts, sounds, or videos.

The collection of objects is preprocessed to avoid the trivial exhaustive search. There are some proposed methods for the case where the data can be represented using a vectorial space. This thesis focuses on a more general case, where we only know the distance of the elements of the dataset. Proximity search consists on identifying objects from a collection that are *near* a given query. The proximity of the elements of the collection is set by a *distance function* or metric. Proximity search is a pervasive problem in computer science, from pattern recognition to textual and multimedia information retrieval, machine learning, streaming, lossless and lossy compression, biometric identification and authentication, and bioinformatics [Skopal10]. Here we only know the distance of the elements of the dataset.

Metric access methods are indexes used when the data has no coordinates or when the representational dimension is large. The most basic challenge in metric access methods is to cope with the high *intrinsic dimensionality* of the data, also known as a concentration of the distribution of the data or the *curse of dimensionality*. This has been studied and documented in several places, for example in Pestov [Pestov07, Pestov08, Pestov10a, Pestov10b], Volnyansky and Pestov [Volnyansky09], Shaft and Ramakrishnan [Shaft06], Indyk [Indyk04b], Samet [Samet06], Hjaltason and Samet [Hjaltason03a], Chavez et al. [Chavez01], Böhm et al. [Böhm01], and Skopal and Bustos [Skopal11]. It can be defined as the $\frac{\mu}{\sigma}$ where μ and σ are the mean and standard deviation of the histogram respectively. Consider for example the case where every element of the collection is at distance one from each other. This case corresponds to the *Discrete metric*. Here, the exhaustive search is inevitable. This is an extreme case of a collection with high intrinsic dimension.

To cope with the above mentioned curse of dimensionality, one possible avenue is to use an approximate or a probabilistic approach. The overall idea in these methods is to accept errors of certain type. Probabilistic approaches aim at giving correct solutions most of the time (some relevant answers can be missed), while approximate solutions give distance-based guarantees. For example, in δ-dimensional spaces it is possible to return an element within $1 + \epsilon$ times the distance to the true nearest neighbor in time $O(\delta(1 + 6\delta/\epsilon)^{\delta} \log n)$ [Arya98].

For some application domains, neither an approximate nor a probabilistic approach are

acceptable solutions. Think for example on biometric identification. In this application, neither a miss (failing to identify the nearest neighbor of an object) nor a false claim (giving an output which is not the nearest neighbor) are acceptable because both lead to a failure of the identification system. For this particular example, the only possible solution is a sequential scan over all the objects in the database. The usual way to scale such a system is by using massive parallelism. For this reason, and also because it is interesting from a theoretical perspective, the search for more competitive indexes and algorithms is relevant. However, the landscape of algorithms and discarding rules only consists of a handful of key ideas. It is possible to find books and surveys devoted to reviewing the algorithmic differences between indexes, for example Chavez et al. [Chavez01], Samet [Samet06], and Zezula et al. [Zezula06].

One sensitive issue is the absence of a complexity model capable of capturing the behavior of an index in realistic circumstances. This limitation implies that indexes would be compared experimentally. Even in this setup, there are two alternatives, the first one is to count the number of distance computations as the yardstick for index comparison. The rationale behind this choice is to consider distance computation as the leading cost operation, which in turn should allow comparing different indexes using disparate datasets. One problem with this approach is that a key factor in performance is the intrinsic dimensionality of the data; and hence the supposed independence of the dataset vanishes.

The other alternative is to use a common benchmark of datasets and queries to compare all the indexes, using as measure the total time spent on queries on average. This method has the disadvantage of being unable to compare between indexes belonging to different authors, in different computer systems, and different papers without implementing everything each time. To avoid this disadvantage, the total query time is normalized using as reference a sequential scan. While this measure still hides a number of practical issues (e.g., cache usage, and workload in a multi-user environment), it will give a better guide for practitioners.

1.1 Objective of the Thesis

Given a query element q, the goal is to find elements in the database near q. The proximity search problem consists on solving the following operations over the database: nearest neighbor

search, or simply $\mathsf{nn}_d(q)$, finds the closest item to q in the database; and range query $(q,r)_d$ which retrieves objects in the database intersecting the ball of radius r centered on q. This thesis presents new methods to find relevant information on metric databases using proximity search.

1.2 Formal Notation

Let U be the set of all valid objects (i.e. the *universe*), and d a distance function. As a distance, for any $u, v, w \in U$, d has the following properties:

- Positivity, $d(u,v) \geq 0$, $d(u,v) = 0 \iff u = v$.
- Symmetry, $d(u,v) = d(v,u)$.
- Triangle inequality, $d(u,w) + d(w,v) \geq d(u,v)$.

The pair (U,d) is called a *Metric Space*.

The thesis will be focused in a particular finite subset S of U (called the database). Given some query element $q \in U$, the goal is to find elements in S near q. The proximity search problem consists on solving the following operations over the database:

- Nearest neighbor search, or simply $\mathsf{nn}_d(q)$, finds the closest item to q in S, i.e., $\mathsf{nn}_d(q) = \arg\min_{u \in S} d(u,q)$. This operation can be extended to retrieve the set of k nearest neighbors (k-nn).
- Range query $(q,r)_d$. This query retrieves objects in S intersecting the ball of radius r centered on q, for $q \in U$ and $S \subseteq U$ i.e. $(q,r)_d = \{u \in S \mid d(q,u) \leq r\}$.

For simplicity, when the context is clear, it will be written (q,r) for $(q,r)_d$ and $\mathsf{nn}(q)$ for $\mathsf{nn}_d(q)$. Our discussion will be centered on nearest neighbor queries, nn queries can be reduced to range queries using the optimal algorithm in [Hjaltason03b]. This algorithm finds the radius r such that the query ball (q,r) only contains the nearest neighbor of q.

It is clear that $\mathsf{nn}(q)$ can be answered by computing the distance from q to all the elements in the database. However, if the database is queried multiple times, and the number of elements is large or the distance function is difficult to compute, it would be advisable to preprocess the database and build an index to have a smaller amortized time.

There are two popular approaches to metric indexing, namely *pivot based indexes* and *compact partitions*.

The pivoting scheme consists on mapping the database (a subset of a general metric space) to a vectorial space. Let $\mathbb{P} = \{p_1, p_2, \ldots, p_m\}$ be a set of elements of U called *pivots*. Each pivot p defines a function $\delta_p(x) = d(p, x)$ for every x in S. Every element $x \in S$ can be mapped to the point $[x] = (\delta_{p_1}(x), \delta_{p_2}(x), \ldots, \delta_{p_m}(x))$ in $\mathbb{R}^m$. Because of the triangle inequality, for every q in U, we have that $d(q, x) \geq |d(q, p) - d(p, x)| = |\delta_p(q) - \delta_p(x)|$ for every $p \in \mathbb{P}$. This means that if $|\delta_p(q) - \delta_p(x)| > r$ for some $r > 0$, then $d(q, x) > r$. If we have the range query (q, r), every element x such that $d(q, x) > r$ cannot be in the solution set. Then we can use the L_∞ norm on the mapped space of $\mathbb{R}^m$ to rule out elements in the search. That is, if $\|[q] - [x]\|_\infty > r$, then $d(q, x) > r$. This is called the *pivot rule*.

Note that, in order to get $[x]$, all the distances from x to every pivot should be computed. All the points $[x]$ for every $x \in S$ can be computed and store. Then, for *every* query $q \in U$, just $[q]$ is needed to compute and $\|[q] - [x]\|_\infty$ which is a simple distance function. On the set of points for which the pivot rule failed, called the *candidates*, an exhaustive search is performed. If two sets of pivots of the same cardinality are given, the set that produced less candidates is said to have more quality pivots.

The pivot rule is basic for building many indexes found in the literature. In fact, AESA [Vidal Ruiz86], the most competitive exact index under the *computed distances* complexity measure, discussed below in Section 2.1, is based on this rule. However, to achieve such unmatched performance, AESA uses a quadratic amount of memory which prevents its use in practice. The performance of a pivot-based algorithm is driven by the number of pivots used and the *quality* (i.e. the discarding power) of those pivots. It will be discussed along the thesis all the known algorithms to select good pivots. The AESA algorithm can be seen as a dynamic method of selecting the best pivots for each query.

The technique described here can be regarded as a method to find high quality pivots for query objects which follow the same distribution of the database. Obtaining good pivots automatically improves all the pivot-based methods, both in main and secondary memory. One example of a very competitive index in secondary memory is the *pivot* M-tree (or PM-tree)[Skopal04]

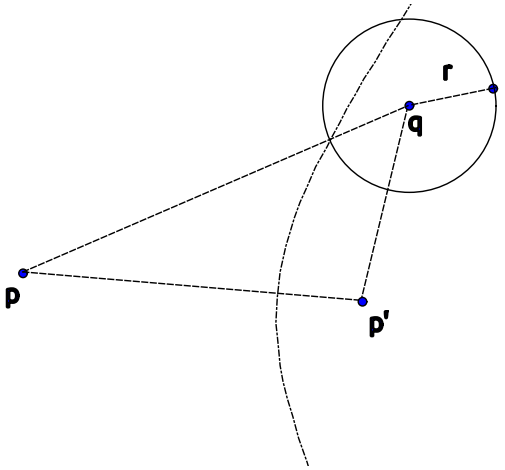


Figure 1.1: The $d(q,p) > d(q,p') + 2r$ criterion (where p' is the closest node to q) says that if the query is on the right side of the hyperbola, the points of p can be discarded.

The *compact partitioning* methods arrange the database in spatially coherent, non-overlapping regions. A set of elements, called centers, $\{c_1, c_2, \ldots, c_m\}$ is selected from S, this set defines a partition of the database such that each *cell* (an element of the partition) is formed by points that are close to a center and to each other. Most indexes of this kind are hierarchical, with a recursive rule as follows: each cell is partitioned by a new set of centers forming sub-cells; this process can be repeated on each sub-cell.

Once the space is divided into cells, we need to know which of these cells intersect the query ball. If a region does not intersect the query ball, then we can discard all its elements. If a region does intersect, the we need to look for elements that might appear on the final result. Usually, this is done by an exhaustive search on the elements of the cells. So, it is important to know which regions intersect the query ball. On general metric spaces, this is not easy but there are two main criteria: the hyperplanes and the covering radius.

The known *hyperplane* criterion is as follows. If q is the query and p' is the center of the cell where q belongs, then we can discard a cell with center p when $d(q,p') < d(q,p) - 2r$. On the euclidean plane, this inequality is defined by a hyperbola with focus points p and p', see Figure 1.1.

The *covering radius* of a cell C with center p is defined by $\mathsf{cov}(C) = \max_{u \in C} d(p,u)$. For simplicity, $\mathsf{cov}(p) = \mathsf{cov}(C)$ will be set. The covering radius can be stored for each cell. A query $(q,r)_d$ is solved by exhaustive searching all the cells with center p such that $|d(q,p) -$

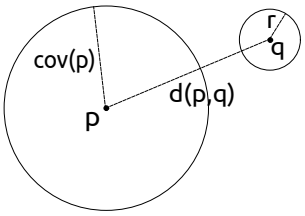


Figure 1.2: If the covering ball of a node p does not intersect the query ball then all points of this node can be discarded.

$\mathsf{cov}(p)| \leq r$ see Figure 1.2.

The complexity model used in metric indexes splits the cost in two main terms: the internal and the external costs. The former refers to the cost of filtering the candidates relevant to a given query, while the later is the cost of the verification of the candidates and is proportional to its size. This is described in the below equation

$$\text{cost} = \text{internal cost} + \text{external cost} \tag{1.1}$$

More filtering implies, on the one hand, a smaller candidate set and a corresponding lower external cost. On the other hand, more filtering also implies a more precise cut point increasing the internal cost. This trade-off is illustrated in Figure 1.3. While the external cost is a monotonically decreasing function, the internal cost is a monotonically increasing function.

1.3 Contributions

The first presented technique is called *Extreme Pivots* (EP), which produces fast indexes with a small memory footprint. The contribution provides a formal method to both prove we can obtain good pivots and give an algorithm to find those good pivots. Our only assumption is very reasonable, we assume our query set has the same distribution of the database. It is also proved that, in practice, this approach is competitive when compared with other exact metric access methods found in the literature. The best performing indexes were selected under a broad number of criteria and compared against the proposed approach with excellent results.

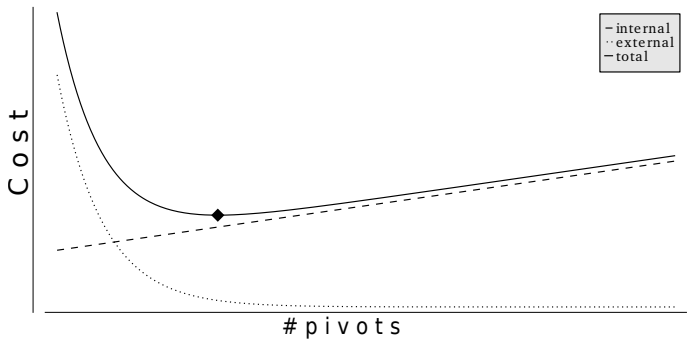


Figure 1.3: Behavior of the cost function. Internal and external costs are monotonically, increasing and decreasing respectively. The total cost is the sum of both. The diamond shows the location of the minimum total cost.

Next, the ANNI family of indexes is presented. It can be seen as a meta-heuristic to automatize the process of building good quality indexes. The quality can be measured in therms of searching time or in the number of distances computed. As an application, various indexes are presented built around this technique. An extensive set of experiments proved the efficiency of the indexes.

Next, in Chapter 4 it is discussed the approximation problem as an optimization problem. This new approach opens the doors to a new series of indexes. Tree new indexes are presented, which are the *APG**, *APG*-R*, and the *BS* that use optimizations techniques to improve the well known *APG* index. It is shown how the APG is improved with properties like: less parameters, easiest configuration, better recall, more speed on the searches, less distance computations, between others.

Finally, the Chapter 5 shows an analysis of the multipurpose *HSP*. The HSP is an algorithm to build a graph given some points in a metric space. The graph has some nice properties but the most important is the ability to rout the path from one point to another. A question about the length of the path between any two points in the HSP is answered, at least in part.

Chapter 2

Proximity Search

2.1 Related Work

For a long time, a popular complexity model has been the number of distance computations to answer a query. The rationale behind this measure is that distances are the most expensive operation in a query. However, when measuring the total search time of a good index under the above complexity measure, it could be comparable to a brute force sequential scan. This behavior is due to a combination of a simple distance function and an expensive indexing structure.

The AESA [Vidal Ruiz86] index consists on computing all the distances between the objects in the database and storing them in a table. For querying, an initial random pivot is selected and using the triangle inequality all the non-relevant objects are filtered. From the remaining objects the next pivot is selected *close* to the query using some cheaply computed distance. This process is repeated iteratively until only relevant objects remain in the collection. Those remaining objects will be the answer to the query. This method stores $O(n^2)$ distances, and hence the construction cost is of the same order. At query time it performs a constant number of distance computations for a fixed database. However, it is necessary to compute a linear number of arithmetic and logical operations. A linear restriction of the same idea is presented in LAESA [Micó94], where a constant number of pivots are used, independently of the size of the database.

Chavez et al. [Chavez01], proved that any pivot based metric index requires at least a logarithmic (on the database size) number of pivots (randomly selected from the database); however,

the base of the logarithm depends on the intrinsic dimension of the database, needing larger indexes as the intrinsic dimension increases. When the dataset has a high intrinsic dimensionality, the optimal number of pivots may not fit in main memory; hence, the rule of thumb is to use as much pivots as they fit. Proceeding in this way, we can reduce the number of distances computed to solve a query, and it is useful for expensive distance functions. However, many of these indexes have a high internal cost, surpassing the cost of a sequential scan. Under this scheme, the selection of pivots is essential to reduce both the memory costs and the number of distances computed. Please notice that to be useful in practice, the actual search time must be several times smaller than the sequential scan for most cases and not only for extraordinarily expensive distance functions.

2.1.1 Pivot Selection Strategies

Since it is critical for the performance of pivot based indexes, a natural question is how to select *good* pivots. A fair rule is to select the pivots randomly, but it is well known that the election affects the performance of the search. The next indexes are built around strategies for selecting pivots.

Bustos et al. [Bustos03] introduced a list of pivot-selection strategies. The core of their contribution is a method to compare collections of pivot sets to decide which one has better performance. The authors claim that a *better* set of pivots will have a distance distribution of the mapped space with a bigger mean. This can be done taking a random sample and comparing the distances of the sample points with each set of pivots. They propose an *incremental* selection strategy, which consists on taking a set of N candidate pivots, and select the best one, say p_1. Then, from another set of N candidates select the one p_2 that makes $\{p_1, p_2\}$ the best set of pivots among the options. This procedure is repeated until k pivots are selected. This technique needs to know in advance the proper value of k.

The *Sparse Spatial Selection* [Pedreira07] (SSS) is a pivot based method that automatically determines and selects the number of *essential* pivots to be used by a pivot table. SSS depends on the intrinsic dimension of the space and not on the size of the database. The maximum distance d_{max} between any two objects in the database is a parameter needed in advance. The algorithm starts with an empty set, and incrementally constructs the set of pivots as follows. Each object is tested to be *well covered* for the current set of pivots, if it is not covered, it is promoted to be a pivot.

More formally, an object will be a new pivot if its distance to all the other pivots is greater than or equal to εd_{max} for a fixed $0 < \varepsilon \leq 1$. The goal is that the set of pivots are selected to be well distributed over the space. The authors conclude, based on their experiments, that the best value for ε should be 0.4; the authors postulate that this constant does not depend on the particular database.

The *Spatial Approximation Tree* (SAT) [Navarro02a] Is a metric tree that creates a compact partition of the data. The root a of the SAT is connected to its children $N(a)$, the remaining elements of the database are assigned to the closest children of a. This process is repeated recursively with each children and its assigned elements. The set $N(a)$ has the following two properties: for every $u, v \in N(a)$, $d(u, a) < d(u, v)$ and $d(v, a) < d(u, v)$; and for every w not in $N(a)$ there exist $z \in N(a)$ such that $d(z, w) < d(z, a)$. For a given element c, multiple $N(c)$ can be build. One way to do that is to take the closest element to c, call it x, and put it into $N(c)$, then, all the points closer to x than to c can not be in $N(c)$ (second property) so they are discarded. From the rest of the elements, take the closest to c and repeat until all the points not in $N(c)$ are discarded. This construction of $N(c)$ will produce the simple SAT. Another way to get $N(c)$ is to take the farthest point y (instead of the closest) and put it in $N(c)$, from the points not discarded by y, take the farthest and repeat. The SAT produced in this way is called Distal SAT (DiSAT), recently described in [Chavez14b]. Note that we can insert in $N(c)$ any element, discard the corresponding items and insert any other valid point, and repeat this process until all the points are discarded or in $N(c)$. Inserting points randomly in $N(c)$ will produce SAT^{Rand}. The authors have found that the DiSAT is the best option. This index has good performance and no construction parameters (other than the order of the objects in $N(c)$); this simplicity of use makes the SAT a fair choice when there is not much knowledge of the database.

The *List of Clusters* was introduced by Chavez and Navarro [Chávez05b] as a robust and memory efficient index, that works as follows. First an element p is selected and its m closer elements are the cluster of p. Then, from the remaining elements, another point is chosen and its cluster is formed with the m closer elements. This procedure continues until there are no points left. The covering radius is stored in each cluster. When doing a query, all the clusters are revised in the same order of construction, sequentially, to see if they may contain an answer. If the query ball is contained in a cluster, there is no need to check further clusters. The LC needs linear memory and nearly quadratic construction time for the useful combination of parameters. Using $O(\log n)$ bits

per item, it remains unbeatable on datasets with very high intrinsic dimensionality.

the *Priority Vantage Points* (KVP) were introduced by Celik in [Celik02a, Celik08], the author notices that disregarding the amount of pivots available, only *good pivots* make the elimination of points during a search. The author postulates that those pivots are the ones that are either close or far from the query. Each point will be *visible* only by its k best pivots. The total number of distances stored will be kN with N the size of the database. This modification allows the KVP to use less space than other pivot methods, the free space can then be used to add more pivots to the system. The parameter k must be set from the start and the authors did not gave a method to tune it.

The *Vantage Point Tree* (VPT) [Yianilos93] is a binary tree where each element of the database is either a node or a leaf. It is constructed recursively from a random point p as the root, then from all the remaining points, the median M is computed, the points x such that $d(p, x) < M$ got to the left side and the rest go to the right side. The value M is stored. For the left and right branches the process is recursive until reaching a leaf with just one element. The search for elements at distance r or less from q is done starting from the root p; if $d(q, p) - r \leq M$ the left side can be discarded; if $d(q, p) + r < M$ the right side will not contain an answer. Note that sometimes both, left and right sides need to be visited.

The methods described above, encompass the state of the art in exact indexing. Other methods are not as competitive, neither in total time nor in computed distances.

2.2 Extreme Pivots

A closer look into the definition of the pivot rule is revealing. Since we take the maximum distance over all pivots, only one pivot realizes the discarding expression. In other words, for every element in the database (for a particular query) only one pivot is responsible for filtering. The query location is unknown in advance and multiple pivots in a table increase the chance of filtering non relevant objects. With this observation in mind, it makes sense to seek for just one pivot for each database element: exactly the pivot that increases the chance of filtering for an unknown query.

2.2.1 The Extreme Approach

Instead of finding the best set of pivots $\mathbb{P}$, the Extreme Pivots (EP) technique tries to associate each item u to a good (maybe the best) available pivot $\mathsf{piv}(u)$ for that object. EP will try to select the pivot $\mathsf{piv}(u) = p$ such that

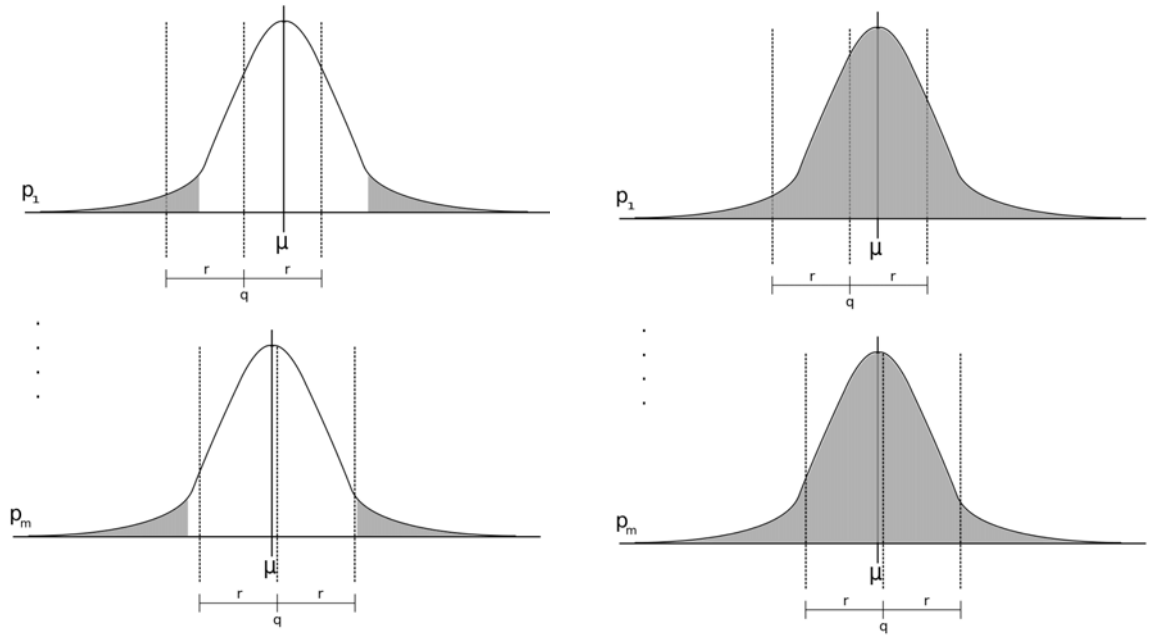
$$|d(u, \mathsf{piv}(u)) - d(q, \mathsf{piv}(u))| \text{ is maximized,}$$

where q follows the distribution of the expected queries. It will be assumed that the queries have the same distribution of the database. More detailed, a pivot p covers u if p can prove that it is closer or farther than any other available pivot. This implies that the points covered by a pivot will be close or far from it. Notice that the set of available pivots does not need an especial selection strategy or optimization, mainly because this task will be performed by the above maximization.

As other pivot selection techniques, the number of pivots can be large, depending on the dimensionality and size of the dataset. Nevertheless, due to the mentioned selection, each object in the database will be covered by exactly *one* pivot and hence the memory is $O(n)$, i.e., it is equivalent to a single pivot in a pivot table. The structure storing $\mathbb{P}$ and $\{d(u, \mathsf{piv}(u))\}$ for each u in the database is called a *pivot group* (PG). It is not hard to see that the discarding power of EP can be improved using several independent pivot groups. This will make the points to have more than one pivot. There does exist a simple implementation of EP using tables (EPT). Each pivot group corresponds to an EPT's row. We will describe it in detail in Section 2.3, but now let us continue the description of the motivating ideas focusing in EPT using a single PG.

Figure 2.1 illustrates the central idea of EPT and its differences with other pivot tables. In a PG, each item u in the database is associated to a single pivot $\mathsf{piv}(u)$, and $d(u, \mathsf{piv}(u))$ is far from the mean μ of the histogram of $\mathsf{piv}(u)$. Only m pivots are stored in a single pivot group needing $O(n)$ memory. Just to fix ideas, consider the query (q, r). The average $d(q, p)$ will be close to μ and all $d(u, p)$ will be far from μ (Figure 2.1(a)). The intersection of the query ball and the populated ranges in each pivot will be small.

In contrast, other pivot tables (no matter the pivot selection technique) will have a full view of the dataset per pivot. Since the queries will follow the same distribution, then $d(q, p)$ will be close to the mean for most pivots (Figure 2.1(b)). A pivot table needs to contain a pivot with a



(a) ***The case of EPT. From the perspective of the pivot*** p***, the closed ranges*** 0 ***to*** $\mu - \alpha$ ***and*** $\mu + \alpha$ ***to*** ∞ ***are populated with items from the datasets, the interior range is empty.***

(b) ***The behavior of other pivot tables, no matter the pivot selection strategy. Each pivot sees the entire dataset.***

Figure 2.1: Solving (q, r) with EPT and other pivot tables.

low populated range (i.e., far from the mean), for each possible query. However, the latter implies a large pivot set, and the pivot table will need $O(kn)$ memory, for k pivots. Moreover, a large pivot set will also contribute to the search cost since the cost of the intersection of the ranges will be proportional to k in the worst case.

A formal definition of the above intuition follows. Recall S is the database. A *Pivot Group* (PG) $\mathbb{P}$ is a subset of S, its elements are called *pivots*. Every $p \in \mathbb{P}$ has a set $A(p) \subset S$ called the associates of p with the following properties:

- Every $x \in A(p)$ is such that $|\mu_p - d(p, x)|$ is maximum for all the pivots in $\mathbb{P}$ where μ_p is the expected value of $d(u, p)$ for all $u \in S$. Ties are broken arbitrarily.
- $\cup_{p \in \mathbb{P}} A(p) = S$.
- $A(p_i) \cap A(p_j) = \emptyset$ for $i \neq j$.

Note that the associate sets define a partition of the database S which induces an equivalence relation. If we have a small number of pivots, we will have equivalence classes with many pivots and many pivots will produce small equivalence classes.

For x in S, let $\mathsf{piv}(x)$ be the pivot p such that $x \in A(p)$. From the definition of a PG, $|\mu_p - d(p, x)|$ where $p = \mathsf{piv}(x)$ is likely to be large as it is the biggest value for all the pivots, then the points in $A(p)$ will be either close or far from p.

2.2.2 Bounding Parameters in PG

The internal cost of the search will be the number of pivots in a PG. We can set the number of pivots and every one will chose the best points. With this, each pivot will have a certain number of points as associates. If the pivots are uniformly distributed, we can expect that everyone will have the same number of points as associates, this will give a probability a of a point to be associated to a given pivot. Since the points in $A(p)$ are either close or far from p, we can suppose that for every $u \in A(p)$ we have $|d(p, u) - \mu_p| > \alpha$ for every pivot p.

It is presented a relation among a and α in the next theorem. Notice how a is closely related to the expected query set. Let $Pr(x)$ be the probability of occurring x.

Theorem 2.2.1 *Let Y be the random variable describing the distance between a pivot p and an indexed item u associated to p, i.e., $Y(u) \sim d(p,u)$, then $a = Pr(u \in A(p)) < \frac{\sigma_Y^2}{\alpha^2}$*

Proof 2.2.1 *[Sketch] Using the Chebyshev's inequality,[1] $Pr(|Y - \mu_Y| > \alpha)$ is upper bounded by $\frac{\sigma_Y^2}{\alpha^2}$. Since $a = Pr(u \in A(p)) < Pr(|Y - \mu_Y| > \alpha)$ (see Figure 2.1(a)). The proof follows then.*

This theorem shows that for a given Y (fixed dataset and query), we can adjust a to obtain the necessary α such that indexed items escape from the zone around the mean from their associate pivot's point of view. However, highly concentrated distributions will produce small a values (large α, see Figure 2.1(a)), then it will need a lot of pivots. Notice that this high number of pivots will not produce memory problems in EP, yet it will increase the index's search cost. Based on Theorem 2.2.1, the number of necessary pivots is bounded by below as:

$$m > \frac{\alpha^2}{\sigma_Y^2}, \text{ since } m = 1/a. \tag{2.1}$$

Note that m could be independent of n, yet dependent of the data and query set distribution. This bound of m is highly informative, yet it does not provide a proper method to upper bound m. In the next section, we introduce formally the EP Table (EPT), a practical implementation of EP. The construction of EPT is based not in determine the necessary m directly, yet it tries to optimize online the expected search cost as a function of m (detailed below). This is just a practical shift of the discussion given above.

2.3 Extreme Pivot Table

Pivot groups can be used to construct independent indexes, the final set of candidates will be the intersection of all the independent outputs. Each pivot in a group can only be used to discard its associates. For simplicity, the index will be thought as a table, although sublinear implementations for pivots are available. An *Extreme Pivot Table* (EPT) will be a collection of ℓ pivot groups. Note that each point will be in ℓ associate sets, this corresponds to each point having ℓ pivots. So, m pivots means m pivots per PG, and ℓ pivots, will mean pivots per point of the database.

[1] For a random variable Z with mean μ_Z and variance σ_Z^2, $Pr(|Z - \mu_Z| > \epsilon) < \sigma_Z^2/\epsilon^2$

2.3.1 Data Structure and Search Algorithm

An EPT is composed of ℓ pivot groups. Each pivot group is a list of tuples $(\mathsf{piv}(u), d(u, \mathsf{piv}(u)))$. More precisely, we have the following matrix:

$$T = \begin{bmatrix} (P_{1,1}, D_{1,1}) & \dots & (P_{1,n}, D_{1,n}) \\ \vdots & & \\ (P_{\ell,1}, D_{\ell,1}) & \dots & (P_{\ell,n}, D_{\ell,n}) \end{bmatrix}$$

where $P_{i,j}$ stores $\mathsf{piv}_i(u_j)$ and $D_{i,j}$ stores $d(u_j, \mathsf{piv}_i(u_j))$. Since each pivot group (row) is independent piv should be parametrized with the PG identifier to properly distinguish among the available pivot groups.

The other typical pivot tables store T just using D. So, EPT will use $\log(m) + 1$ extra bits per item which is non-significant for most applications, as the number of pivots m is fixed for a fixed data and query set (Expression 2.1). Notice that m can be fixed as the maximum m among all the pivot groups without significant memory waste.

Once T is constructed, a query (q, r) is solved using Algorithm 1. Every point in the database is tested if one of its ℓ pivots can discard it using the pivot rule, if it is not discarded, a direct comparison with the query is done to see if it is part of the result. A $\mathsf{k\text{-}nn}$ query is solved in a similar way, yet using min-priority queues of maximum size k. This means that the furthest item will be discarded whenever a closer object is found. To solve $\mathsf{k\text{-}nn}$ queries, Algorithm 1 replaces r by the currently known covering radius cov of the result set. Formally, $\mathsf{cov}(R)$ is evaluated to the furthest distant in R if $|R| \geq k$ and ∞ otherwise. To make this algorithm's transformation clearer, the pair (distance and object) will be append to the result set (line 14).

2.3.2 Probability of Discarding

Given a query (q, r), the probability that an element u is not discarded by ℓ pivots is

$$Pr(|d(q, p_1) - d(u, p_1)| \leq r, \dots, |d(q, p_\ell) - d(u, p_\ell)| \leq r).$$

To simplify the analysis, it will be assumed that all the objects are described by inde-

Algorithm 1 The search algorithm for the EP Table

Input: A query (q, r), the EP Table T of size $\ell \times n$, the dataset S
Output: The result set R

```
1: Let R be an empty set
2: Let H : S → R be a hash-map caching pivot distances to q, i.e., H(p) = d(p, q). For the sake of
   simplicity, if H(p) is not cached, then it is computed and stored in H(p).
3: for u ∈ S do
4:     Let review be true.
5:     for i = 1 to ℓ do
6:         if |H(piv_i(u)) − d(u, piv_i(u))| > r then
7:             Set review to false.
8:             break for
9:         end if
10:    end for
11:    if review then
12:        Let d* = d(u, q)
13:        if d* ≤ r then
14:            R ← R ∪ {(d*, u)}
15:        end if
16:    end if
17: end for
```

pendent identical distributed random variables (i.i.d.r.v). Therefore, the previous expression can be rewritten as follows

$$Pr(|d(q,p) - d(u,p)| \leq r)^{\ell}.$$

In order to predict the above probability, the distribution of distances from the pivots to all the database elements will be used. First, the probability that a pivot would discard one of its associates will be studied.

For pivot p, let X be the random variable such that $X(u) \sim d(p,u)$ if $u \in A(p)$ and $X(u) \sim 0$ otherwise, that is because p only has information for $A(p)$. Similarly, let Y be the random variable such that $Y(q) \sim d(p,q)$. The variable Y will represent the distribution of q. Let assume, X and Y have a normal distribution with mean μ, $E[X] = E[Y] = \mu$. The domain will be divided on intervals of length r. Let I_i be the interval $[ir, (i+1)r)$ for $i \in \mathbb{N}$. It is convenient to express the mean in terms of the searching radius. It is safe to assume $\mu > r$, because otherwise the query is not selective at all and the entire database would be the answer to the query. Let $\mu = rc$ with $c > 0$.

$$Pr(|d(q,p) - d(u,p)| > r) = Pr(u \text{ is discarded}) =$$

$$\sum_{i=0}^{K} Pr(u \text{ is discarded}|q \in I_i)Pr(q \in I_i)$$

$$+Pr(u \text{ is discarded}|q \in [(K+1)r, \infty)Pr\left(q \in [(K+1)r, \infty)\right) \text{ for every } K \in \mathbb{N}.$$

Remember that the distance from any u in $A(p)$ to μ is bigger than α, it will also be assumed that $\mu - \alpha = br$ for some b.

$$Pr(u \text{ is discarded}) = 2\sum_{i=0}^{c-1} Pr(u \text{ is discarded}|q \in I_i)Pr(q \in I_i)$$

$$= 2\sum_{i=0}^{b} Pr(u \text{ is discarded}|q \in I_i)Pr(q \in I_i)+$$

$$2\sum_{i=b+1}^{c-1} Pr(u \text{ is discarded}|q \in I_i)Pr(q \in I_i).$$

Note that $Pr(u \text{ is discarded}|q \in I_i) = 1$ when q is in $[b(r+1), \mu]$, so

$$Pr(u \text{ is discarded}) \geq 2\left(\frac{1}{2} + \int_0^{\mu-\alpha-3r} f_X(x)dx\right)\int_0^{\mu-\alpha+r} f_Y(x)dx$$

$$+2\int_{\mu-\alpha+r}^{\mu} f_Y(x)dx$$

Let us see an example of applying the last inequality. In this assumption Y distributes normally, so the values of the above integrals can be found on the literature. Let σ_Y^2 be its variance, and $\alpha = \sigma + r$. Then

$$Pr(u \text{ is discarded}) \geq 2\left(\frac{1}{2} + \int_0^{\mu-\sigma_Y-4r} f_X(x)dx\right)(.16) + 2(.34)$$

$$\geq 2\left(\frac{1}{2}\right)(0.16) + 2(0.34) = 0.84$$

With the last parameters we have a probability of discarding bigger than 0.84.[2]

[2]The precise values 0.16 and 0.34 can be numerically computed or retrieved in traditional pre-computed tables for the

An insightful analysis. Perhaps a less precise yet richer analysis can be obtained using probabilistic analysis. In this terms, the discarding problem can be stated as

$$Pr(|d(p,u) - d(p,q)| > r) = Pr(|X - Y| > r)$$

then, using the Chebyshev's inequality,

$$Pr(|X - Y| > r) < (\sigma_X^2 + \sigma_Y^2)/r^2.$$

The probability that a given u would not be discarded by its covering pivot $\mathsf{piv}(u)$, is

$$1 - Pr(|X - Y| > r) \geq 1 - (\sigma_X^2 + \sigma_Y^2)/r^2.$$

This expression directly relates the distribution of the dataset, the pivots as are seen by its associated pivots, and the query set. Recall that we can always adjust σ_X^2 increasing or decreasing α. To the best of our knowledge, EP is the first index having this capacity of adaptation to the intrinsic dimensionality.

2.3.3 Search Cost

The ultimate measure for efficiency will be the search cost. In a first approach the model of complexity where only the number of distance computations are counted will be used.

Let s be the probability that an element u is not discarded. Then the algorithm computes the following distances.

$$\text{cost} \quad = \quad m\ell + ns^\ell \tag{2.2}$$

for a database of size n, where m is the number of pivots in the PG and ℓ the number of groups. Notice that the number of groups corresponds with the number of extreme pivots saw per item, so, at this point it is correct to say that each item is associated with ℓ pivots. Since m is fixed for a given

normal distribution.

dataset (Expression 2.1), the focus will be on determining ℓ. We have

$$\frac{\partial \text{cost}}{\partial \ell} = m + ns^{\ell} \ln(s)$$

then

$$m + ns^{\ell} \ln(s) = 0$$

$$\ell^{\star} = \frac{\ln m/n - \ln \ln (1/s)}{\ln (s)}. \tag{2.3}$$

Using this optimal ℓ we can compute the minimum cost

$$\text{cost}^{\star} = m\ell^{\star} + ns^{\ell^{\star}}$$

$$= \frac{m \ln(\frac{m}{n}) - m \ln \ln 1/s}{\ln s} + ns^{\frac{\ln \frac{m}{n} - \ln \ln 1/s}{\ln s}}$$

$$= \frac{m \ln(\frac{m}{n}) - m \ln \ln 1/s}{\ln s} + n\left(\frac{-m}{n \ln s}\right)$$

$$= \frac{m \ln(\frac{m}{n}) - m \ln \ln 1/s - m}{\ln s}$$

$$\text{cost}^{\star} = \frac{m\left(\ln \frac{n}{m} + \ln \ln (1/s) + 1\right)}{\ln (1/s)} \tag{2.4}$$

$$= m \log_{1/s} \frac{n}{m} + o(m \ln (1/s)). \tag{2.5}$$

This expression is related to the cost obtained by Chavez et al. [Chavez01] for randomly selected pivots; i.e.,

$$\text{cost} \geq (\ln n + \ln \ln (1/t))/ \ln (1/t), \tag{2.6}$$

with $t = 1 - 2\sigma_Y^2/r^2$. This expression gets an optimal number of pivots,

$$k^\star = \frac{\ln n + \ln\ln(1/t)}{\ln(1/t)} \tag{2.7}$$

$$= \log_{1/t} n + o(\ln(1/t)). \tag{2.8}$$

The equations seem similar, however the main difference is that the search cost of random pivots (proposed in Chavez et al [Chavez01]) depends on the database and the query radius, and hence the only way to improve the search cost is by increasing the number of pivots. On the other hand, the cost of extreme pivots also depends of σ_X, adjusted through α, a parameter that helps us to adapt to the database and to a set of queries of interest. Remember that $\sigma_X^2 = E[(d(u,p) - \mu_p)^2]$, and by construction $|d(u,p) - \mu_p| \geq \alpha$, thus $\sigma_X^2 \geq \alpha^2$. Therefore we have the chance to make adjustments to get better results on average, by knowing how the database is distributed. Also, in our cost equation, we can set the ℓ and m parameters (m depends on α) at construction time, ℓ can control the memory needed by the index. A larger m will increase the probability of discarding a given element, without increasing the index size.

2.3.4 Construction Algorithms

Now, it will be shown two algorithms to create EPT indexes. The first algorithm corresponds to a first approximation appeared in [Ruiz13]. It is convenient and correct, but it uses a parameter $\beta = 0.8$ to control some practical issues with the estimation of the needed statistics. The second one (Algorithm 3) is part of the contribution of this new version and replaces β in favor of a more intuitive and self-explained parameter.

We can optimize the parameters m and ℓ using the model and the analysis described above. The optimal α is achieved maximizing the probability of discarding an object u, which is approximated by $1 - (\sigma_X^2 + \sigma_Y^2)/r^2$. For high intrinsic dimensional datasets α will be large, and it will not be useful (because it will produce a very large m). In this case, a suboptimal α can be used and the overall performance could be improved increasing the number of pivot groups, i.e., increasing ℓ. The same strategy is useful when the computation of the distance function is too expensive.

The total amount of memory used can be set by fixing ℓ. Once fixed, we can approximate

the optimal m numerically. For this purpose Equation 2.2 will be used as detailed in Algorithm 2. Here, the idea is to increment m by one at each step, and stop whenever the derivative of Expression 2.2 becomes zero or positive. This procedure will create a single group, so it must be called ℓ times.

Algorithm 2 Numerically optimized construction of the EP-Table. β controls the variance.

Input: The input database $S = \{u_1, u_2, \cdots, u_n\}$, and the number of groups ℓ.
Output: The set of pivots P, and the array g of n tuples $(\mathsf{piv}(u), d(u, \mathsf{piv}(u)))\ \forall u \in S$.

1: Estimate σ_Y^2 and r^2.
2: Select a random pivot p_1, $P \leftarrow P \cup \{p_1\}$
3: Initialize $g[1, n] = (p_1, d(u_1, p_1)), (p_1, d(u_2, p_1)), \cdots, (p_1, d(u_n, p_1))$
4: Define $\beta = 0.8$
5: Let $m \leftarrow 1$
6: Let $prev = m\ell + n(1 - \beta(\sigma_X^2 + \sigma_Y^2)/r^2)^\ell$
7: **while** True **do**
8: $m \leftarrow m + 1$
9: Select p_m randomly from S, $P \leftarrow P \cup \{p_m\}$
10: **for** $j = 1$ to n **do**
11: $g[j] \leftarrow (p_m, d(u_j, p_m))$ **if** $|d(u_j, p_m) - \mu| > |d(u_j, \mathsf{piv}(u_j)) - \mu|$
12: **end for**
13: Update σ_X^2 with the current tuples in g
14: $cost \leftarrow m\ell + n\left(1 - \beta(\sigma_X^2 + \sigma_Y^2)/r^2\right)^\ell$.
15: **if** $cost \geq prev$ **then**
16: **stop** loop
17: **end if**
18: $prev \leftarrow cost$
19: **end while**

It is important to notice that this procedure depends on the estimated values σ_Y^2 and r^2. Also, for real world databases the i.i.d.r.v. assumption can be far from true. In a preliminary version of this index [Ruiz13] a constant $\beta \leq 1$ was introduced to compute the discarding probability in lines 6 and 14. The precise value of β depends on how much both the database and the query set differ from the i.i.d.r.v. assumption. In the experiments, the value $\beta = 0.8$ works for most setups.

Now, a new construction algorithm, not based on β is introduced where a more explicit manipulation of the data variance is used. Algorithm 3 shows the new construction algorithm. The central idea is to consider the cost function as a *noisy signal* that should be smoothed. This is done taking the average of the cost after a number of pivot insertions (line 19). New pivots are added until the average cost starts to being worst. Any value $window > 1$ will produce a smoother signal (line 1), but values from 4 to 32 works well for all our datasets. Notice that this strategy can add

$window \times \ell$ unnecessary pivots, which can be a problem if $window \times \ell$ is large (thousands); this effect will not appear when $window$ is small.

Algorithm 3 Numerically optimized construction of the EP-Table using signal processing ideas to handle the variance of the data. Without loss of generality, the algorithm assumes that database length is a multiple of $window$.

Input: The input database $S = \{u_1, u_2, \cdots, u_n\}$, and the number of groups ℓ.
Output: The set of pivots P, and the array g of n tuples $(\mathsf{piv}(u), d(u, \mathsf{piv}(u)))\ \forall u \in S$.

Define $window = 16$.
Estimate σ_Y^2 and r^2
Let $prev \leftarrow n$, and $m \leftarrow 0$
while True **do**
  $cost \leftarrow 0$
  for 1 to $window$ **do**
    $m \leftarrow m + 1$
    Select p_m randomly from S, $P \leftarrow P \cup \{p_m\}$
    if $m = 1$ **then**
      $g[1, n] \leftarrow (p_1, d(u_1, p_1)), (p_1, d(u_2, p_1)), \cdots, (p_1, d(u_n, p_1))$
    else
      for $j = 1$ to n **do**
        $g[j] = (p_m, d(u_j, p_m))$ **if** $|d(u_j, p_m) - \mu| > |d(u_j, \mathsf{piv}(u_j)) - \mu|$
      end for
    end if
    Update σ_X^2 with the current tuples in g
    $cost \leftarrow cost + m\ell + n\left(1 - (\sigma_X^2 + \sigma_Y^2)/r^2\right)^{\ell}$
  end for
  if $cost/window - prev \geq 0$ **then**
    stop loop
  end if
  $prev \leftarrow cost/window$
end while

2.4 Experiments

The definitive test for an indexing algorithm is the comparison against state of the art indexes in a common testbed. This comparison is necessary because most indexes are not analysed in the standard sense of algorithms. This in turn is because of the curse of dimensionality and the relative weakness of the distance properties. This also happens in other algorithmic areas like compression. Now, it is presented an exhaustive empirical analysis in a standard testbed used in the literature. The datasets used were both real world and artificial, as described below.

— **Nasa.** This database is a collection of 40,150 vectors of 20 coordinates obtained from the SISAP project (`http://www.sisap.org`). It uses L_2 as distance function. A single search is completed on 0.014 seconds.

— **Colors.** The second benchmark is a set of 112,682 color histograms (112-dimensional vectors) from SISAP, under the L_2 distance. Each query needs 0.165 seconds to be solved.

— **Gutenberg.** A dictionary of 2,178,587 words extracted from the Gutenberg project (`www.gutenberg.org`). Several languages were considered and mixed. The Levenshtein's distance was used to measure proximity between objects. An exhaustive verification of all items in the dictionary needs 16.245 seconds to solve a single query.

— **CoPhIR-1M.** A one million subset of the Cophir dataset [Bolettieri09]. Each object is a 208-dimensional vector, using L_1 as distance. Each query needs 2.052 seconds to be completed.

— **RVEC.** In order to know the effect of the intrinsic dimension in the indexes random vectors databases were generated in the unitary cube, in five dimensions: 4, 8, 12, 16 and 20, each one with one million unique items. Also, in order to study the effect of the database's length, five datasets were created, of dimension sixteen, containing $10^5, 3\times10^5, 10^6, 3\times10^6$, and 10^7 items, respectively. Each database is dubbed as RVEC-$\{$4, 8, 12, 16, 20$\}$ for the one million case, and RVEC-16-SIZE, for the variable length dataset. The L_2 distance was used with the RVEC dataset; however, the cost of each distance computation was artificially increased by computing δ times each distance, with δ the dimension of the vectors. The general idea is to simulate a costly distance, as found in real datasets. This test is common in indexing because real-world data tend to be clustered and the intrinsic dimension is way smaller than the representational dimension.

The distance function was used as a black box, without using the coordinate information. This is also standard in benchmarking metric indexes.

Notice that this study is placed in a general metric space model, that is, every search method uses the distance as a black box, in other words, the methods cannot take advantage of the underlying structure of the objects. This restriction allows supporting any metric distance function,

and enables the creation of generic search methods without taking care a particular application domain.

Each plot represents average values over 256 nearest neighbor queries. Query objects were not indexed. In the Gutenberg case, each query was created modifying an indexed word in a single position, ensuring that the final object is not part of the original dataset. Several canonical and state of the art metric indexes were used as baseline for the comparison.

1. The *Sequential* or *exhaustive* scan to bound the searching time when the dimension is large.
2. The *LAESA*, the standard pivot table.
3. The *List of Clusters* (LC) [Chávez05b], which until now it holds the best performance (using the right setup) at equality of memory. The LC cannot improve its performance adding memory resources.
4. The fourth baseline is a randomized SAT [Chávez14a], which is a recent improvement over the classical *SAT* [Navarro02a]. The SAT is a parameterless metric index.
5. A fifth baseline is the incremental selection of pivots by [Bustos03] (dubbed as *BNC-Inc*).
6. The Spatial Selection of Sparse Pivots (*SSS*) of [Pedreira07].
7. Finally, the K Vantage Pivots (*KVP*) by [Celik08].

It will be shown that EPT can be faster than LAESA with 64 pivots, using a small fraction of the memory. If allowing the same index size, EPT is several times faster. For LC, which uses a fixed amount of memory, the performance is governed by the bucket size (which in turn increase the construction time). The best bucket size for each dataset was used, selected from sizes $16, 32, 64, 128, 256, 512, 1024,$ and 2048.

The two versions of the EPT construction algorithm were tested, those described in Algorithm 2 and 3. The figures labeled these indexes as EPT and EPT*, respectively. However, both will be referred as EPT whenever the context allows, this will be the Extreme Pivot Table data structure. As described in the preliminary work [Ruiz13], EPT fixes $\beta = 0.8$ to ease the negative effects of the variance of the data. On the other hand, EPT* fixes $window$ to be 16, which works fine and will

produce a small effects when overfitting. The construction algorithms need to estimate the query set statistics. Remember the assumption that queries have the same distribution of the objects.

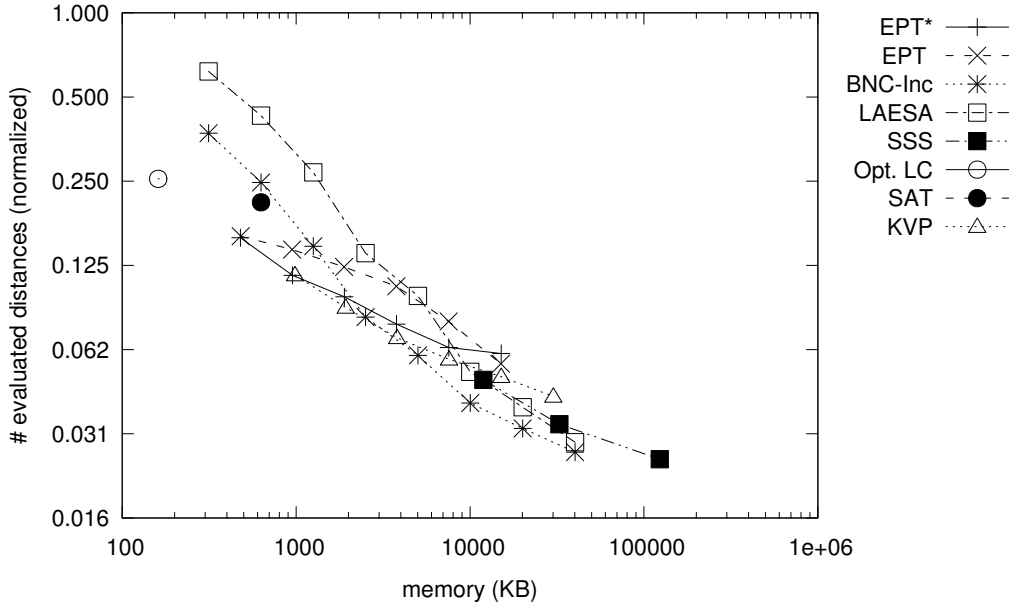
The algorithms were implemented in C# with the Mono framework[3]. Algorithms and indexes are available as open source software in the *natix* library[4]. All experiments were executed in a 4x quadcore Intel Xeon 2.40 GHz workstation with 32GiB of RAM, running Linux / CentOS 5.5 without exploiting the multicore architecture for search experiments.

2.4.1 Performance of the EPT per Database

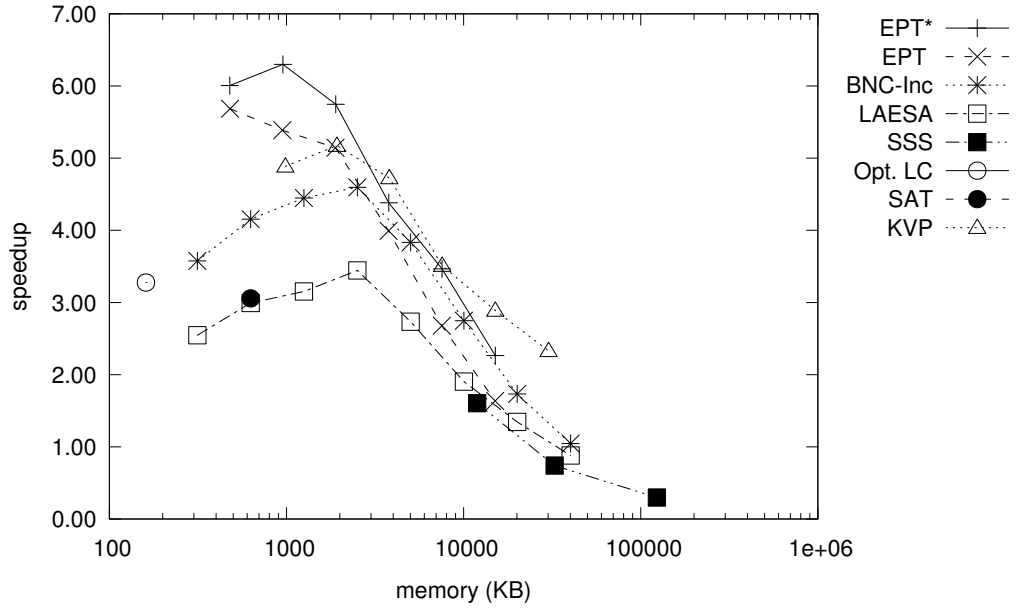
The Nasa database is the smallest and all the methods have a good performance on it. Figure 2.2(a) shows that all the methods have a similar performance in the number of distances computed. Figure 2.2(b) presents how much faster are the methods compared to sequential search. Not all the indexes profit from using more memory because the internal mechanism to navigate the index have complexity comparable to sequential scan. This phenomenon is seen in all the databases tested.

In Figure 2.3(a) we see that in the Colors database the EPT is the best choice when we have fixed memory. Figure 2.3(b) clearly shows the EPT dominance in this database. It is important to notice that the usual way to compare methods is to count the number of distances needed to solve the queries to make the analysis independent of the metric, a finer analysis need to include total search time for real applications.

[3] http://www.mono-project.org
[4] http://github.com/sadit/natix/

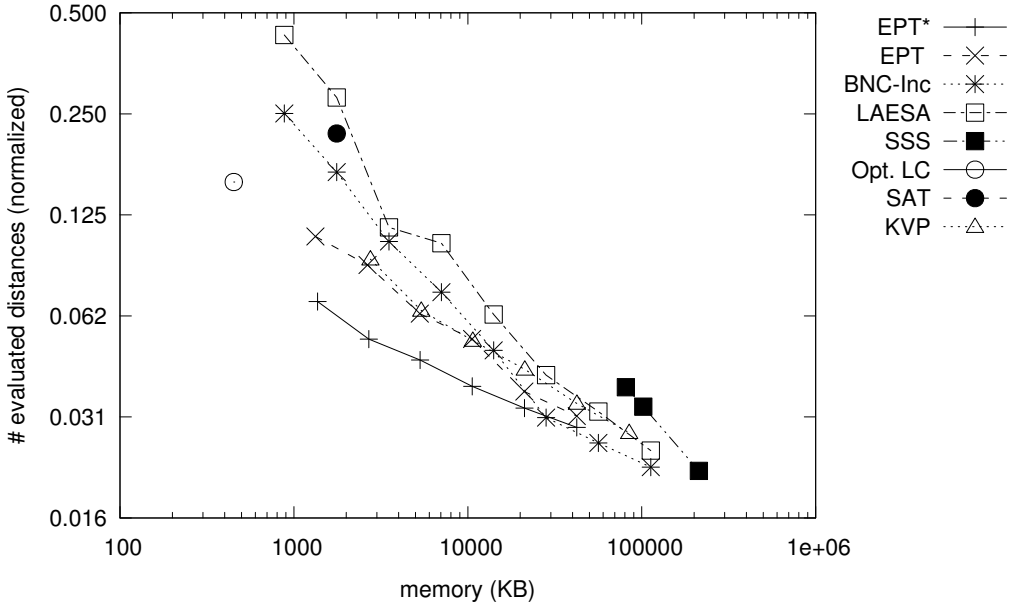


(a) ***Average search cost***

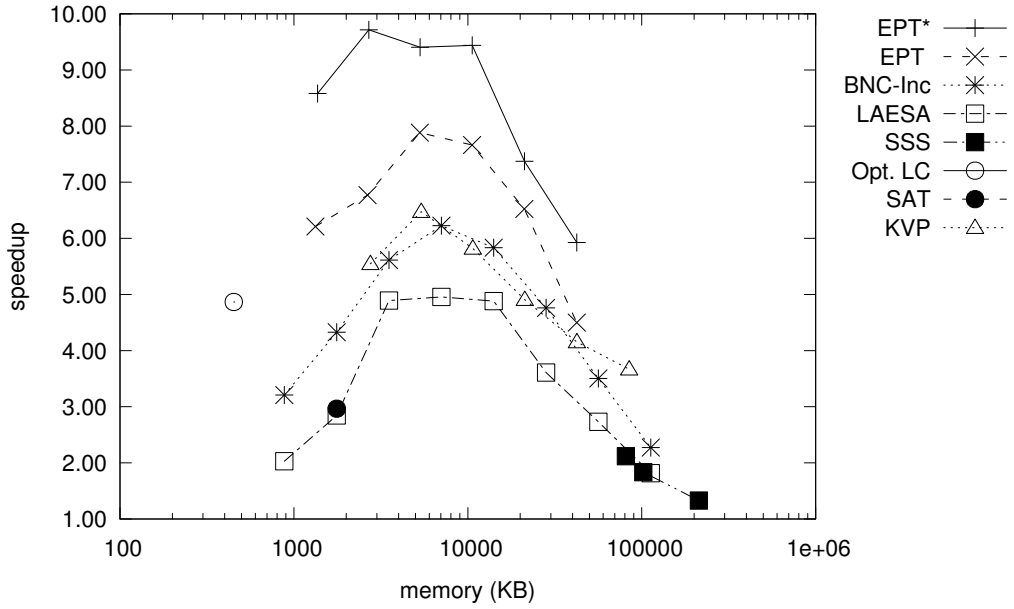


(b) ***Average speedup as compared with exhaustive search***

Figure 2.2: Performance comparison of state of the art exact metric indexes for nearest neighbor search on the Nasa database. The memory of the index is proportional to the number of groups or pivots of the index. EPT* and EPT use $1, 2, 4, 8, 16$, and 32 groups; BNC-Inc uses $1, 2, 4, 8, 16, 32, 64$ and 128 pivots. SSS uses $\varepsilon = 0.3, 0.4$, and 0.5 (the precise number of pivots depends of ε and the properties of the database). Eight bytes were used for floating point numbers. SAT and LC have the smallest memory usage. It is worth noticing that in the plots LC use the best setup selected among $n/m = 1024, 512, 256, 128$, and 64.



(a) ***Average search cost***



(b) ***Average speedup as compared with exhaustive search***

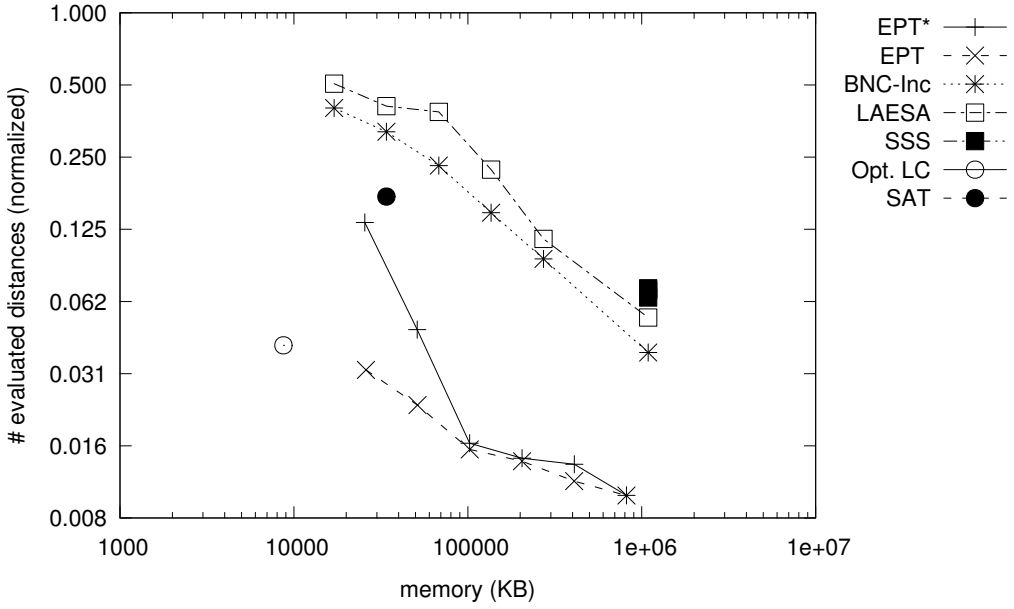
Figure 2.3: Performance comparison of state of the art exact metric indexes for nearest neighbor search on the Colors database. The memory of the index is proportional to the number of groups or pivots of the index. EPT* and EPT use $1, 2, 4, 8, 16$, and 32 groups; BNC-Inc uses $1, 2, 4, 8, 16, 32, 64$ and 128 pivots. SSS uses $\varepsilon = 0.3, 0.4$, and 0.5 (the precise number of pivots depends of ε and the properties of the database). Eight bytes were used for floating point numbers. SAT and LC have the smallest memory usage. In the plots the LC use the best setup selected among $n/m = 1024, 512, 256, 128$, and 64.

The Gutenberg database experiments took more time than the others. The results are shown on Figures 2.4(a) and 2.4(b) and it is clear that the EPT and EPT* are set apart, and are clear winners. The only possible exception is the LC. If we compare all the figures, we will see that the LC needs small space and has a good performance as compared with memory-limited configurations of the other methods. We will see that, when compared in memory-free configurations the LC is easily surpassed.

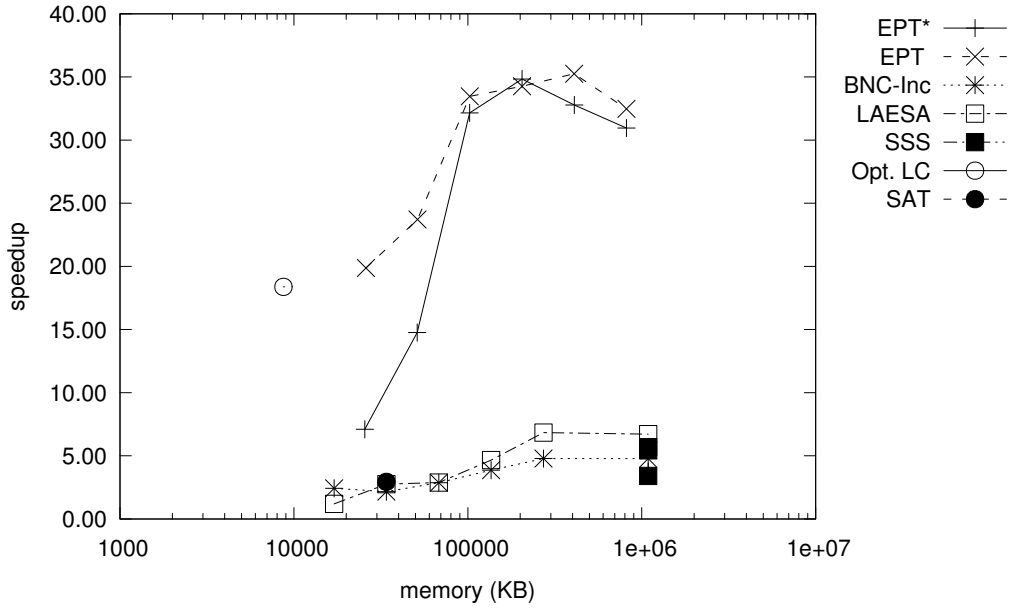
Figure 2.5(a) shows that on memory-limited configurations the best performance is with the EPT on Cophir. This is also true for speedup, as shown in Figure 2.5(b). One observation is that memory-limited Indexes can be used in the mobile devices. The smartphones and tablets processors are powerful enough to run these indexes and, since their memory is limited, the EPT is the best option for them. On the other hand, high end workstations and servers can use larger databases, without using secondary memory.

The above experiments encompass all the real world databases available for benchmark. In all of them the EPT had a better performance than the other methods in memory-limited configurations.

The results of the experiments with the random database of dimension 16 is on Figure 2.6. This dataset was considered because this dimension marks the limit where exact indexes can work. For higher dimensions the Curse of Dimensionality prevents satisfying a query without making a sequential scan. This is well documented in the literature [Chavez01]. In this dataset the dominance of the EPT over the other methods is more clear. Also notice that EPT* is better than EPT. This is important because the $window$ parameter has a simpler definition than the EPT's β.

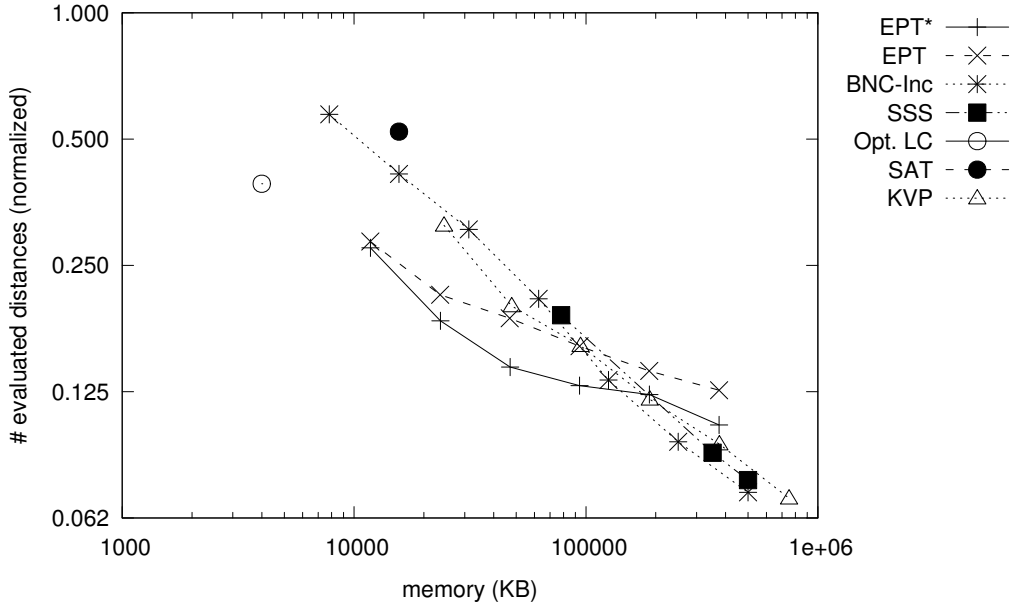


(a) ***Average search cost***

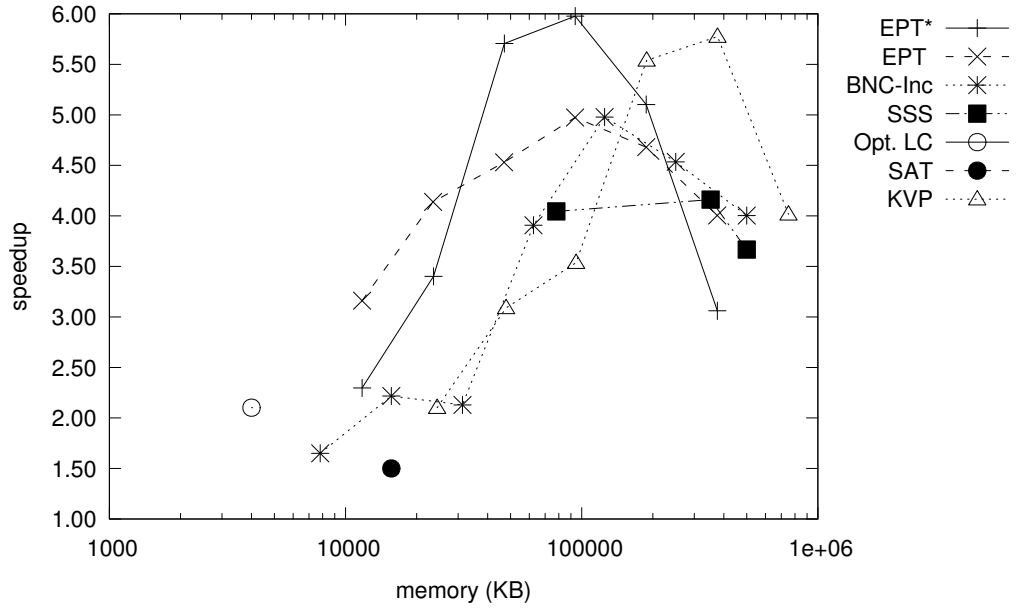


(b) ***Average speedup as compared with exhaustive search***

Figure 2.4: Performance's comparison using the Gutenberg database. All figures are compared using the memory as the basis. Each point in every curve correspond to an index setup using more memory (from left to right.)EPT* and EPT use $1, 2, 4, 8, 16$, and 32 groups; BNC-Inc uses $1, 2, 4, 8, 16, 32, 64$ and 128 pivots. SSS uses $\varepsilon = 0.3, 0.4$, and 0.5 (the precise number of pivots depends of ε and the properties of the database). Eight bytes were used for floating point numbers. SAT and LC have the smallest memory usage. In the plots the LC use the best setup selected among $n/m = 1024, 512, 256, 128$, and 64.

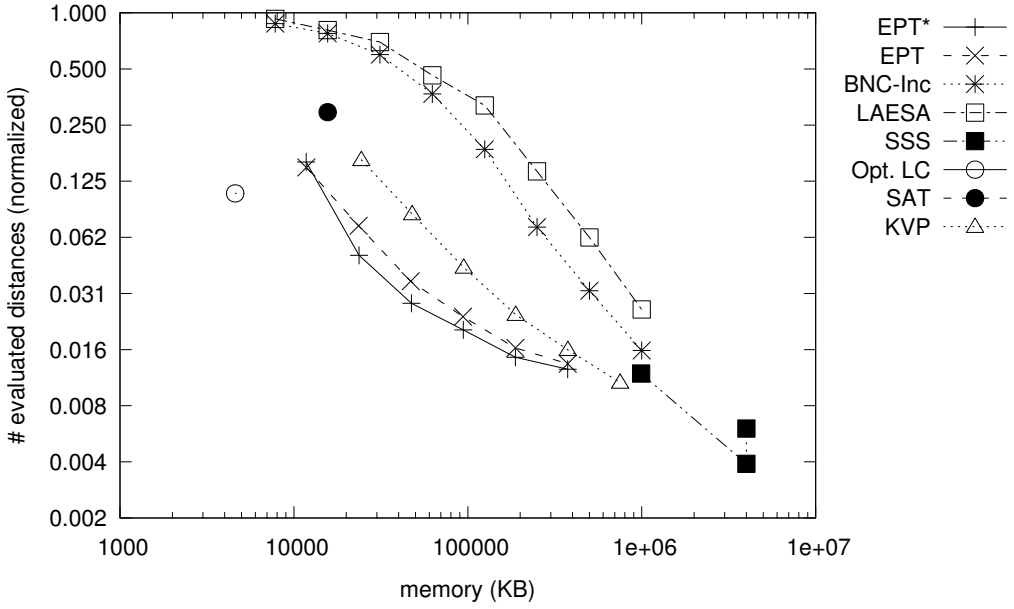


(a) ***Average search cost***

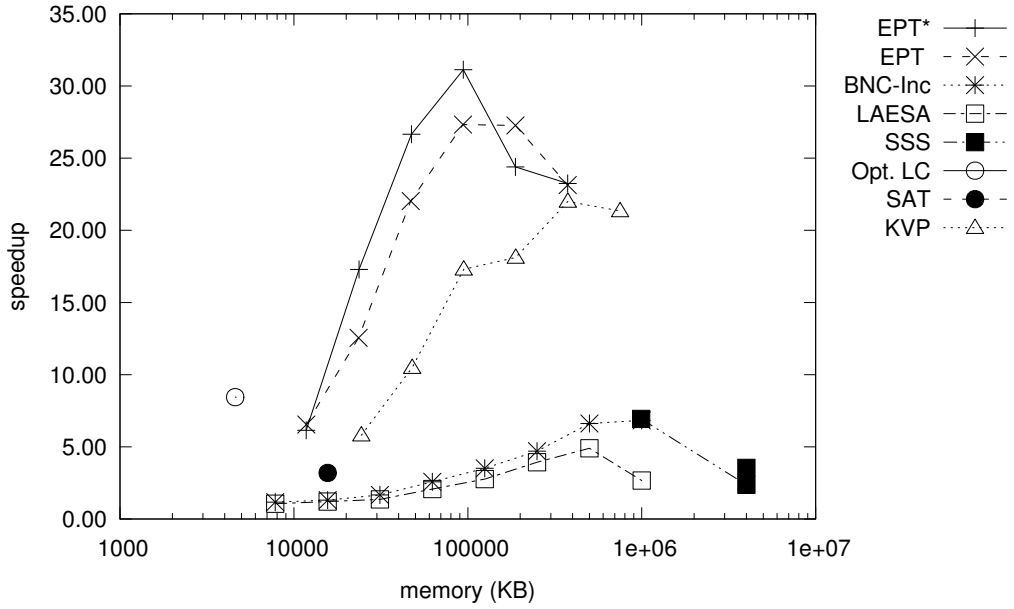


(b) ***Average speedup as compared with exhaustive search***

Figure 2.5: Performance's comparison using the CoPhIR-1M database. All figures are compared using the memory as the basis. Each point in every curve correspond to an index setup using more memory (from left to right.)EPT* and EPT use $1, 2, 4, 8, 16$, and 32 groups; BNC-Inc uses $1, 2, 4, 8, 16, 32, 64$ and 128 pivots. SSS uses $\varepsilon = 0.3, 0.4$, and 0.5 (the precise number of pivots depends of ε and the properties of the database). Eight bytes were used for floating point numbers. SAT and LC have the smallest memory usage. In the plots the LC use the best setup selected among $n/m = 1024, 512, 256, 128$, and 64.



(a) ***Average search cost***



(b) ***Average speedup as compared with exhaustive search***

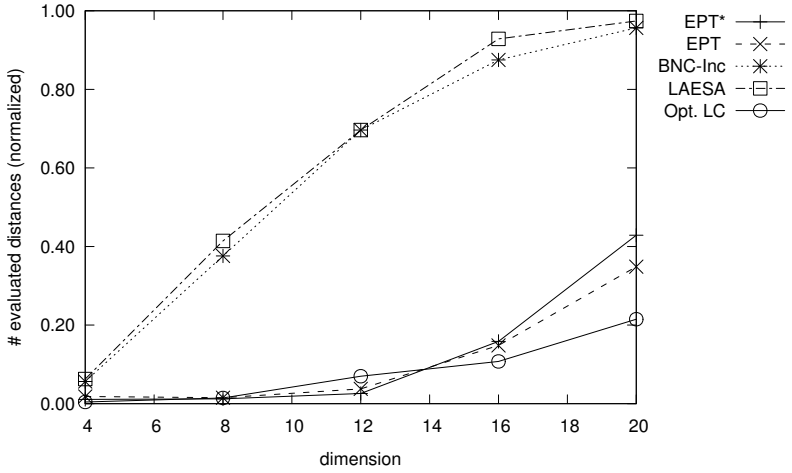
Figure 2.6: Performance comparison of state of the art exact metric indexes for nearest neighbor search on the RVEC-16-1M database. The memory of the index is proportional to the number of groups or pivots of the index. EPT* and EPT use $1, 2, 4, 8, 16$, and 32 groups; BNC-Inc uses $1, 2, 4, 8, 16, 32, 64$ and 128 pivots. SSS uses $\varepsilon = 0.3, 0.4$, and 0.5 (the precise number of pivots depends of ε and the properties of the database). Eight bytes were used for floating point numbers. SAT and LC have the smallest memory usage. In the plots the LC use the best setup selected among $n/m = 1024, 512, 256, 128$, and 64.

2.4.2 The Effect of the Dimension on the Search Performance

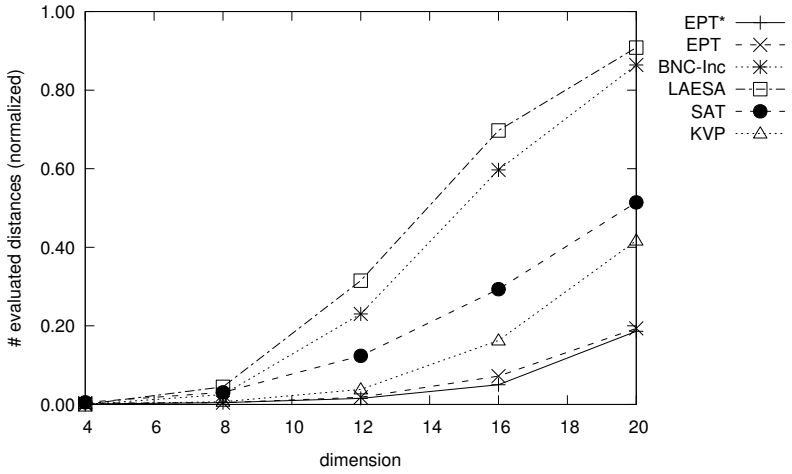
This subsection compares the performance of the methods when the dimension of the space increases, leaving the size of the dataset fixed. Four memory usage categories were arranged; the first with at most 3 integers per element; the second using between 3 and 10 integers; the next category uses at most 30 integers per item and finally, setting the limit to 100 integers per element in the database. In Figure 2.7(a) we have all the memory-limited setup, up to tree integers per element in the database. As it is natural, all the methods need to compute more distances as the dimension of the space increases. The optimal setup for LC is the best for high dimensions and together with the EPT, are the only choices to consider for those cases. The proposal is faster than LC when using more memory, Figure 2.7(b) shows. One important aspect of EPT is that it is easy to configure, and have small construction time while the LC needs an expensive and manual optimization and have a quadratic construction time. Moreover, after reaching the optimum, the LC cannot be improved further by using more memory, while the proposal can use as much memory as it fits in the system's RAM. In Figures 2.8(a) and 2.8(b) it is apparent that EPT has the best performance. Since EPT and EPT* have very similar performance we can choose EPT*.

2.4.3 When the Size of the Database Changes

Now the performance when the size of the database changes is studied. The random database of dimension 16 was used for these experiments. The sizes are $10^5, 3 \times 10^5, 10^6, 3 \times 10^6$, and 10^7 vectors. The LC only appears in the first four benchmarks because with 10^7 elements it took too much time to build. However, we can see that the tendency is clear. In Figure 2.11 each point of the curves correspond, from left to right, to an increasing size of the database. Figure 2.11(a) shows that for a memory-limited setups, the LC is the best but it is surpassed easily by the EPT and EPT* when more memory is used. Note that here the KVP has a similar performance but uses more memory. Next, in Figure 2.11(b) we have the number of distances computed based on the sequential search. Here we see that as the size of the database increases, the percentage of computed distances decreases. Finally, Figure 2.11(c) shows that for big databases, the best choice are the EPT indexes. EPT produces faster indexes than the other methods while using the same amount of memory. When the space is limited, the EPT is beaten just by the optimum LC, but remember that it is simpler to



(a) ***Up to 3 integers per item.***



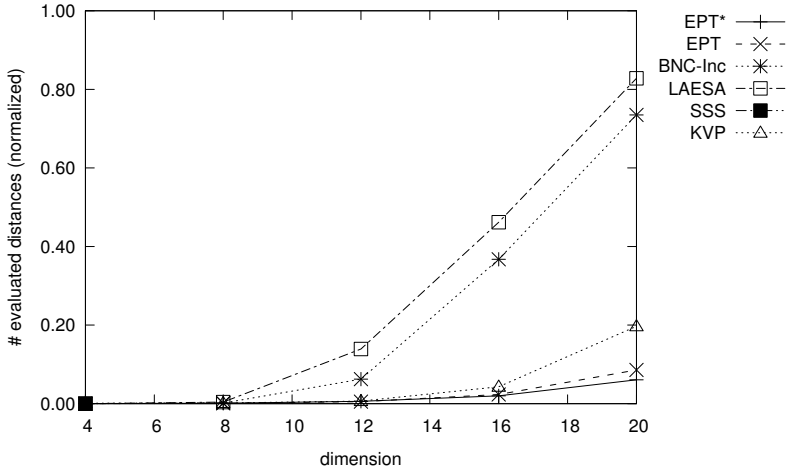
(b) ***From 3 to 10 integers per item.***

Figure 2.7: Number of distance evaluations (normalized) necessaries to solve nearest neighbor searches per dimension. Each figure shows the best indexes on ranges of restricted memory (32 bit integers were used to provide a natural reference). Lower values are better.

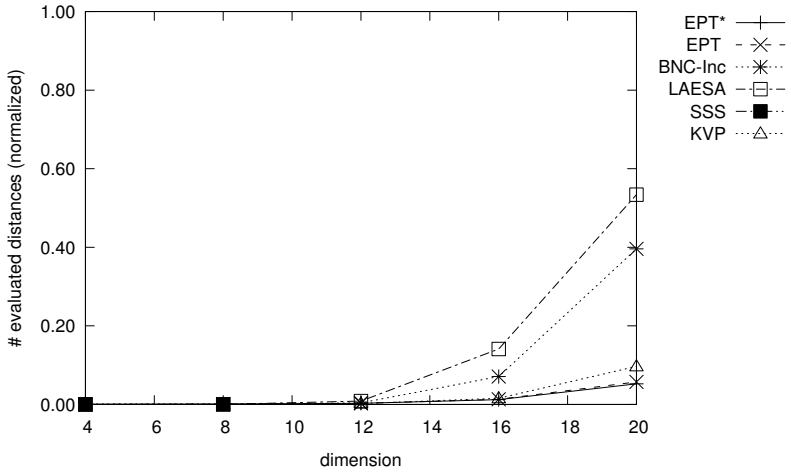
configure and less expensive to construct.

2.5 Summary and Future Work

This chapter describes a novelty metric search technique called Extreme Pivots (EP). EP describes a way to create efficient metric indexes, mainly based on the study of how to relate objects



(a) ***From 10 to 30 integers per item.***

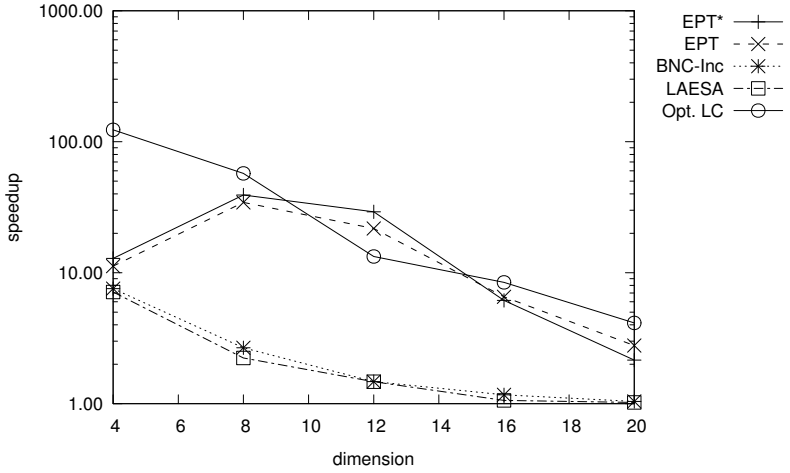


(b) ***From 30 to 100 integers per item.***

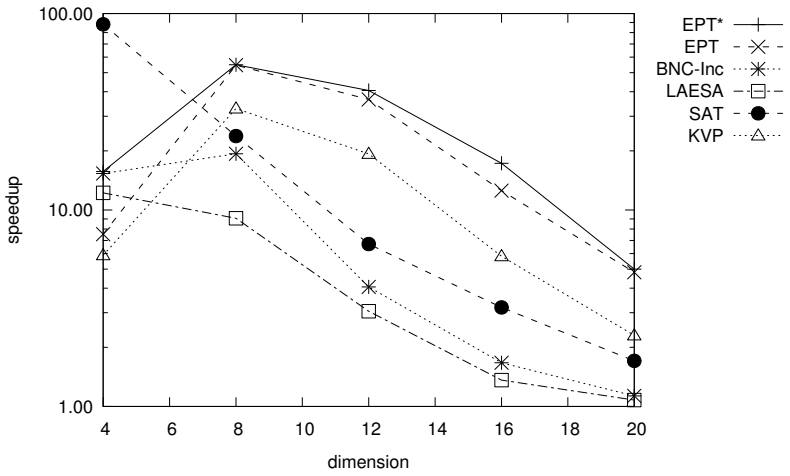
Figure 2.8: Number of distance evaluations (normalized) necessaries to solve nearest neighbor searches per dimension. Each figure shows the best indexes on ranges of restricted memory (32 bit integers were used to provide a natural reference). Lower values are better.

in a dataset with routing objects (pivots). Until now, the research has been devoted to find a set of pivots that best discards items in a dataset for a given query. EP shifts the view of the problem to select the best items for a given pivot. This simple change of perspective allow us to produce small and fast metric indexes. In particular, the implementation was fixed to a table-based index called EP Table (EPT).

Two algorithms to construct EPT indexes were introduced, both of them only need to



(a) ***Up to 3 integers per item.***

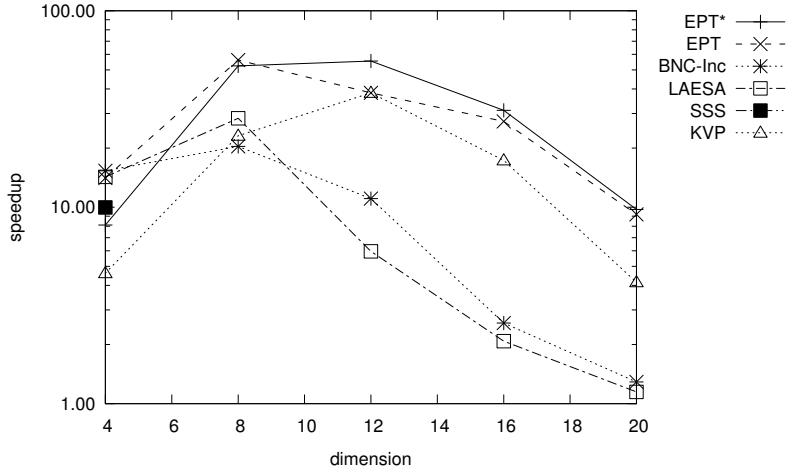


(b) ***From 3 to 10 integers per item.***

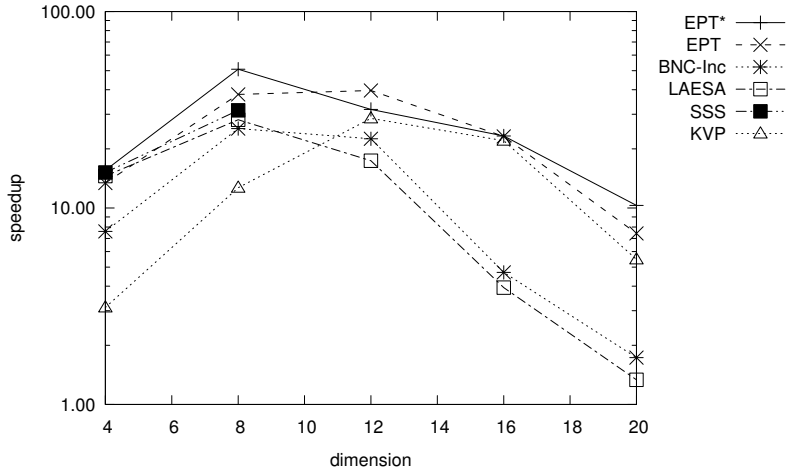
Figure 2.9: Speedup of the nearest neighbor search as compared with the exhaustive scan, the performance is presented per dimension. Each figure shows the best indexes on ranges of restricted memory (32 bit integers were used to provide a natural reference). Higher values are better.

know how much memory should be used in the index structure, the distribution of the objects and queries. The rest of the parameters are adjusted based on this information. Notice that both distributions are estimated from the given dataset.

An EP index is composed of a fixed number of instances (pivot groups). Also, each pivot group uses a small constant of machine words per database element (two or three depending on the width of floating point values). The experiments have shown that this implementation based on



(a) ***From 10 to 30 integers per item.***



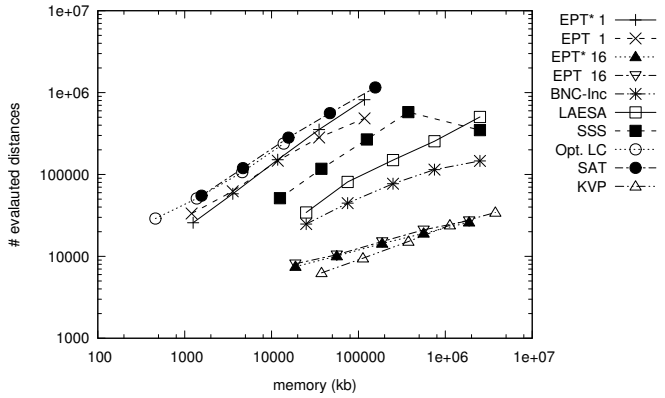
(b) ***From 30 to 100 integers per item.***

Figure 2.10: Speedup of the nearest neighbor search as compared with the exhaustive scan, the performance is presented per dimension. Each figure shows the best indexes on ranges of restricted memory (32 bit integers were used to provide a natural reference). Higher values are better.

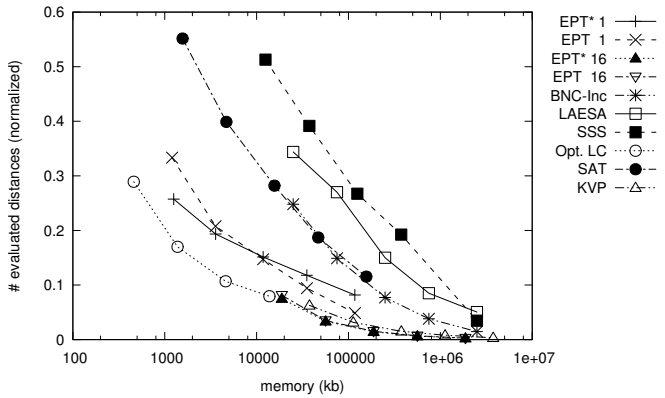
tables surpasses the performance of both pivot tables and compact partitioning indexes, in almost any memory setup.

Nevertheless, EP is not a cure-all remedy for metric indexing. In the one hand, the curse of dimensionality is not avoided, it is only deferred. More detailed, as the dimension increases the index EP needs larger α values. This degrades EP to a sequential search in the limit. On the second hand, even in the practical range of dimension, EP is highly dependent on the quality of

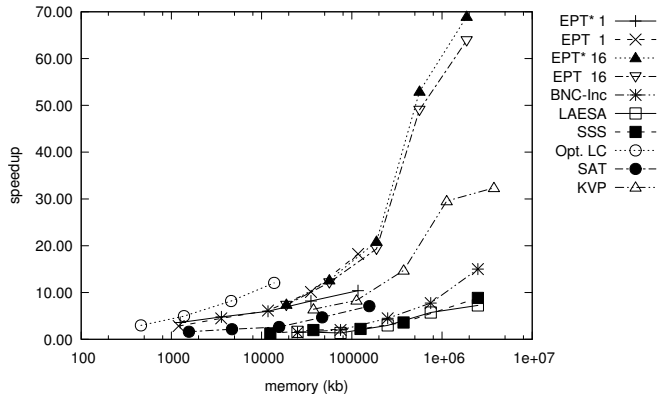
the estimations of σ_X^2, σ_Y^2 and r. So, if the query set does not have a good estimation then the performance will be poor. However, the assumption that the query distribution can be estimated using the database seems reasonable for a broad range of practical applications.



(a) ***Cost***



(b) ***Relative cost***



(c) ***Speedup***

Figure 2.11: Performance on RVEC of 16 dimensions and increasing n, i.e., $10^5, 3 \times 10^5, 10^6, 3 \times 10^6$, and 10^7. EPT and EPT* use 1 and 16 groups, SSS $\varepsilon = 0.4$ with a maximum of 32 pivots, BNC-Inc 32 pivots. LC uses the optimal setup for dimension 16. Each point in a curve corresponds to a value of n in increasing order from left to right.

Chapter 3

Singleton Indexes for Nearest Neighbor Search

The construction algorithm of the EPT can find the right number of pivots in each pivot group based on the distribution of the data, but the theory behind this assumes an independence of the distribution of the data, pivots, and queries which some times is asking too much. So, this chapter presents a meta-heuristic to make the construction of (sub)optimal indexes an automated process that is more robust to the distribution of the data. Again, the available memory is a parameter to control its size.

Remember that there is not a universal complexity model to measure the efficiency of an index. We can compare the number of distance computations needed to solve a search between indexes or we can rank them based on the time they need to find the result elements. The model of using the number of distances has some hidden costs, for example, a sequential scan over the data to filter. Clear examples are the AESA algorithm [Vidal Ruiz86] for exact proximity searching, and the Permutation-Based index [Chavez08] in approximate proximity search. They compute a small number of distances but a sequential scan over the elements is needed in order to know if they can be discarded. Then, the combination of a relatively cheap distance function and high internal cost can lead to a fast index when counting the distance computations, which will be slow in practice.

It is hard to draw a line between exact indexes and those which only accept an approximate solution due to its high intrinsic dimensionality. One of the sources of this ambiguity is the large

number of potential solutions using indexes with claimed low complexity. If a practitioner looks for a solution, the efficiency claims of many papers could be misleading. Those factors will be analysed from this practical perspective.

An additional source of unfairness in the comparison of indexes is the space and the preprocessing time overhead for index construction and maintenance. Some indexes are claimed to be competitive, but the construction cost and/or the space overhead are prohibitive.

The goal now is to produce an index that will be competitive in practice. The requirements will be a cost function, a memory bound and a sample of the query set; then, the output would be the optimized index. In this sense, this approach can be considered as a meta-heuristic for optimal index construction. However, instead of adapting existing indexes to be of use with this technique a family of indexes fitting the new construction scheme was designed. The results are indexes outperforming the state of the art in most of the datasets.

Remember that a metric space is a tuple (U, d) where U is a domain, and $d : U \times U \to \mathbb{R}$ is a *distance function*. Also, the *database* (or dataset) is a finite subset $S \subseteq U$ with $n = |S|$ the number of elements of S.

It is accepted in the literature that the distance axioms are too weak to develop a complete complexity model for metric indexing. The total query time depends on an array of factors that makes parameter tuning difficult and cumbersome. In the absence of a complexity model covering all the relevant aspects, users of this technology have to consider several aspects to make a selection: i) the intrinsic dimensionality and the size of the dataset, ii) the query distribution and search radius, iii) the cost of the distance function, and iv) the underlying computer architecture.

Those factors should be accounted for in a complexity model. The practitioner's ultimate request is to have a smaller (amortized) time for solving a large batch of m queries, as compared to m sequential scans of the database.

3.1 Auto-tuned Nearest Neighbor Index (ANNI)

An index being able to optimize its parameters to best fit a given query sample, a dataset, and a fixed search cost function will be called an *auto-tuned* index.

Suppose we have an index with following operations:

- It accepts incremental improvements.
- After each improvement, it can solve queries.
- The search cost of queries can always be measured.

Under these operations, the construction of an index is an optimization problem. The goal would be to have an optimal index for a large set of unknown queries, having just a sample of the query set. Due to the high cost of construction, backtracking in the index construction will be avoided, and design an incremental process adding more routing objects at each step. In addition, a strong assumption is made: the total search cost is a function *essentially* unimodal and convex. Each step in the optimization will take $O(n)$ distance operations.

A generic sketch of the auto-tuned index construction is as follows: start with the database, a set of queries, and an empty index. One *step* consists of δ *improvements* to the index. From the empty index, improve it δ times, measure the total search cost of the query sample, and repeat until the cost increases.

The above heuristic is suboptimal, because even with the assumed convexity and unimodality, it did not stop at the optimum; which lies between the second-to-last and the last step. In practice, adhering to this suboptimal procedure has a small impact on the total query time. It is worth noticing, however, that backtracking could be done in principle using persistent data structures as described in [Kaplan95].

Notice that the full process depends on the query set. If we have precise knowledge of the query distribution, we will end up with a finely tuned index. From a practitioner's point of view, having a query sample should not be a problem. As a last resort, one can use a sample of the database instead. In the experimental section this last approach was used. The search cost could be the total search time, the number of distances, or a combination of both.

The construction of the index will be discussed in detail later. For now, let us consider the effect of the step size (the number of improvements δ) on the search cost.

3.1.1 Measuring the Search Cost

This method will be optimal on convex cost functions, which have a form depicted in Figure 1.3. Searching for a local minimum, will lead with finding the global minimum in this

circumstances. In this case, the best choice we have is to set $\delta = 1$.

When the cost function is the total search time, the shape will be a convex, unimodal function plus noise. The noise comes from fluctuations of a multi-user OS, the occupancy of the memory hierarchy, data locality and in general some unpredictable small variations. A greedy approach will get stuck in false global minimum because of the noise. This behavior is the reason of introducing the $\delta > 1$ as a parameter. The local minima can be overcome by making δ improvements at once. If δ is set to a relatively large value (e.g. 64), it can surpass the local minima and continue in the search for a faster instance.

In this chapter, the goal is to provide a practical tool for practitioners.The cost will be estimated online using the query set and the tools from the running system. The focus will be for total search time.

3.1.2 Auto-tuning Nearest Neighbor Indexes

Definition 3.1.1 (ANNI Index) *Given a database $S[1,n] = u_1, u_2, \ldots, u_n$ and a set of pivots $\mathbb{P} \subset S$ with size m; an Auto-tuning Nearest Neighbor Index (ANNI) is composed of two arrays: $P[1,n]$ and $D[1,n]$. Those arrays are dynamic, and their content vary during index construction and tuning. P and D are defined as follow:*

- *$P[i] = \mathsf{piv}(u_i)$, the pivot associated to u_i. Some technicalities:*

  i) Objects and indexes will be used indistinctly to describe items, i.e., $\mathsf{piv}(u_1) \in [1..n]$

  ii) Pivots have distinguished entries in P, i.e., $P[i] = 0 \iff u_i \in \mathbb{P}$

- *$D[i] = d(u_i, \mathsf{piv}(u_i))$. Notice that $D[i] = 0$ when $P[i] = 0$.*

- *P and D define a Dirichlet domain, i.e.,*

$$\forall u \in S \setminus \mathbb{P},\ d(u, \mathsf{piv}(u)) = \min_{p \in \mathbb{P}} d(u,p).$$

A range search can be done using ANNI as a pivot table, with one pivot for each database element. Each element u of the database check if pivot $\mathsf{piv}(u)$ can discard it using the pivot rule, if

it cannot, then the distance between the query and u must be computed and, u could be eventually reported in the result. This is formalized in Algorithm 4.

Algorithm 4 Searching with an ANNI index.

Input: The database $S[1,n] = u_1, \ldots, u_n$, the ANNI index (P, D), see Definition 3.1.1, and the query (q, r)

Output: The result set R satisfying the query

```
1:  Let H(a) be a cache map storing the distance among q and a
2:  Populate H with ℙ
3:  for i = 1 to n do
4:     Define d_pq = H(P[i]) {The distance between the covering pivot and query}
5:     if D[i] = 0 then
6:        if d_pq ≤ r then
7:           R ← R ∪ {(d_pq, S[i])}
8:        end if
9:     else if |d_pq − D[i]| ≤ r then
10:       Define d_i = d(q, S[i])
11:       if d_iq ≤ r then
12:          R ← R ∪ {(d_iq, S[i])}
13:       end if
14:    end if
15: end for
```

Remember that the cost of an index is the sum of the internal and external cost. The internal cost of ANNI is m and the external is $n\gamma$ where γ is the discarding probability of the index. The construction of ℓ ANNI indexes is done starting by adding δ pivots to an empty ANNI and distribute the points in a Dirichlet domain. Then estimate γ by performing the set of queries and taking their average cost. Finally, estimate the cost of ℓ ANNI's using $cost = m\ell + n\gamma^{\ell}$. Repeat this until the current cost is worst than the previous one. This is shown in Algorithm 5.

Definition 3.1.2 (NANNI) *A Node Auto-tuning Nearest Neighbor Index (NANNI) is an ANNI implemented with clusters. Each database object has its associated cluster center or pivot. Given a database $S[1,n] = u_1, u_2, \ldots, u_n$ and $\mathbb{P} = p_1, p_2, \ldots, p_m$ a NANNI index is composed by two functions: items and cov:*

- $\mathit{items}(p) = \{(u, d(u, \mathit{piv}(u))) | \, \mathit{piv}(u) = p\}$.
- $\mathit{cov}(p) = \max_{(u,d_u)\in \mathit{items}(p)} \{d_u\}$, *this is called the* covering radius *of* p.

Algorithm 5 Construction of an ANNI Index

Input: The database $S[1,n] = u_1, \ldots, u_n$, a training query set Q, the number of sister indexes ℓ, the step's length δ

Output: The new ANNI index $(P[1,n], D[1,n])$

Let $prev\_cost \leftarrow n+1$ and $curr\_cost \leftarrow n$
Let $H(a,b)$ be a cache map for storing the distance among a and b, if pair (a,b) is undefined then it computes $d(a,b)$, also, it adds the new pair and distance to H.
Initialize P and D using a pivot randomly selected from S.
while $prev\_cost > curr\_cost$ **do**
  $prev\_cost \leftarrow curr\_cost$
  for $i = 1$ to δ **do**
    Randomly select pivot id c from $[1..n]$
    $P[c] = 0$, $D[c] = 0$ {Tag c as pivot}
    for $j = 1$ to n **do**
      if $P[j] > 0$ and $H(S[c], S[P[j]]) \leq 2D[j]$ **then**
        Define $d_j = d(S[c], S[j])$
        if $d_j < D[j]$ **then**
          $D[j] \leftarrow d_j$
          $P[j] \leftarrow c$
        end if
      end if
    end for
  end for
  $cost \leftarrow$ average of the cost of searching all $q \in Q$, using Algorithm 4.
  Let $\gamma = (cost - m)/n$
  $curr\_cost \leftarrow m\ell + n\gamma^{\ell}$
end while

- *As ANNI, NANNI also defines a Dirichlet domain.*
- *The search algorithm is similar to ANNI, however since it has clusters, it has the following modifications:*
  - *i) The search algorithm is modified to prioritize closer pivots to the query q.*
  - *ii) For each pivot p, its region can be safely discarded when $d(q,p) > cov(p) + r$ (ball condition), see Figure 3.1.*
  - *iii) Also, the hyperplane condition could help to discard a region when $d(q,p) > d_{nearest\text{-}p} + 2r$, where $d_{nearest\text{-}p}$ is the distance to the closer pivot to q, see Figure 3.2.*

NANNI can be constructed processing a working ANNI (Algorithm 5), just fusing data into clusters to efficiently get items and cov. Or iteratively, a construction algorithm randomly promotes objects to pivots, then each new pivot will steal objects from other pivots to maintain a valid Dirichlet domain until the cost of the queries gets worst (Algorithm 6). A NANNI where the cost function is the time will be called TNANNI.

Algorithm 6 The construction of NANNI.

Input: The database $S[1,n] = u_1, \ldots, u_n$, the set of training queries Q, the length of the steps δ, and the function cost f to minimize.
Output: The new NANNI ($\mathbb{P}$, items, cov).

1: Set prev_cost $\leftarrow n+1$ and curr_cost$\leftarrow n$.
2: Set $\mathbb{P} = \{p_1\}$ for some random object p_1.
3: Populate items(p_1) with all the objects of S.
4: Compute cov(p_1).
5: **while** curr_cost $\leq$ prev_cost **do**
6: prev_cost $\leftarrow$ curr_cost
7: Add δ items to $\mathbb{P}$ and arrange all the objects on items$(\mathbb{P})$ to maintain a Dirichlet domain. Also update all the cov's.
8: Search for all the queries in Q and evaluate the cost function f.
9: Store the average cost of the queries in curr_cost.
10: **end while**

Definition 3.1.3 (MANNI) *A Multiple-ANNI (MANNI) M is defined as a collection of ℓ ANNI's, $M = I_1, \ldots, I_\ell$. In particular, I_1 is called the* leader *and is a NANNI index, the rest are plain ANNI's.*

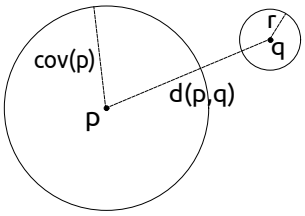


Figure 3.1: If the covering ball of a node p does not intersect the query ball then all points of this node can be discarded.

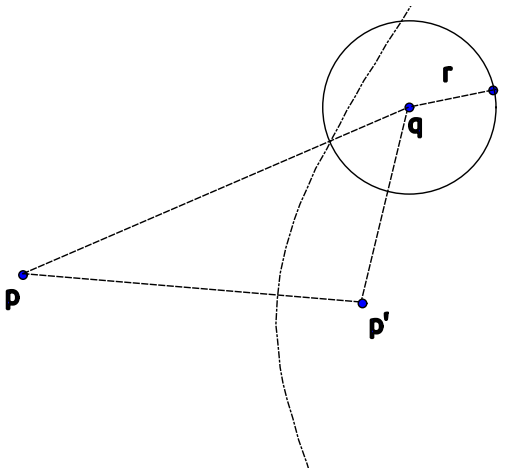


Figure 3.2: The $d(q,p) > d(q,p') + 2r$ criterion (where p' is the closest node to q) says that if the query is on the right side of the hyperbola, the points of p can be discarded.

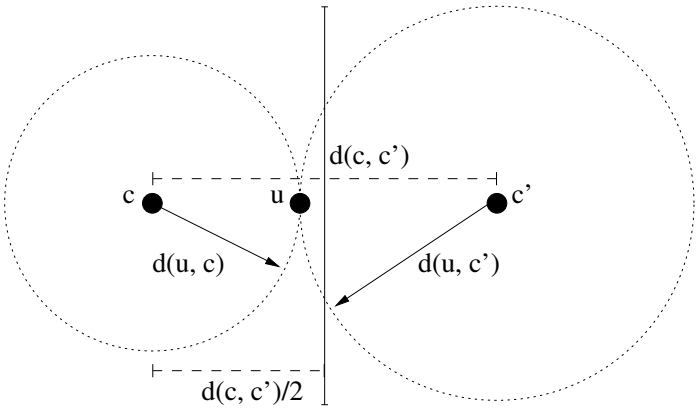


Figure 3.3: Stealing algorithm. Pivot c' can steal u from c when u is located at the right side of the hyperplane.

– *Each one of the ℓ ANNI's is constructed independently, using Algorithm 5 reporting ℓ instances. The leader is then converted to a NANNI index.*

– *Algorithm 7 describes the searching procedure, coordinating both NANNI and ANNI indexes. First, the NANNI tries to discard cells using the covering radius and the hyperplane condition, then, the points in cells not discarded are analysed using the pivot rule, if a point still can not be discarded, the ANNI's use the pivot rule to try to discard it. If, after all this, the point could not be discarded, it is compared with the query.*

– *Since MANNI is composed of several indexes, and those indexes have functions and internal structures, when necessary it will be added a subscript to the names to clarify the target index. For example, suppose we have indexes A and B, the notation changes to items_A, cov_B, D_A, P_B, etc.*

Definition 3.1.4 (DMANNI and TMANNI Indexes) *DMANNI is a MANNI index designed to reduce the number of distance computations. As MANNI, it is composed of a collection of one NANNI and some ANNI instances. A DMANNI works as follows:*

– *The search algorithm is the same as MANNI.*

– *The construction is described in Algorithm 8. Notice how pivots are promoted in all instances before measuring the search cost. This contrasts with the independent construction and estimation policy of MANNI.*

Algorithm 7 Searching with a MANNI index.

Input: The database $S[1,n] = u_1, \ldots, u_n$, the MANNI index $M = I_1, \ldots, I_\ell$, (see Definition 3.1.3) and the query (q, r)

Output: The result set R satisfying the query

```
1: Let H(a) be a cache storing distances between q and a
2: Populate H with the distances from q to all pivots in I_1, ··· , I_ℓ
3: Select ℙ from I_1 and order the pivots with respect to their distance to q
4: Compute d_nearest-p as the smallest distance from a pivot in ℙ to q
5: for p ∈ ℙ do
6:    Define d_pq = H(p)
7:    if d_pq ≤ cov_{I_1}(p) + r and d_pq ≤ d_nearest-p + 2r then
8:       for (i, d_pi) ∈ items_{I_1}(p) do
9:          if |d_pq − d_pi| ≤ r then
10:            for j = 2 to ℓ do
11:               Try to prove in I_j that d(q, S[i]) > r using the same rules than Algorithm 4 {This
                  does not evaluate d(q, S[i]))}
12:            end for
13:            if there is no proof that d(q, S[i]) > r then
14:               Define d_iq = d(q, S[i])
15:               if d_iq ≤ r then
16:                  R ← R ∪ {(d_iq, S[i])}
17:               end if
18:            end if
19:         end if
20:      end for
21:   end if
22: end for
```

Algorithm 8 Auto-tuning of a DMANNI Index

Input: The database $S[1,n] = u_1, \ldots, u_n$, a training query set $Q[1,m]$, the number of indexes ℓ, the step's length δ

Output: The new DMANNI index as a collection of one NANNI and $\ell - 1$ ANNI instances

1: Create one empty NANNI and $\ell - 1$ empty ANNI.
2: Let $prev\_cost \leftarrow n + 1$ and $curr\_cost \leftarrow n$
3: **while** $prev\_cost > curr\_cost$ **do**
4: $prev\_cost \leftarrow curr\_cost$
5: **for** $i = 1$ to δ **do**
6: **for** $I \in$ DMANNI **do**
7: Promote an item to pivot and re-arrange objects to maintain a Dirichlet domain.
i) For NANNI it implies to apply the stealing pivot procedure as described in Definition 3.1.2.
ii) ANNI will need to apply the procedure starting at line 7 of Algorithm 5.
8: **end for**
9: **end for**
10: Search all $q \in Q$ using Algorithm 7 and store the cost.
11: $curr\_cost \leftarrow$ average of the cost of the searches.
12: **end while**

TMANNI is identical to DMANNI but it optimizes the total search time instead of the number of distance computations. It is a practical way to incorporate all the hidden costs in the operating system and the index's internal complexity. It is straightforward to adapt Algorithm 8 to measure the cost as real time.

3.1.3 Analysis of the Search Cost

The ANNI uses the separation in classes of the Dirichlet domain and the verification of the pivot methods, i.e., a point u is discarded when

$$|d(u, c_u) - d(q, c_u)| > r$$

for a query q with radius r where c_u is the center of the class of u. It has a cost $nS_A + m$ where n is the size of the database, m is the number of centers and S_A is the probability for the ANNI to discard a point. Now, S_A is analysed. Let X_{c_i} be the random variable $X_{c_i}(u) = d(c_i, u)$ for u in the class of the center c_i, $Y_{c_i}(q) = d(c_i, q)$. It will be assumed that X_{c_i} and Y_{c_i} are independent

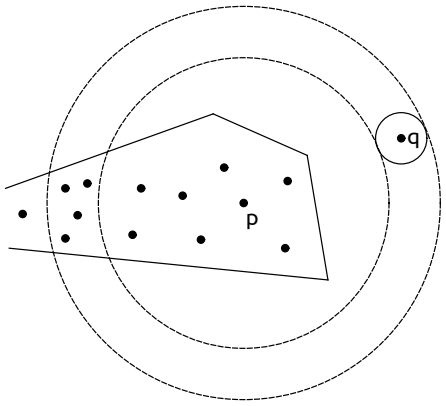


Figure 3.4: If a center cannot discard a group of points, no other center can and those points must be manually checked.

identically distributed random variables (i.i.d.r.v.). If X_{c_u} is the random variable of the class of u then

$$S_A = Pr(u \text{ is discarded}) = Pr(|X_{c_u}(u) - Y_{c_u}(q)| > r).$$

This probability is similar to the pivot based algorithms analyzed in [Chavez01]; however, the difference is that X_{c_i} is defined only for those elements associated to c_i. Therefore, the elements in this class can only be discarded by c_i. Since the pivot covering an object is not random, they are close, then the probability of discarding a covered object is larger, although there is only one pivot to cover an object; and hence the probability S_A is smaller than the random pivot table discarding probability. This is illustrated in the example in Figure 3.4. At first sight, this would make ANNI less competitive than pivots; however, it can use more pivots as MANNI does.

It is well known that pivots near and far from a query are more likely to discard objects [Celik02a, Celik02b, Ruiz13]. In this case, just the near pivots are stored. The whole picture is that both the external and the total number of evaluated distances, at a minimum increasing in the internal cost.

The NANNI is similar but has two extra conditions. First, it checks whether $d(c_i, q) > \mathsf{cov}(c_i) + r$ where $\mathsf{cov}(c_i)$ is the maximum distance from the center c_i to the elements on the cell. If the expression is true, it can discard the entire cell of c_i. If not, we verify if $d(c_q, q) < d(q, c_i) - 2r$ where c_q is the cell of q, to see if we can discard the cell of c_i. Finally, if both tests fail, we use the ANNI criterion. Note that the latter condition implies the first, and hence the first condition does

not affect the probability of discarding. So, we have

$$S_N = Pr(u \text{ is discarded}) = Pr(d(c_q, q) < d(q, c_u) - 2r \cup |d(u, c_u) - d(q, c_u)| > r) =$$

$$Pr(Y_{c_q}(q) < Y_{c_u}(q) - 2r \cup |X_{c_u}(u) - Y_{c_u}(q)| > r)$$

which makes the NANNI to have a smaller external cost.

The MANNI is a combination of a NANNI and multiple ANNI's. The cost is $n(S_N * S_A^{\ell-1}) + \ell m$. Remember that the ANNI and NANNI (and in consequence MANNI) have the properties required for using the auto-tune method. In the case of the DMANNI (and TMANNI), the NANNI and each ANNI could have been individually auto-tuned, but the DMANNI is seen as a whole. The auto-tuning method is a meta-heuristic and uses the indexes as black boxes.

3.2 Experimental Results

The experimental setup is standard for testing indexes. A common dataset is used for all the indexes, the algorithms run on the same machine, implemented in the same language. The first part corresponds to real world datasets, the second part corresponds to random vectors on the unitary cube to test the dependency on the dimension and the database size.

3.2.1 Description of the Experiments

The databases used in the experiments were Nasa, Colors, and, English Wiktionary. Also, randomly generated databases (RVEC) of several dimensions and sizes were used. The English Wiktionary database is a dictionary with $736,639$ entries with Levenshtein's distance as metric. A sequential search completes in 0.940 seconds.

Each plot depicts the average of 256 nearest neighbor queries. Query objects were not indexed. Along with this contribution (NANNI, TNANNI, MANNI, DMANNI, and TMANNI) we used several state of the art metric indexes, as well as canonical representatives of the three known discarding rules (i.e. pivots, Dirichlet domains and clusters) as baseline for the comparison.

In all cases, the ANNI indexes were created using 64 random elements of the database as the training query set, fixing the step σ to 128 for NANNI and TNANNI (see Algorithm 5), and

512 as a sum of all steps in the individual instances for MANNI, DMANNI, and TMANNI (see Algorithm 8).

3.2.2 Index Comparison

The comparison method consists in contrasting all the indexes using a single dataset and iterate over several representative datasets. The experiments will show that the presented self-optimized indexes are faster than the state of the art alternatives. Incidentally, they help to clear the notion that reducing the number of distance computations leads to faster indexes, showing experimental evidence that this is only valid when the complexity of the indexes is low.

Colors Figure 3.5(a) shows in the x-axis the size of the index in memory and in the y-axis the number of distances computed as a fraction of the database size (distances/size). Here the LC, NANNI and TNANNI have the smallest memory footprint. They also compute more distances than most of the other indexes. If we consider only this measure we will arrive at the misleading conclusion that those other indexes are better. However, when comparing these results with the plot in Figure 3.5(b) size vs. speedup, the NANNI and TNANNI stand apart being faster than most of the others. This is one of the cases when a ”bad”index in terms of number of distances computed, is the fastest. As had been mentioned, this is because of various factors that depends of the metric space and/or the index itself. We should not focus on just how many distances an index computes, performance metrics should be more inclusive.

Comparing just the number of computed distances is a necessary abstraction for a more comprehensive analysis, and has its theoretical importance. However, the inner complexity of the indexes (requiring for example a sequential scan of the database with a cheaper filtering procedure) make them unsuitable from the perspective of a practitioner. One extreme example is SSS, it computes a very small number of distances but it is the slowest and also is the one that needs more space. This indicates that it has a lot of pivots that does not help for discarding an element. The pivots really needed are just a few. Compare the SSS with the EPT for example, some EPT have the same number of distances computed but are smaller and faster, that means the EPT have better pivots that boost the searches. So, less space and more speedup indicates better pivots, this implies that the pivots from NANNI and TNANNI are very efficient. That is because they have the *right*

number of pivots.

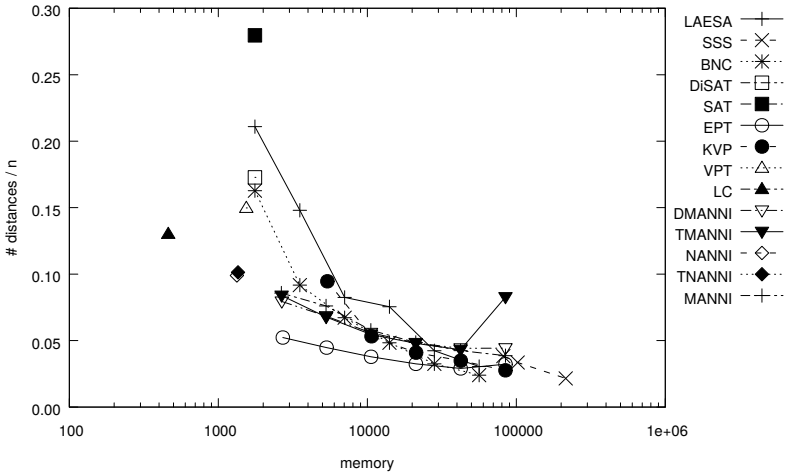
Seeing the experimental time results, we realize that the MANNI, the TMANNI, and DMANNI, have similar performance among them. As the amount of memory increases, the searches become slower even when the number of distances computed is decreasing. This is because the indexes have too many pivots. That happens very early in this database because it is small. This behavior is analyzed with more detail in the next databases of larger size. It is also worth noting that, at same memory usage, the presented methods are very competitive with a performance similar to the EPT and KVP in number of distances and are usually faster. That becomes apparent in the speedup plot (Figure 3.5(b)).

The LAESA, SSS, and BNC are left behind in this and all the other tests. The SAT's only advantage is the space, it needs less memory than most of the others (but more than the NANNI and TNANNI). The space advantage becomes a drawback because there is no mechanism to allow SAT to use more space. The DiSAT computes less distances than the simple SAT and is faster but even the DiSAT ranks below ANNI indexes in this and the other experiments. The VPT needs to compute many distances and its speedup is low in this and all the tests except with the larger dataset.

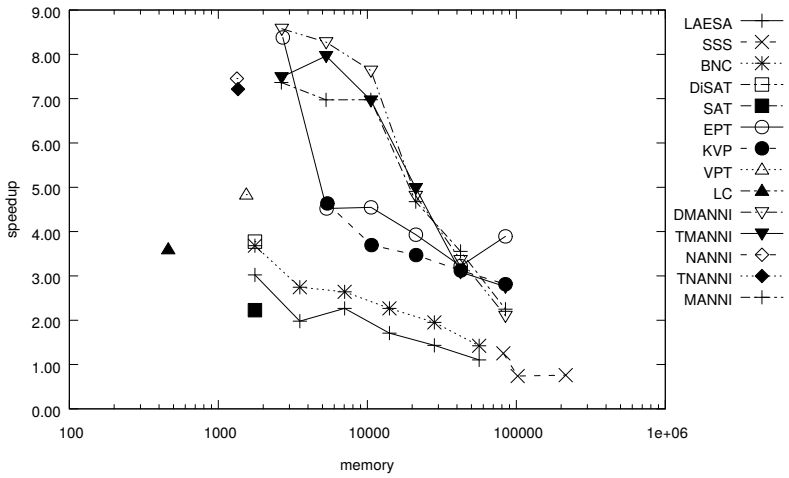
The special case of the LC is left to discuss at the end of the experiments.

Nasa The format of the experiment is the same as before. The results are summarized in Figure 3.6. Again, the NANNI and TNANNI are not the best when comparing the number of distances computed, but, they are much faster than almost all of the others. The MANNI family has similar performance and they are the fastest when compare to the other methods of same size. Notice that even if the computed distances for TMANNI increase, its search time is very close to the others.

Wiktionary In this database something different happened. The edit distance have quadratic complexity in the word's size and the number of computed distances dominates the total search time. The LC is the fastest and uses the smallest amount of memory (as discussed, this result should be taken with caution, since tuning the LC and the construction time makes it impractical, see Table 3.1 in Section 3.2.4). In Figure 3.7 the NANNI is small, computes a tiny number of distances, and is very fast. On Figure 3.7(b) the DMANNI and TMANNI have a similar tendency with the DMANNI being faster. The MANNI is a little different because the construction method assumes the i.i.d.r.v.

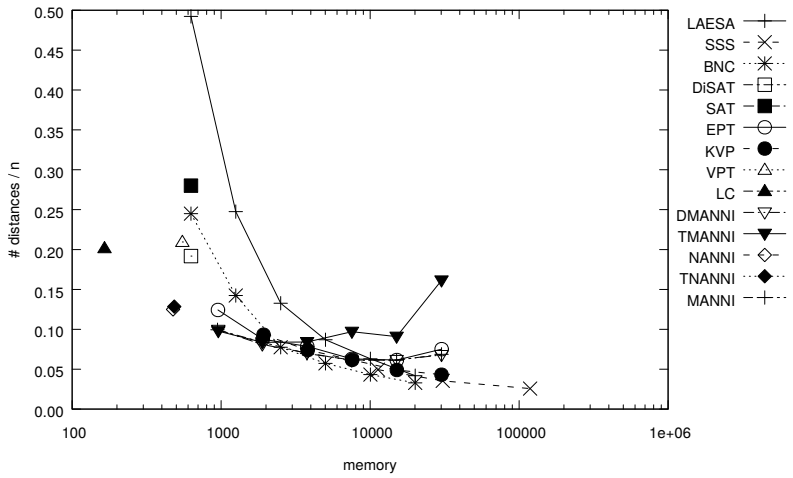


(a) ***Number of computed distances, bottom-left corner is better***

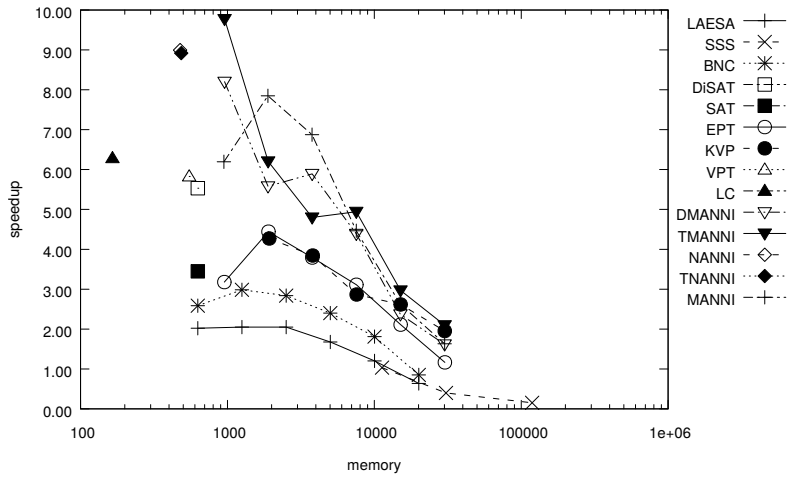


(b) ***Speedup as compared with sequential search, top-left corner is better***

Figure 3.5: Performance comparison among the ANNI indexes and the state of the art alternatives over Colors database.



(a) ***Number of computed distances, bottom-left corner is better***



(b) ***Speedup as compared with sequential search, top-left corner is better***

Figure 3.6: Performance comparison among the ANNI indexes and the state of the art alternatives over Nasa database.

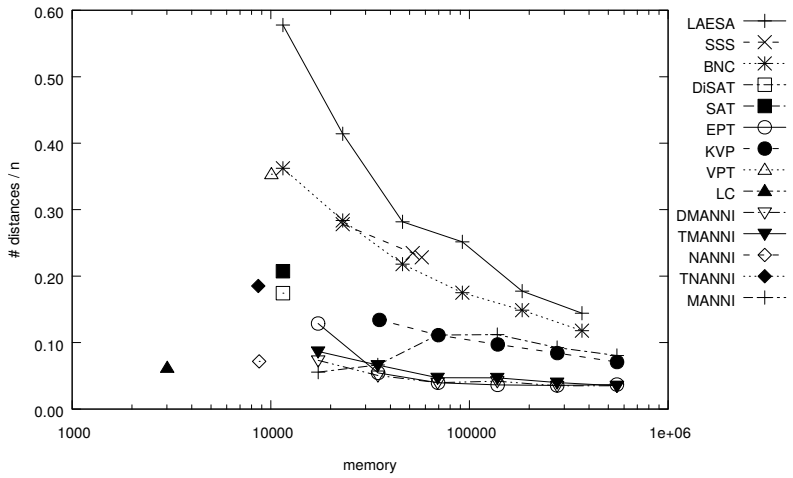
for the data. It reaches the faster speed with small memory. The only index that compares them is the ETP. Here, the VPT is specially affected in the time search because of the high number of costly distances it needs.

Random Vectors 12 coordinates In this database the objective is to test an instance of known dimension, in this case is a 12 dimensional dataset. We can observe in Figure 3.8 that most of the indexes are comparable in number of computed distances with the NANNI and TNANNI being ordinary. In contrast, for the speedup, in Figure 3.8(b), we can see that the NANNI and TNANNI are extraordinary, also DMANNI is the fastest comparing with the others of similar space. The only competition at hig memory is with the EPT that keeps a consistent performance. This is not a surprise because those methods want to construct the optimal index, the advantage of the ANNI construction is that it is empirically tested.

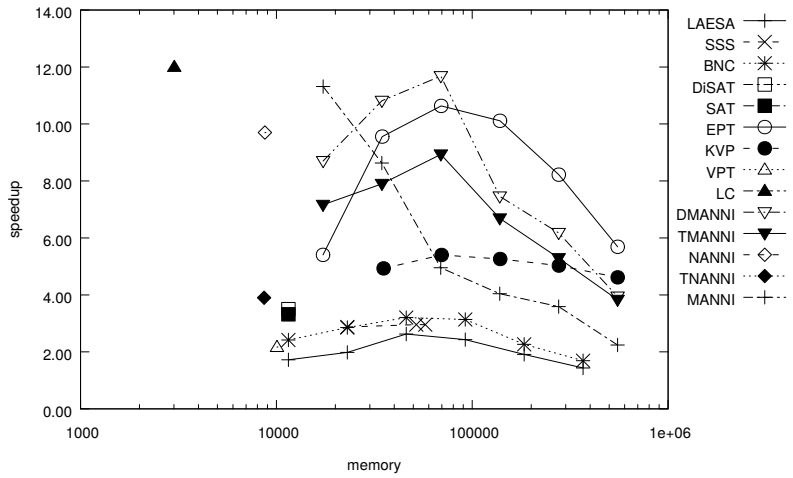
Summary In all the experiments we can see that NANNI is consistently fast, with a small memory footprint.

3.2.3 Performance when Dimensionality Grows

The curse of dimensionality is a well known phenomenon limiting the performance of metric indexes. As the dimension increases, the relative gain in speed is shattered and it is even possible that an indexed search become more costly than a sequential search. The database size was fixed to one million objects and vary the dimension. The databases used were randomly generated in the unitary cube. The dimensions tried were 4, 8, 12, 16, 20, and 24. The results were grouped by space usage. First, in the smallest indexes, they take 10 or less integers per element in the dataset (Figure 3.9(a)). In this setup we see that the best choices are NANNI and TNANNI. Next, the indexes increased the size to take between 10 and 30 integers per item (Figure 3.9(b)). Here, the MANNI family is faster. Finally, allowing between 30 and 100 integers we have the second largest instances (Figure 3.10(a)), and allowing up to 300 integers per item we have the largest instance (Figure 3.10(b)). The result is the same: the MANNI family had a clear dominance on small dimensions. Here, the ANNI's empirical construction takes a big advantage because of the nature of the dataset: since the training queries are similar to the actual queries, the resulting index

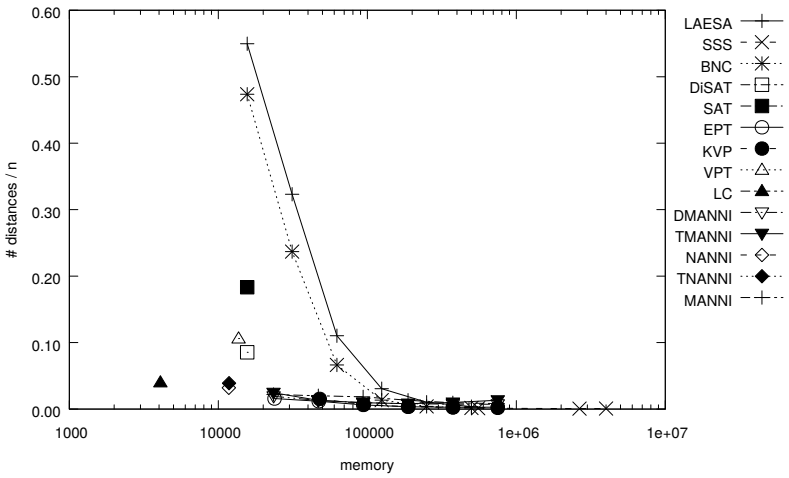


(a) ***Number of computed distances, bottom-left corner is better***

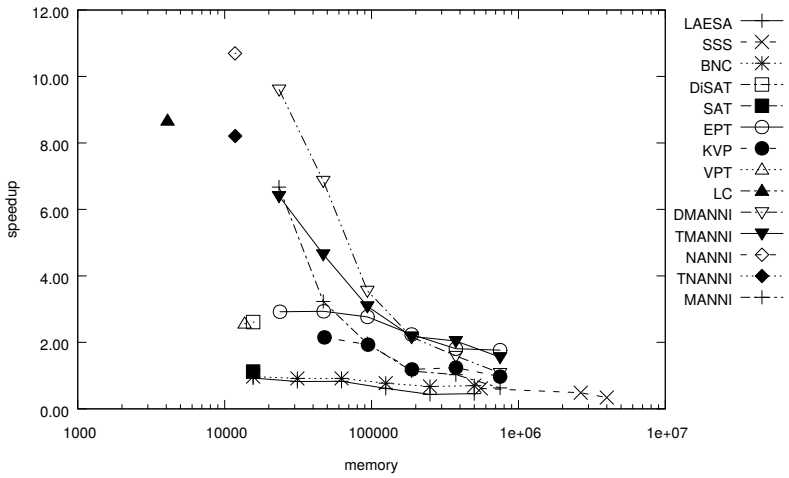


(b) ***Speedup as compared with sequential search, top-left corner is better***

Figure 3.7: Performance comparison among the ANNI indexes and the state of the art alternatives over the Wiktionary database.



(a) ***Number of computed distances, bottom-left corner is better***



(b) ***Speedup as compared with sequential search, top-left corner is better***

Figure 3.8: Performance comparison among the ANNI indexes and the state of the art alternatives over the RVEC-12-1M database.

is very close to the optimal. For large dimensions, all the methods collapse and are equally bad; this is a known effect of the curse of dimensionality.

The set of plots in Figure 3.11 and 3.12 shows the number of distances grouped in the same way as before. Here, we can see what happen with the number of distances when the dimension increases. The Figure 3.11(a) shows that the ANNI have the smaller increase. On figures 3.9(b) and 3.12(a) we see that the EPT is the best, very close followed by DMANNI and MANNI. In Figure 3.12(b) we note that the DMANNI surpasses the EPT.

3.2.4 Performance when n Grows

An interesting case is when all the parameters but the size of the dataset are fixed. The next set of results show the behavior of the indexes for increasing the database size. The exact sizes used were $10^5, 3 \times 10^5, 10^6, 3 \times 10^6, 10^7, 3 \times 10^7$, and 10^8.

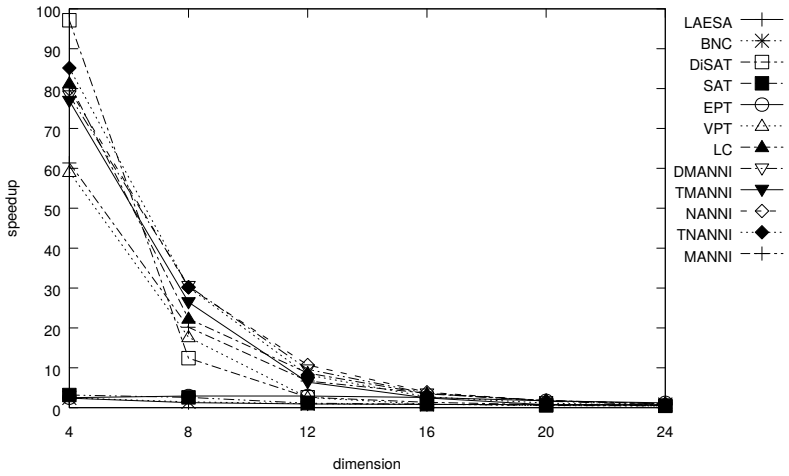
Medium size databases The results are grouped, as before, by the amount of space of the indexes for the first 4 sizes of the datasets. In figures 3.13(a) and 3.15(a) we have the speed and the number of distances for the smallest indexes. We can see that the NANNI is the fastest and all the ANNI indexes have a much better speed than the others. Also they are really competitive in computed distances.

We can also see Figures 3.13(b) and 3.15(b) where the DMANNI and TMANNI are the fastest with the MANNI on third place followed closely by the EPT. The MANNI family is also very competitive in the number of distances computed.

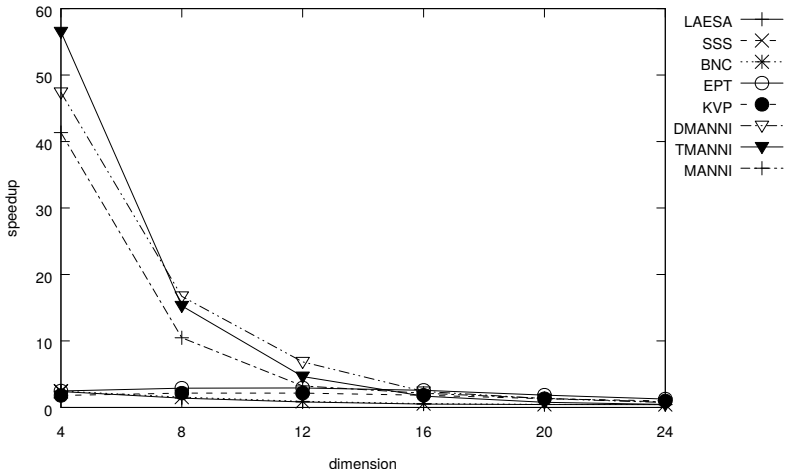
Now, on figures 3.14(a) and 3.16(a) the DMANNI and TMANNI are again the fastest, with the EPT catching up. Note how these two indexes are very fast although they compute more distances than the EPT, KVP and BNC.

Finally, on the Figure 3.14(b) note how the EPT finally becomes the fastest index followed by the TMANNI but they are not very different from the sequential search. Look how in Figure 3.16(b) the other indexes compute less distances than the MANNI family. Not also that the TMANNI computes more distances than the others but is one of the fastest.

The experimental results show how the search cost can diverge as the units of measure, i.e. the number of distances computed or the number of elapsed seconds. So, we cannot optimize

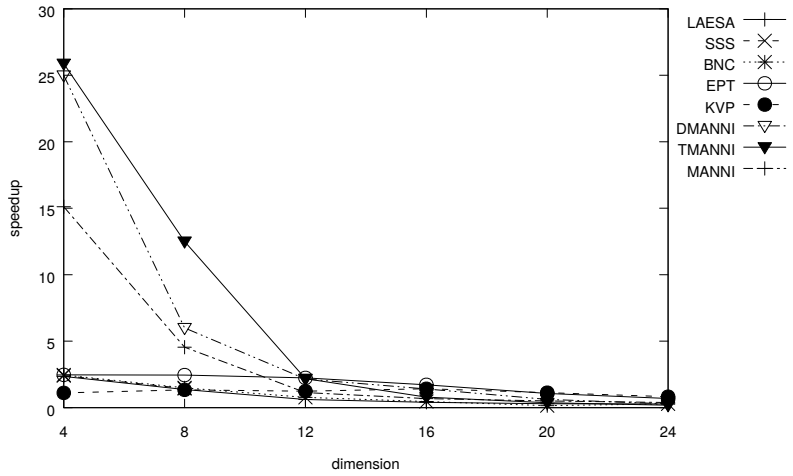


(a) ***up to*** 10 ***integers***

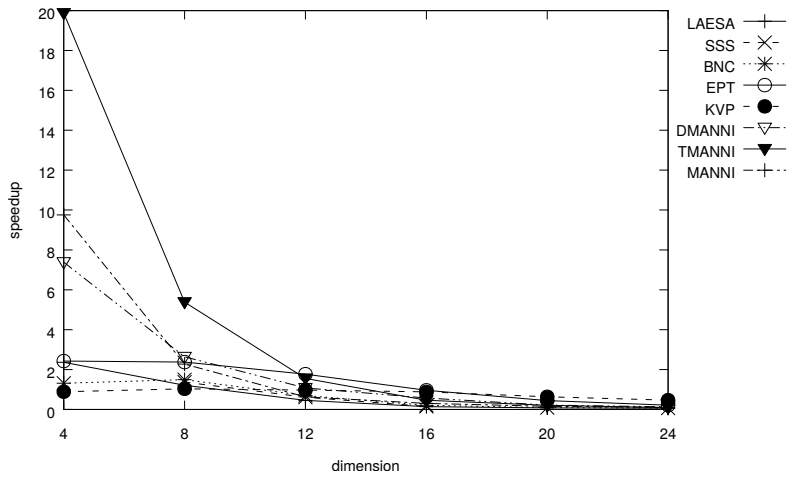


(b) 10 ***to*** 30 ***integers***

Figure 3.9: Search speedup for increasing dimensionality, higher is better. The indexes are segmented into classes of memory usage. One million RVEC datasets.

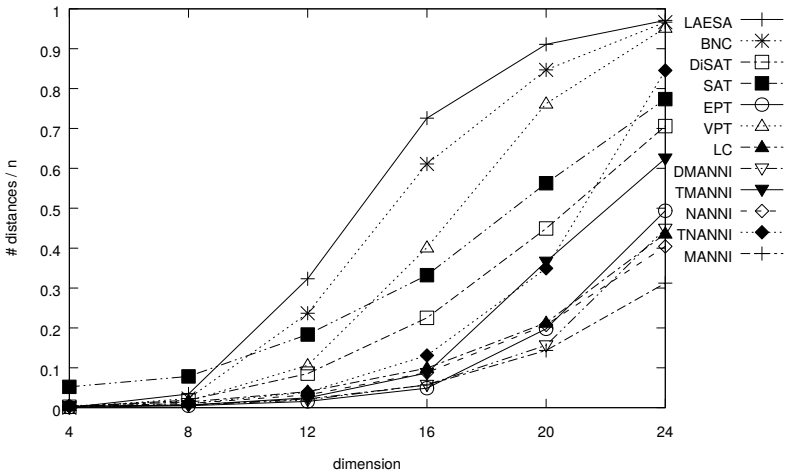


(a) 30 ***to*** 100 ***integers***

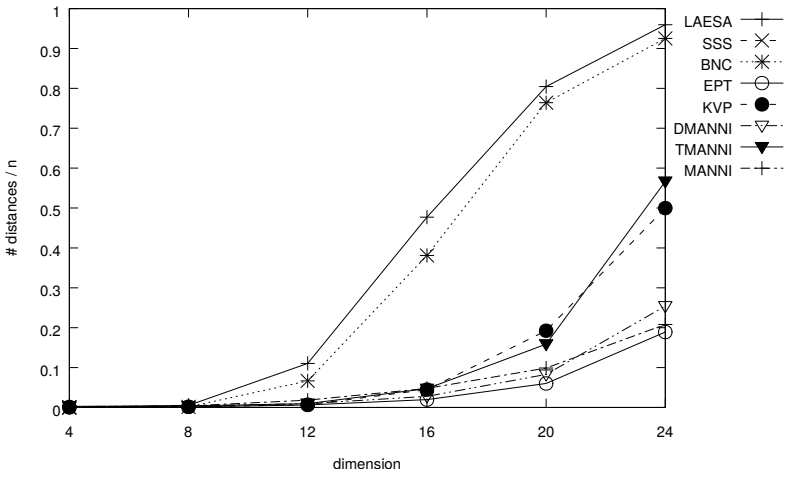


(b) 100 ***to*** 300 ***integers***

Figure 3.10: Search speedup for increasing dimensionality, higher is better. The indexes are segmented into classes of memory usage. One million RVEC datasets.

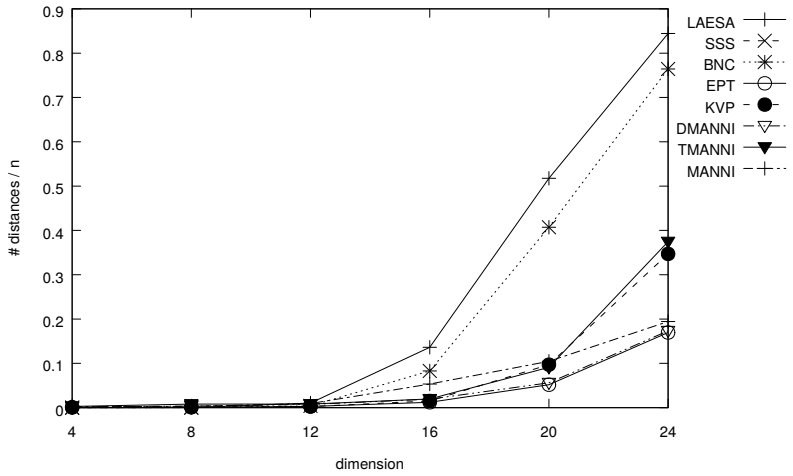


(a) ***up to*** 10 ***integers***

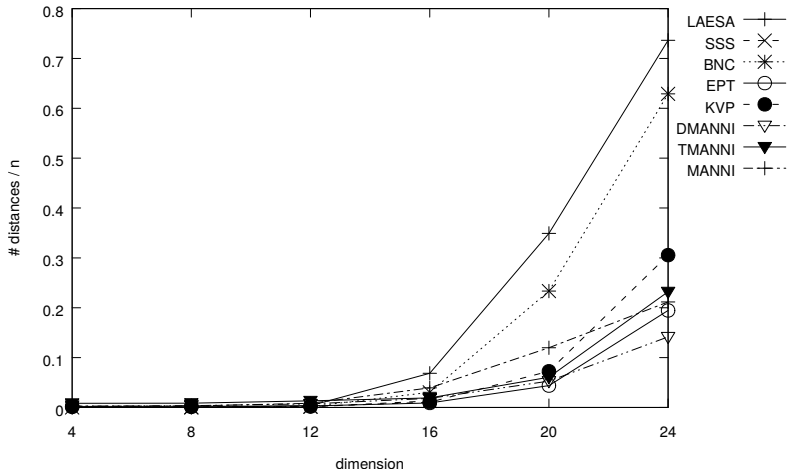


(b) 10 ***to*** 30 ***integers***

Figure 3.11: Search cost measured as the number of distance evaluations for increasing dimensionality (RVEC-*-1M), lower is better. The indexes are segmented into classes of memory usage.



(a) 30 ***to*** 100 ***integers***



(b) 100 ***to*** 300 ***integers***

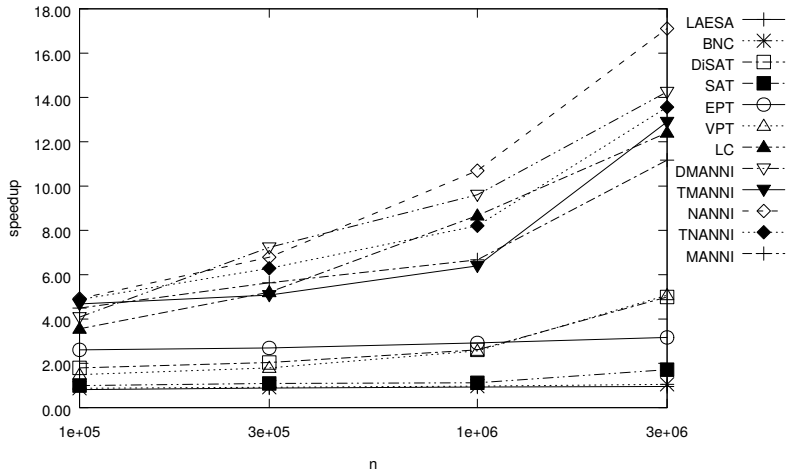
Figure 3.12: Search cost measured as the number of distance evaluations for increasing dimensionality (RVEC-*-1M), lower is better. The indexes are segmented into classes of memory usage.

for one unit and expect to work in an another, in general. Fortunately, indexes designed to have low complexity in both measures, like *ANNI family, have both setups close to the optimum; therefore, it is possible to optimize for distances and work in real time. This is a good idea whenever is possible since measuring elapsed time can be affected by a lot of factors in real-world computer systems.

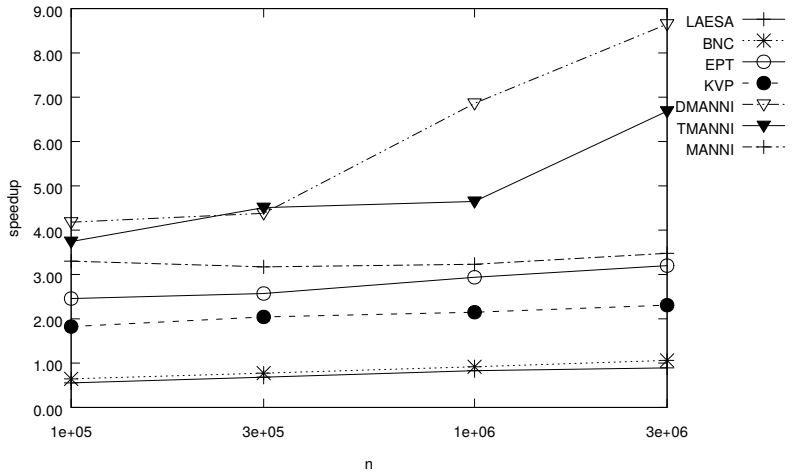
Large databases Results for $3 \times 10^6, 10^7, 3 \times 10^7$, and 10^8 sized databases. Figure 3.17 shows the cost of the queries for the indexes having at most 12 integer per element. The number of distance computations is shown in Figure 3.17(a), here we see that the EPT is the best followed very closely by DMANNI, also TMANNI, NANNI, and MANNI are close. All the indexes exhibit a linear grow. For the speedup the results are different, this is shown in Figure 3.17(b). The NANNI clearly separates from the others. It is the fastest and has a non linear growth. This is because in the search of the NANNI is no sequential scan of the objects to try to discard them. Also, the DMANNI is very good, only surpassed at the end by the VPT. The VPT, that had a rather poor performance in all the other experiments, finally shines because of the combinations of its simple structure, the very high size of the database and a simple distance function.

With all this data, the conclusion is that the MANNI family is the best choice among the other indexes of similar memory, closely followed by the EPT in some sets. Hence, it is interesting to take a closer look to them. Both are partitions of the space, the pivots of NANNI take only the closer points, and the pivots of EPT also take the ones far away. Both indexes can be seen as a set of layers, the EPT has pivot groups [Ruiz13], the MANNI has ANNI's, but the layers of EPT are similar, meanwhile the MANNI is composed by a NANNI and several ANNI's. The search in the EPT requires that each point in the database to be individually checked if it is discarded o not; the search in the MANNI can discard regions of points.

Another difference is the simpler implementation of an ANNI based index compared with the EPT. More detailed, EPT's construction lies in the estimation of random variables for both dataset and queries that allow to estimate search costs. In this regard, an ANNI directly measures the search cost of a training collection.

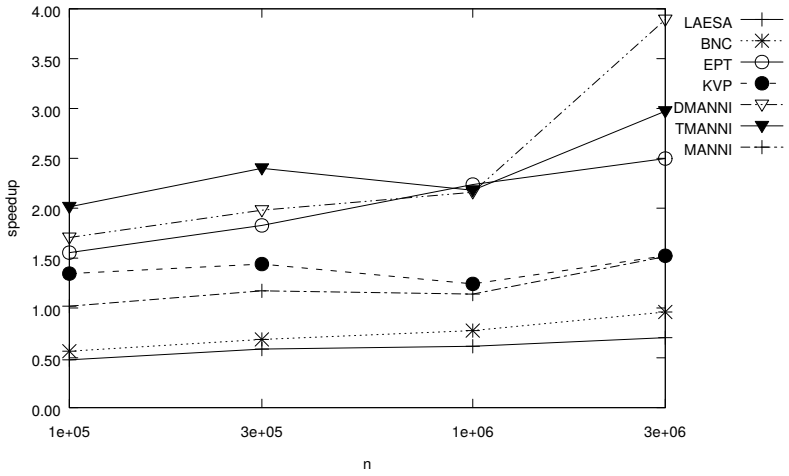


(a) ***up to*** 10 ***integers***

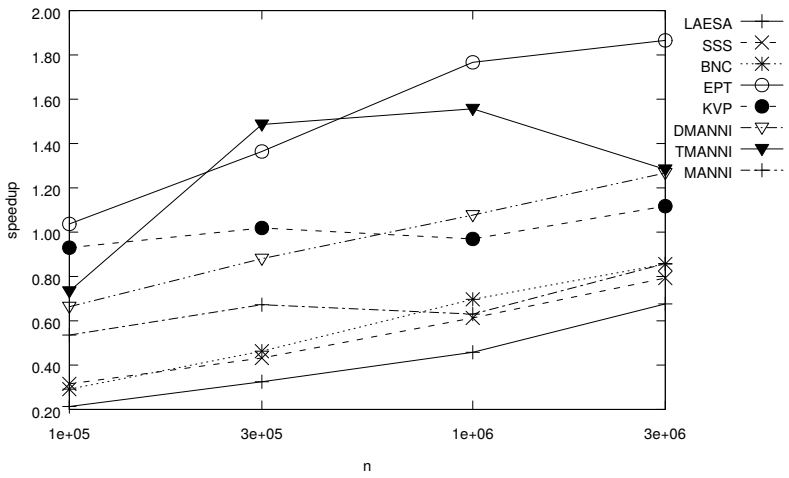


(b) 10 ***to*** 30 ***integers***

Figure 3.13: Search speedup for increasing N, the indexes are segmented into classes of memory usage. Synthetic datasets (RVEC-12). Higher is better.

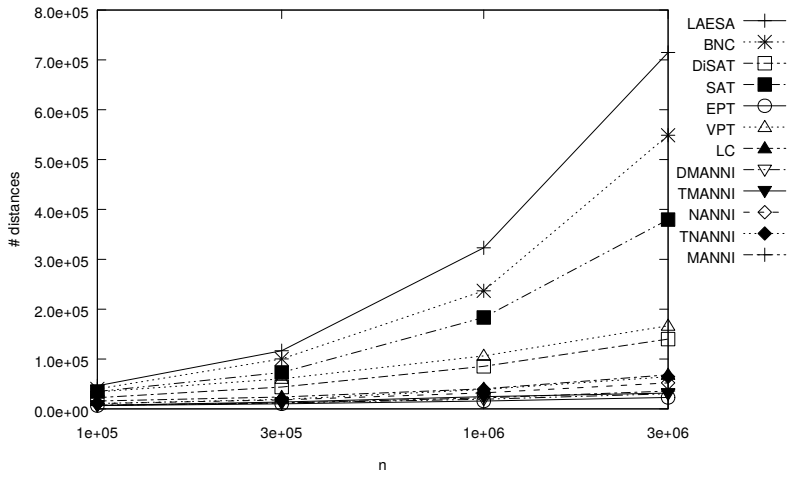


(a) 30 ***to*** 100 ***integers***

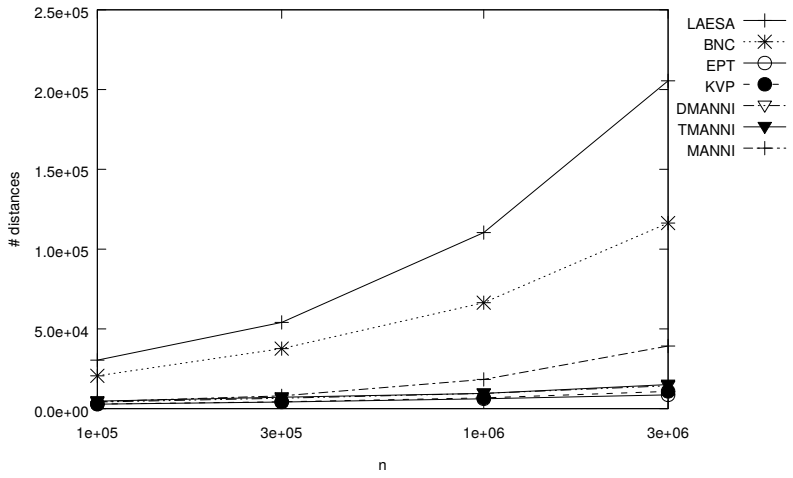


(b) 100 ***to*** 300 ***integers***

Figure 3.14: Search speedup for increasing N, the indexes are segmented into classes of memory usage. Synthetic datasets (RVEC-12). Higher is better.

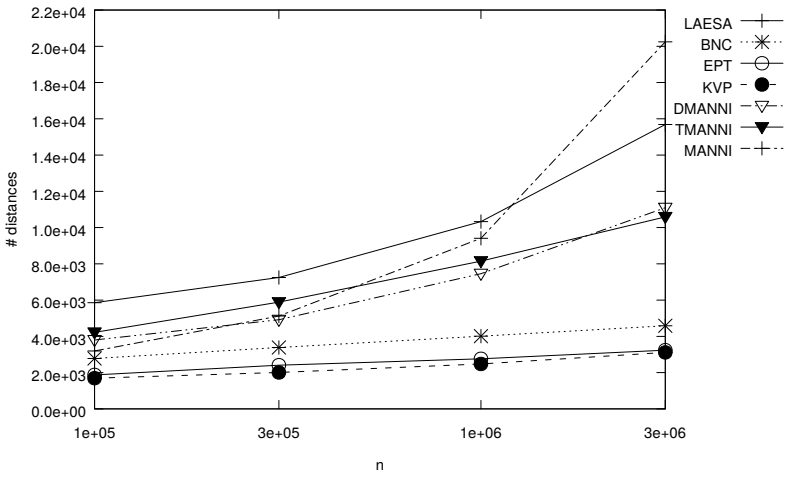


(a) ***up to*** 10 ***integers***

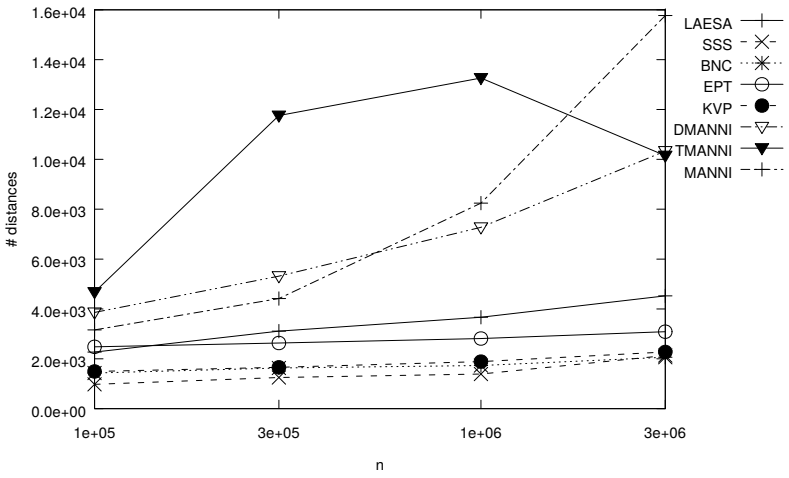


(b) 10 ***to*** 30 ***integers***

Figure 3.15: Search cost performance comparison for increasing N (RVEC-12), the indexes are segmented into four classes of memory usage. Lower is better.

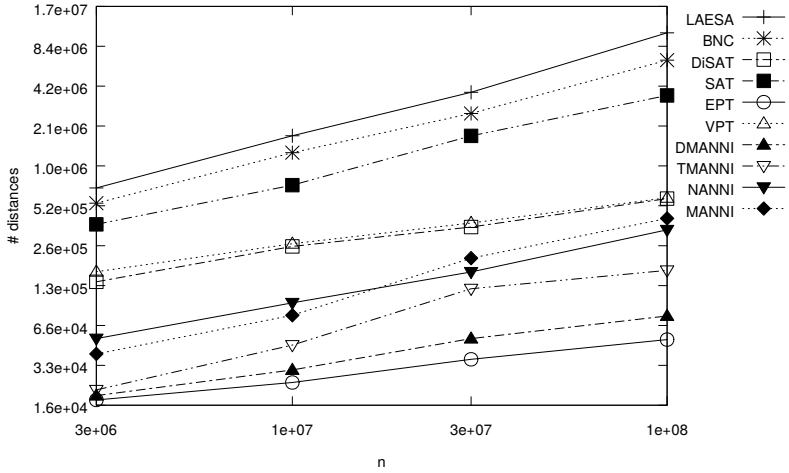


(a) 30 ***to*** 100 ***integers***

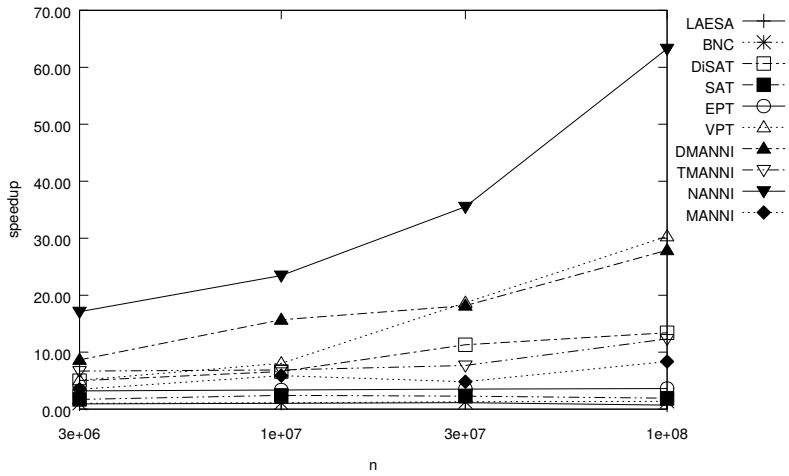


(b) 100 ***to*** 300 ***integers***

Figure 3.16: Search cost performance comparison for increasing N (RVEC-12), the indexes are segmented into four classes of memory usage. Lower is better.



(a) ***Number of evaluated distances. Lower is better***



(b) ***Speedup. Higher is better***

Figure 3.17: Search cost performance comparison for increasing N (RVEC-12), the indexes are segmented into four classes of memory usage. Datasets of $3, 10, 30$, and 100 million items are used. All indexes have use at most 4 instances (12 integers)

Construction and Speedup

From a practitioner point of view, the essential aspect of an index is its speed. The entire system will be dragged by a slow index (or no index at all) or will be lifted by a speedy index. However, the *search life* of an index (the expected number of queries over a fixed set of data) must be taken into account. Since the construction cost should be amortized over the search life of the index, in some scenarios a costly construction is not an option. For large database sizes the index could not even be constructed in due time.

To put this contribution in perspective with respect to other state of the art alternatives having comparable total speed, The construction times will be experimentally analysed. The idea of these experiments is to increase the insight about both the relation between construction cost and the search performance. The hope is to help potential users to decide what to apply in a given dataset.

A trivial parallelization (with a shared-nothing scheme) was applied, which cannot be considered a contribution. Simple, more threads ere added for each independent process, those cases are SSS, LAESA, EPT, NANNI, TNANNI, MANNI, DMANNI and TMANNI. Their construction algorithms allow to use as many threads as the number of pivots per item. On the other hand, the LC, SAT and KVP were constructed using a single thread because it is not trivial to parallelize them. Finally, the preprocessing time of LC takes into account the number of times the index needs to be constructed to obtain the best parameter combination for a given database. In other words, if the optimal ratio is determined to be $n/m = 64$ thus the ratios $n/m = 1024, 512, 256, 128, 64$, and 32 are computed. The last step (i.e., $m/n = 32$) is needed because the search stops when the parameter causes the index to slowdown.

Table 3.1 shows the speed and construction time for nearest neighbor queries in the Wiktionary database. As expected, the construction time of LC is prohibitively expensive for a large database. The second to last is NANNI, this is expected since the optimal number of centers is related to the optimal m/n parameter of the LC. It is however noticeable that it needs a single instance, and hence it is considerably faster to build than LC. On the other hand LC is the fastest; but the problem is that it needs a large search life to amortize the construction cost. Other comparable indexes are MANNI, DMANNI and TMANNI. An interesting result arises on DMANNI 8, i.e., it is constructed many times faster than LC and searches are almost as fast as LC. Notice that from those

Table 3.1: Construction time and speedup for Wiktionary in a collection of indexes. The three best results are marked.

index	speedup	construction (s)
LC 64	**12.00**	30344.85
VPT	2.17	**27.70**
SSS 0.5	2.96	840.02
BNC 8	3.20	**48.56**
LAESA 8	2.63	**4.51**
SAT	3.23	824.78
KVP 4	5.40	1587.49
EPT 8	10.63	2172.20
NANNI	9.70	12436.46
TNANNI	3.90	1014.42
MANNI 2	**11.31**	3976.15
MANNI 4	8.63	1043.22
MANNI 8	4.96	487.79
DMANNI 2	8.70	2829.77
DMANNI 4	10.81	3459.95
DMANNI 8	**11.68**	3605.05
TMANNI 2	7.16	2006.01
TMANNI 4	7.90	1721.82
TMANNI 8	8.93	1904.41

three indexes of the MANNI family, three instances were presented, all of them are fast. However, it remains an open problem to select the correct memory allowed automatically. Notice that even with three instances of MANNI (i.e., 2, 4 and 8) we have one third of the cost of LC.

Table 3.2 shows the construction and speedup performances in the RVEC-12-1M dataset. Again LC has the slower construction. In contrast with Wiktionary, LC is not the faster solving queries. This position is for NANNI, which also has a small construction time. As in the previous experiment, the DMANNI indexes are faster than MANNI and TMANNI, with a reasonable preprocessing time.

Finally, Table 3.3 shows a comparative of construction and speedup performances for very large databases (from 3 to 100 million items). In these cases,the tests of LC, SAT, BNC, and SSS were avoided. The first one, LC, cannot be properly created for the given databases because its preprocessing time is prohibitively large. On the other hand, the rest of the indexes (SAT, BNC, and SSS) have little filtering power and the search speed is comparable to the sequential scan. The EPT have the largest construction costs, this is because its construction algorithm estimates variables

Table 3.2: Construction time and speedup for RVEC-12-1M. The construction of each index is allowed to use at most many threads as references per item has the index. The best three results are marked.

index	speedup	construction (s)
LC 64	**8.67**	12485.90
VPT	2.56	**17.28**
SSS 0.5	0.61	109.34
BNC 8	0.92	**8.91**
LAESA 2	0.93	**0.34**
SAT	1.97	149.97
KVP 2	2.15	216.54
EPT 4	2.94	940.35
NANNI	**10.70**	870.88
TNANNI	8.21	528.50
MANNI 4	3.23	110.40
MANNI 2	6.67	188.29
DMANNI 4	6.86	494.50
DMANNI 2	**9.61**	641.96
TMANNI 4	4.65	274.63
TMANNI 2	6.40	301.95

Table 3.3: Construction time and speedup for RVEC-12 over 3, 10, 30, and 100 million datasets. In this experiment, all indexes use at most four pivots per item, mainly to maintain a small memory footprint as it is mandatory in large databases. In the case of KVP this means two close and two far pivots. Construction time is in seconds. The best result per column is marked.

index	3×10^6		10^7		3×10^7		10^8	
	speedup	construc.	speedup	construc.	speedup	construc.	speedup	construc.
VPT	5.07	**85.10**	8.03	**999.11**	18.67	7291.36	30.32	40400.25
KVP/2	1.87	762.83	2.42	2482.13	2.49	10064.21	2.48	43331.18
EPT/4	3.20	2926.22	3.35	18298.50	3.52	94019.37	3.61	357873.98
MANNI/4	3.48	322.45	5.87	1772.35	4.81	**3655.37**	8.36	**18511.84**
DMANNI/4	8.65	1634.77	15.69	7396.48	18.17	21917.46	27.89	96508.12
TMANNI/4	6.68	1054.33	6.87	3416.31	7.68	8514.90	12.32	51430.25
NANNI	**17.11**	4746.09	**23.45**	18013.59	**35.51**	63381.12	**63.26**	239106.84

that converge slowly, which can be problematic on large databases. On the other hand, DMANNI is faster, achieving up to 27.89 times faster than sequential search over the 100 million dataset. It is followed by TMANNI, yet with a significant speed difference.

3.3 Summary and Future Work

This chapter introduced the singleton indexes which allow self-optimization, accepting as parameter the amount of memory available for indexing, and using as prior knowledge only the query distribution. This last requirement could be partially fulfilled using a sample of the database; which supposes that query set will follow the same distribution than the dataset.

The ANNI's approach is oriented towards applications; then, even if programming the indexes could need some technical skills, the end user will require no additional knowledge to have an index consistently faster than the sequential scan. The experimental results show how the search cost can diverge as the units of measure, i.e. the number of distances computed or the number of elapsed seconds. So, we cannot optimize for one unit and expect to work for the other, in general. Fortunately, indexes designed to have low complexity in both measures, like *ANNI family, have both setups close to the optimum; therefore, it is possible to optimize for distances and work in real time. This is a good idea whenever is possible since measuring elapsed time can be affected by a lot of factors in real-world computer systems. As a rule of thumb, a suggestion is to use from two to four pivot groups in a {D, T}MANNI based index, or NANNI if the construction time can be amortized over a large set of queries. However, determining the right memory setup is an open problem.

In all the experiments, the LC have a small memory footprint and fast searches. It seems an adequate index for most tasks; however, the major drawback of LC is the construction time. Finding the correct parameters for a given metric space implies making several tries, mainly because there is no analysis for parameter selection in the LC. More detailed, each attempt consists of building the index, and each index construction takes almost quadratic time. This is too big in practice. The presented approach is consistently faster to build than the state of the art LC in a sequential environment. Unlike the LC, the MANNI structure consists of independent parts, there are no race conditions to consider in a parallel construction. This contrast with the construction

of the LC, where the center selection depends on all the previous selections; making the parallel construction non-trivial. Nonetheless, the (massive) parallelization of the search algorithms remain as an open problem.

There exist small but significant differences among the ANNI indexes. MANNI assumes independence between routing groups (groups of pivots or centers) while {D,T}MANNI does not. The former indexes see the index as a monolithic system, this essentially implies {D,T}MANNI are more robust indexes. For notation, D stands for Distance optimized indexes, and T for Time optimized indexes. The total search time is noisy in a multiuser environment, but adding groups of routing objects alleviate this. However, the MANNI indexes are simpler and with low internal complexity, they converge to similar optima when considering time and distances.

3.3.1 Comments on Disk-based MANNI Indexes

This thesis focuses on main memory indexes. However, it is worth mentioning the importance of an index stored in secondary memory,[1] mostly when the size of the dataset plus the size of the indexes surpasses the available main memory. There exist some metric indexes designed to work on disk. Ciaccia et al. [Ciaccia97] introduced the M-Tree, a dynamic, disk-based metric index. It can be seen as a compact partition index. However, the dynamic operations are the distinguishing property of the M-Tree, this algorithm takes similar decisions than B-Tree or R-Tree [Cormen01]. Skopal introduced the PM-Tree in [Skopal04]. The PM-Tree is an M-Tree enriched with a set of pivots. It significantly improves the search performance of the M-Tree, at the cost of extra complexity on the pivot operations and extra memory to store distance values to pivots. Another approach of secondary memory indexes is M-Index exhibited by Novak and Batko [Novak09]. The M-Index consists of a compact partition index stored in a B+-tree. The idea is that centers of regions are assigned to integers, distant enough among them to store all items in each region. Then, each item is encoded to an integer based on its closer center. This strategy reminds that of iDistance [Jagadish05], this later working under multidimensional data.

Notice that larger memories imply slower accesses; so, it must be done whenever this is the best (only) alternative. Think, for example, in a very large datasets. It is impossible to overcome

[1] In modern hardware there is a memory hierarchy, from the traditional magnetic disk storage, solid state-based storage, hybrid storage, distributed storage, etc. This discussion is focused on traditional magnetic disks, where random accesses are slow operations due to the internal mechanical movements, and sequential access are fast.

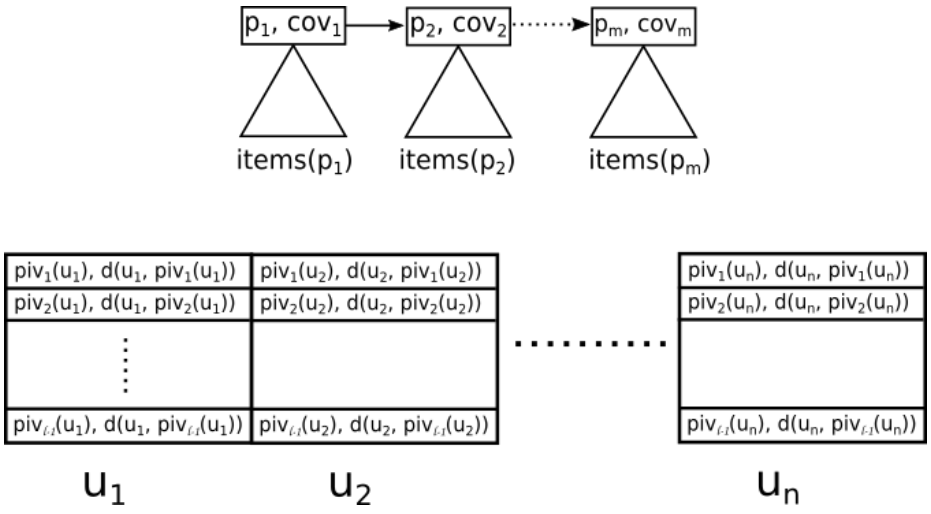


Figure 3.18: A MANNI index. The top figure corresponds to NANNI and the bottom matrix illustrates the $\ell - 1$ ANNI's. Notice that piv is parametrized by the number of ANNI denoting the independence among sets of pivots.

the memory issues arising of their indexing. Three cases are considered: i) the case where the index and the pivots can be stored in memory, ii) the case where the index can be in memory but the pivots and the dataset must be stored on the disk, and iii) the case where all must be stored in secondary memory.

In the former case, the index and all pivots can be stored in memory, but the database must be on the disk. Since the ANNI family has a small memory footprint, this could be the case for many practical cases. This approach simplifies the index since it does not require a significant modification. However, there exist important conditions that need to be kept. Please recall that MANNI is composed of a node based index and $\ell - 1$ ANNI indexes. The NANNI induces a permutation of the dataset since it stores a list of compact regions, i.e., the set of items being assigned to the same pivot/center. The dataset must be stored in the same order induced by the *leader* to reduce the number of random access to the disk. In some way, this can be performed in the same way than M-Index, or with an inverted index structure. Figure 3.19(a) illustrates how the dataset should be stored; the MANNI will follow its normal organization (Figure 3.18). The idea is that items in the same region are accessed in a single random access.

In the second case, the index is the only structure allowed to be in memory. This is similar to the previous one yet the pivots need to be stored on the disk. So, we must ensure that the distances

$$\begin{aligned}\mathsf{items}(p_1) &\to u_{11}, u_{12}, \cdots, u_{1n_1}\\ &\vdots\\ \mathsf{items}(p_m) &\to u_{m1}, u_{m2}, \cdots, u_{mn_m}\end{aligned}$$

(a) ***The index and the pivots are stored in RAM and the dataset in secondary memory. Each set of items is accessed sequentially.***

$$\begin{aligned}\mathsf{actual\ pivots} &\to p_1, p_2, \ldots, p_{\ell m}\\ \mathsf{items}(p_1) &\to u_{11}, u_{12}, \cdots, u_{1n_1}\\ &\vdots\\ \mathsf{items}(p_m) &\to u_{m1}, u_{m2}, \cdots, u_{mn_m}\end{aligned}$$

(b) ***The index is in main memory plus pivots and the dataset in secondary memory. The pivots are stored as a sequential list.***

$$\begin{aligned}&\mathsf{actual\ pivots} \to p_1, p_2, \ldots, p_{\ell m}\\ &\mathsf{NANNI's\ cov} \to \mathsf{cov}(p_1), \mathsf{cov}(p_2), \ldots, \mathsf{cov}(p_m)\\ \\ &p_1 \to \mathsf{COL}(u_{m1}), \ldots, \mathsf{COL}(u_{mn_m}); u_{11}, \ldots, u_{1n_1}\\ &\vdots\\ &p_m \to \mathsf{COL}(u_{m1}), \ldots, \mathsf{COL}(u_{mn_m}); u_{m1}, \ldots, u_{mn_m}\end{aligned}$$

(c) ***The index, the pivots and the dataset are stored in secondary memory. Here,*** n_i ***is the number of items in the region defined by*** p_i***, i.e.,*** $n_i = |\,\mathsf{items}(p_i)|$***. The*** $\ell - 1$ ***ANNI's are encoded per item as*** $\mathsf{COL}(u)$***, i.e., a column of Figure 3.18.***

Figure 3.19: Three sketches to solve the three cases of disk-based MANNIs. On the top-left, the storing order; each entry represents the items in the region induced by p_i. On the right, the order of the pivots on disk; basically, a large list of explicit objects. On the bottom, everything is into the disk. For simplicity, each of the NANNI and ANNI contains m pivots.

$d(q, p)$ for all pivots p in all parts of MANNI should be computed using a single random access. For this matter, all pivots need to be stored sequentially, no matter if they belong to the leader or to the ANNIs. Then, a consisting pivot identifier must be globally ensured. Figure 3.19(b).

On the last case, none of the structures can be stored in main memory. This is an extreme case. Firstly, as the second case, we need to store all pivots sequentially, but also we need to save all the covering radii. All of these should be done in one random access. Such that $d(q, p)$ is evaluated for all pivots, and all regions are also selected to be visited. As case 2, all regions are stored, but we also store all ANNI cells related to items in the region together to the actual items. Figure 3.19(c) illustrates this new arrangement. Despite the cumbersome details, this secondary memory index is just a MANNI taking care the number of random accesses.

Even when these approaches are promising it is necessary to notice that they suppose that the structure is already optimized, this is because the construction step is not trivial. It is important to determine if this optimizing process can be successfully applied to secondary memory indexes. Also, our comments are too general since a lot of details appear in disk-based indexes.

Chapter 4

Approximate Indexes

As mentioned on the previous chapters, there are some metric spaces that are really difficult for indexes to perform searches faster than the sequential scan. A reason for this is because those metric spaces have a high intrinsic dimension. For those kind of situations, there exist the *approximate indexes* that are allowed to miss in some of the results. Some elements of the query ball can be missed during the search, but in favor of a much faster index. Generally, this type of indexes have a parameter to adjust how precise we want the set of results.

Some indexes are designed to be approximate since the beginning and they scale better with both the intrinsic dimension and the database size [Clarkson99, Chavez08, Gionis99, Patella09, Tellez09, Amato08, Esuli09, Tellez10, Tellez11a, Tellez11b, Tellez13]. The approximated indexes most related with this contribution are described below.

In non-exact indexes, it is needed a way to measure the precision of a result. We need to decide, given two sets of results, which one is closer to the exact result. In the rest of this chapter, it is used the *recall* value. The recall for a result R' is the ratio between the number of elements of R' that appear in the exact result R, and the number $|R|$. That is, $\frac{|R \cap R'|}{|R|}$.

4.1 Techniques Used in Approximate Indexes

Next is reviewed a list of the most common techniques used by approximate indexes.

Sensitive Hashing. Perhaps the most popular NN search strategy is that based on Locality Sensitive Hashing (LSH). Gionis et al. [Gionis99] introduced LSH, a fast approximate proximity searching technique giving probabilistic guarantees on the quality of the result. The general idea of an LSH index is to find hashing functions such that close items share the same bucket with high probability, while distant items share the same bucket with low probability.

More formally, a family of hashing functions $\mathcal{H} = \{g_1, g_2, \cdots, g_h\}$, $g_i : U \rightarrow \{0, 1\}$ is called (p_1, p_2, r_1, r_2)-sensitive if, for any $p, q \in U$,

— if $d(p, q) < r_1$ then $Pr[\mathsf{hash}(p) = \mathsf{hash}(q)] > p_1$,

— if $d(p, q) > r_2$ then $Pr[\mathsf{hash}(p) = \mathsf{hash}(q)] < p_2$,

where $\mathsf{hash}(u)$ is the concatenation of the output of individual hashing functions g_i, following a fixed order, that is, $\mathsf{hash}(u) = g_1(u)g_2(u) \cdots g_h(u)$.

Notice that LSH gives guarantees in terms of distances, not in terms of recall. To achieve the desired p_1 and p_2 values, many empty buckets may have to be created (for potential queries of U that are far from any element in S). This poses a memory problem, which can be alleviated by combining several hashing function families. Still, high-quality indexes usually need high amounts of memory.

LSH can be defined for general metric spaces, but the process of finding suitable hashing functions g_i is not trivial and has to be engineered for each case. There exist simple LSH functions for the Euclidean (L_2), Hamming, Jaccard, and Cosine distances [Andoni08a, Gionis99].

RNG*. In [Arya93] Arya et. al. presented a graph over a set of points to perform nearest neighbor searches. Their theoretical approach was a division of the space for every point p in cones of the same angle centered at p. Then p will be connected to some *distributed* points in each cone. This method only works with euclidean spaces and although has some result guaranties (in the precision), their hidden constant are expensive. A more practical version consist on a modification of the *relative neighborhood graph* (RNG) which is an undirected graph where two points are connected if there is no point that is simultaneously closer to both points. They called the modification RNG* which turns out to be the HSP (see Chapter 5). They build a $k-d$ tree over the data and the starting point would be the leaf containing the query point. Then a *best-first* search is performed on the

graph. The initial attempt had long links but the authors preferred to *simulate* them using better starting points. The RNG* takes $O(n^2)$ time to build and if the $k-d$ tree will also be constructed, it would be impractical for big databases.

Combinatorial Approaches and Rank Cover Trees. In [Goyal08] near neighbors are found using a random walk over a precomputed table of neighbors. The required preprocessing time and memory for the table is quadratic, preventing the scalability of the approach in large sets. In [Beygelzimer06] the Cover Trees are introduced. CT is composed of levels, where a nodes at some level cover lower levels using a notion of proximity. The SASH data structure [Beygelzimer06] is constructed by sampling the database, connecting the non-sampled elements to their approximate nearest neighbors and repeating in the next level.

In [Houle13], the authors introduced the Rank Cover Trees (RCT) incorporing ideas from the above algorithms. A tree is built using ordered rank for pruning, instead of rules derived from distances and the triangle inequality. Node descendants in the tree are obtained using a rank order. Since only ranks are used for navigation, the number of nodes visited is known a priori. The authors include proofs, or sketches, for complexity and accuracy bounds.

k-Nearest References (KNR). In recent years, a set of small and fast indexes have appeared, among the remarkable ones are CNAPP [Tellez13], PP-Index [Esuli12], MIF [Amato14], and the quantized permutations [Mohamed14, Mohamed13]. In these indexes, a similar structure arises: every node is associated with a set of k nearest references (KNR), where the set of references is a sample of the database. The similarity between items is hinted by a similarity function over the shared neighbors. In this approach there is no navigation and proximity queries are solved using an inverted index. These structural similarity was systematically explored in [Chávez15], adding several new indexes to the list. In the same paper, the authors include experimental comparisons showing the performance of KNR indexes as compared against near neighbor techniques based in hashing, in particular, LSH [Indyk04a, Andoni08b] and DSH [Gao14].

Among the best KNR indexes introduced in [Chávez15] are KNR-Cos and KNR-LCS. These indexes use the cosine similarity and the longest common sub-sequence as similarity functions as proximity predictors, respectively. The experimental evidence suggests that KNR-Cos sur-

passes the majority of the state of the art KNR indexes, so it will be used as the canonical member of the KNR family in the experimental section.

Approximate Proximity Graph. Recently Malkov et al. [Malkov12] introduced the *Approximate Proximity Graph* (APG), an index with excellent searching times. The authors suggest that the speed and accuracy boost is a consequence of the construction step, since it is claimed that it produces a graph with *small world* properties. APG can trade speed and accuracy for memory usage. If memory is scarce, speed and accuracy will also suffer.

Construction and searching are entangled in APG. The construction is incremental and consists of a simple rule: To insert the j-th element, simply find the (approximate) t-nearest neighbors among the $j-1$ elements already indexed; then, the new item is linked (in both directions) to its t near neighbors. The authors encourage the preservation of links to distant items (large links), defining them as essential to preserve their search performance. Notice that these links naturally arise using the incremental construction.

The search consists on performing m greedy search steps. At each step, the algorithm selects a random starting point and greedily follows that neighbor minimizing the distance to the query. This procedure is recursively applied until the distance cannot be decreased. The accuracy of one search step is poor, but it gets boosted by selecting m starting points. The authors indicate that $m = O(\log n)$ attempts will be enough to achieve recall close to one.

The chief parameters governing accuracy and speed are the number of starting points m and t, the number of direct neighbors in the graph. To achieve high accuracy, it is necessary to use many starting points, which hurts the speed; simultaneously, increasing t could boost speed but the index will need more memory.

Recently, in [Malkov14], the searching procedure is updated as follows. As in the previous search, the search is repeated m times; however, the search steps share knowledge and each starting point is now the best-known non-visited item, similarly to a best-first search [Edmund K. Burke14]. More detailed, a set of visited items and a set of candidates are kept along the entire search process. At the beginning of each restart, the algorithm appends a random item to the list of candidates in order to add diversity to the search process. Algorithm 9 formally describes this search.

Algorithm 9 The search algorithm for APG as described in [Malkov14].

Name: Search algorithm APG
Input: The database S, a graph G over S, the query q, and the number of restarts m
Output: The set res of near neighbors of q.

```
1: Let res be an empty min-queue of fixed size k
2: Let candidates be an empty min-queue
3: Let visited be an empty set of object identifiers
4: for i = 1 to m do
5:    Randomly select c ∈ S \ visited
6:    Append c into visited and (d(q, c), c) into candidates and res
7:    loop
8:       Let (r_b, best) be the nearest pair in candidates
9:       Remove best from candidates
10:      Let r be the biggest radius in res, empty res defines r = ∞
11:      if r_b > r then
12:         break loop
13:      end if
14:      for all u in the neighbors of best do
15:         if u ∉ visited then
16:            Add u to visited and (d(q, u), u) to candidates and res
17:         end if
18:      end for
19:   end loop
20: end for
```

4.2 Proximity Search as an Optimization Problem

The nearest neighbor search can be seen as an optimization problem where the task is to find either the minimum or maximum value of a function f that can be a distance or similarity, respectively. Based on APG and this new framework, it is proposed a number of new indexes listed below:

- APG* is an APG without needing to know the parameter m (the number of restarts).
- APG*-R improves APG* limiting the shared state among search steps.
- Finally, the BS, a beam search based algorithm is introduced.

The problem of combinatorial optimization consists on finding the optimal state among a finite set Ω of states. Each state is associated to a value by an objective function $f\colon \Omega \to \mathbb{R}$. An optimal state is one that reaches either the global minimum or maximum. In other words, the goal is to obtain ω such that

$$f(\omega) = \min_{x \in \Omega} f(x) \text{ [or, } \max_{x \in \Omega} f(x) \text{ in the case of maximization]}.$$

Minimization should be used in the case of working with distances, while a similarity should be used along with a maximization process. The focus is going to be in distance functions.

As stated, the approach is to formulate the proximity search as an optimization problem. In this transformation the generality of the metric (similarity) approach will be kept; then, the following characteristics are preserved:

- The distance function is the only way to compare any two items, also there is no way to get the derivative between items in the metric space.
- Since objects are seen as a black box, items cannot be generated.
- There is no special assumption over the database S or the distance function d. So, for example, we cannot ensure a unique minimum, or a particular landscape of the optimized function.
- The size of the database can be large, also the intrinsic dimensionality can be high.

Taking these points into account, the optimization techniques based on local search were selected to be used. Some reviews for optimizations techniques are presented next. The interested reader is referenced to [Edmund K. Burke14] for more information. In the following paragraphs the basic notation and the problem statement are introduced.

Definition 4.2.1 (State transitions $\mathcal{N}$) *$\mathcal{N} : \Omega \to \Omega^+$ is a function associating each state u with a set of possible transitions, it resembles the notion of neighborhood of u.*

Definition 4.2.2 (The function to be optimized d_q) *Each query q defines a function to minimize, $d_q(u) = d(q, u)$ for $u \in \Omega$.*

Definition 4.2.3 (The minimization algorithm $\mathcal{A}$) *Let $\mathcal{A}$ be an algorithm that tries to minimize d_q. It explores Ω with the help of $\mathcal{N}$ to reach an acceptable result.*

Nearest neighbor search is then stated as the problem to find u such that $d_q(u)$ is minimum and $u \in S$, using $\mathcal{A}$. The assumption $S = \Omega$ is made, other cases could be of interest but the algorithms were designed to work under this restriction. Even when the case $S \subset \Omega$ is of use and significant in some fields, it is beyond the scope of this work.

There exist two main components in the proposed approach, the minimization algorithm $\mathcal{A}$ and transition function $\mathcal{N}$.

— Due to the difficulty of the general optimization problem (and the proximity search problem under high dimensionality), $\mathcal{A}$ does not ensure to find the global minimum, yet it should provide a good local minimum.

— On the other side, $\mathcal{N}$ should describe with precision the relation from u to its neighborhood, such relation will be exploited by $\mathcal{A}$ to minimize d_q.

— Notice that $\mathcal{A}$ and $\mathcal{N}$ are tightly coupled.

4.2.1 About $\mathcal{A}$

There exist many heuristics and meta-heuristics to solve large and hard combinatorial optimization problems. As commented, the local search strategy was used due to the properties of the problem statement and restrictions. In the next paragraphs, the most prominent heuristics and meta-heuristics suitable for this case are reviewed.

Greedy Search.

Perhaps the simpler working algorithm for local search is the *Greedy search* algorithm, it roughly works as follows to solve a nearest neighbor query ($\mathsf{nn}(q)$):

i) Take a random state from Ω as the first answer.

ii) Let s be the current best answer to $\mathsf{nn}(q)$. Try to minimize d_q looking at $\mathcal{N}(s)$.

iii) If there exists some u that decreases d_q then u becomes the best answer and repeat step ii); the search is stopped otherwise.

iv) The best guess of $\mathsf{nn}(q)$ is such item producing the minimum known value of d_q.

Greedy search will retrieve a local minimum. If the retrieved local minimum is not good enough, then it can be improved repeating Greedy search with random starting points as many times as we need. The idea is to try another initial point and route, in the hope to surpass the previous estimation of d_q (and the involved u).

From the proximity search perspective, if each search has a probability P of finding the desired $\mathsf{nn}(q)$, then we can achieve a desired probability $P^\star$ with m independent searches where

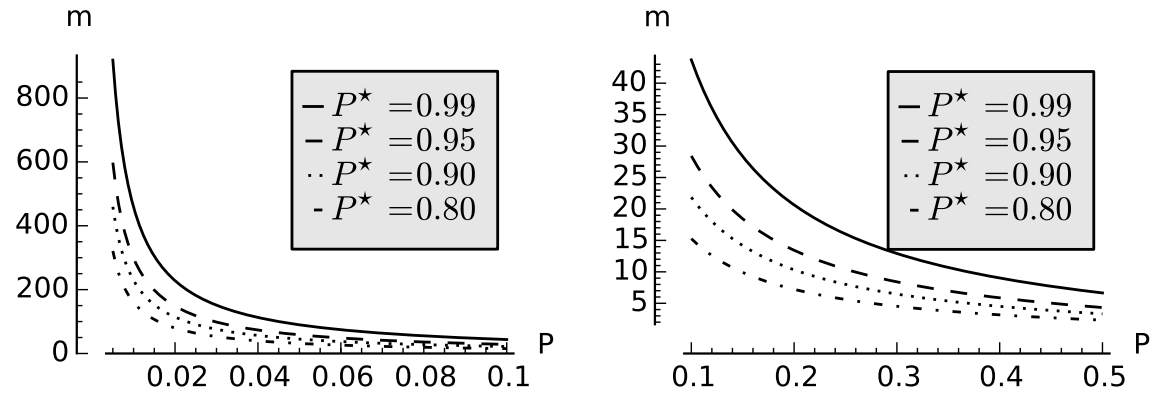
$$m = \frac{\log(1 - P^\star)}{\log(1 - P)}; \tag{4.1}$$

this expression supposes independence of $\mathcal{N}(\cdot)$, which can be hard to achieve. Notice that this formulae does not predict the closeness of the retrieved item at any given m; however, it is well known that it is an easier task than retrieving those items satisfying the query.

From the optimization perspective, a low P implies a large number of local minimums. This is the counterpart of the difficulty expected due to the curse of dimensionality in the proximity search problem. Figure 4.1 illustrates this behavior. Notice that this method is practical when

$$cost = m \times hops \times |\mathcal{N}(\cdot)| + 1 < n, \tag{4.2}$$

where $hops$ is the expected length of the path (number of transitions) from a random starting point to the local minimum found, and $|\mathcal{N}(\cdot)|$ the expected number of transitions per item. Hopefully, this cost will be fairly practical, as shown in the experimental section.



(a) P ***ranges from 0.005 to 0.1*** **(b)** P ***ranges from 0.1 to 0.5***

Figure 4.1: The number of steps m for several desired recall ($P^\star$) as a function of the expected recall per step P. Observe how low P values need extremely large m values. This is consequent with the curse of dimensionality.

Tabu Search.

Tabu search is a meta-heuristic allowing to take non greedy decisions at each step, i.e., not always decreasing d_q at each transition. This strategy recognizes the existence of local minima and plateaus, then the idea is to avoid loops with *a non always improving strategy*. Tabu search solves the problem adding memory to mark already visited states as tabu, i.e., prohibited, and then ignoring them on posterior stages of the search process. Due to is living time, the memory can be seen as short-term, medium-term, or long-term. The usage of each flavor is dictated by the desired behaviour, the problem itself, and the available resources.

Beam Search.

Beam search is a strategy that optimizes d_q using a beam of states $B \subset \Omega$ of size b. At each step, beam search computes $B' = \cup_{s \in B} \mathcal{N}(s)$ and selects the best b from B and B'. The search stops whenever it is impossible to improve B. Tabu search can be used along beam search to avoid non-essential computations. At the beginning of the search process, B is populated randomly from Ω. Notice that the low memory requirement is the main advantage specially for large datasets or systems with scarse memory.

4.2.2 The difficulty of finding an optimal $\mathcal{N}$

A fundamental component of the local search meta-heuristic is the transition function $\mathcal{N}$, which properly defined, can ensure the convergence to the global minimum, or less strict, to a good local minima. In order to sketch the problem's complexity of defining a good $\mathcal{N}$, let $\mathcal{A}$ be the Greedy Search algorithm. The function $\mathcal{N}$ defines a graph where the vertices are the points of the database and a vertex u is connected with the elements of $\mathcal{N}(u)$. Note that the greedy search starting on a point u will always stay in the path connected component of u, therefore, it is very important that the whole graph would be path connected.

Definition 4.2.4 (Query set Q) *Let $Q \subset U$ be the collection (possibly infinite) of all possible queries.*

Definition 4.2.5 (Optimal $\mathcal{N}$) *Define an* optimal $\mathcal{N}$ *as the one having the minimum cardinality that ensures that $\mathcal{A}$ finds the global minimum.*

Lemma 1 (There is no optimal $\mathcal{N}$ in a general and practical case) *It is impossible to create an optimal $\mathcal{N}$ in the general case satisfying inequality 4.2. Ensuring that property implies that $\mathcal{N}(u)$ contains at least one item that minimizes d_q at each stage until we found the global minimum (for all $u \in \Omega$).*

Proof 4.2.1 *For every Ω and every distance function d_Ω defined in it, there exists a metric space (M, d) such that d is an extension of d_Ω in M and the optimal $\mathcal{N}$ is the one defined by the complete graph on it as described in Ω [Navarro02b]. Then it is not practical.*

Since the approximate approach is supposed, the transition function does not need to ensure that $\mathcal{A}$ finds the global optimum d_q. From the similarity search perspective, it is allowed to lose some results, and also, some retrieved items will not be part of the exact query. Nonetheless, the error rate is low and the performance gain is significant, as supported by the experimental section.

How to build $\mathcal{N}$.

The previous section shows the complexity to find the optimal $\mathcal{N}$ functions. Since the focus is neither on optimality nor exact results, this is just a complication. Fortunately, Malkov

et al. [Malkov12, Malkov14] introduced a working $\mathcal{N}$ construction for its APG which appears in Section 4.1. The routing is done using improved versions of greedy search and best-first search. A deep study is given by APG's authors. As APG, a non-directed version of the graph was used.

4.3 Direct Improvements on APG

Algorithm 10 The search algorithm for APG*. The main difference is that m is not longer necessary and a small value σ is needed to guess a good number of restarts. The main idea is that search is restarted while the covering radius of res improves. In other words, a good local minimum is found.

Name: Search algorithm APG*
Input: A transition function $\mathcal{N}$, the database $\Omega = S$, the query $k - \mathsf{nn}(q)$, a minimum number of repeats σ (pretty small, e.g., 4)
Output: The set res of near neighbor of q.

```
1: Let res be an empty min-queue of fixed size k
2: Let candidates be an empty min-queue
3: Let visited be an empty set of object identifiers
4: repeat
5:    Let cov* ← cov(res)
6:    for i = 1 to σ do
7:       {This for loop is identical to that of APG (Algorithm 9)}
8:       Randomly select c ∈ S \ visited
9:       Append c into visited and (d(q, c), c) into candidates and res
10:      loop
11:         Let (r_b, best) be the nearest pair in candidates
12:         Remove best from candidates
13:         if r_b > cov(res) then
14:            break loop
15:         end if
16:         for u ∈ N(best) do
17:            if u ∉ visited then
18:               Add u to visited and (d(q, u), u) to candidates and res
19:            end if
20:         end for
21:      end loop
22:   end for
23: until cov* = cov(res) {Stops when there is no improvement over d_q, i.e., cov(res)}
```

The new indexes are based on modifications over the well known APG. As previously detailed on Section 4.1, APG's search is repeated m times to improve the expected recall. As

authors suggest in [Malkov14], m must be adjusted for each dataset. This assumes an additional step, which can be difficult because the best m ideally should be fitted as n grows. A significant overestimation of m can be a problem since construction is incremental, based on searches, then it would impact both search and construction performances.

Before introducing the proposed algorithms, an important function will be defined. Let $result$ be a min priority queue (min-queue) of fixed size k. The $\texttt{COV}(result)$ function is defined as follows: if $result$ is under its full capacity then it evaluates to the maximum possible distance value; if it is full, then $\texttt{COV}(result)$ is the radius of the furthest item.

The first index contribution is Algorithm 10. The objective is to remove the m parameter, checking the changes over the global result set, i.e., the covering radius $\texttt{COV}(res)$. In some sense, d_q is been minimized, and the algorithm stops when a guess of d_q cannot be improved after σ tries. All the proposed new algorithms use the same technique. Notice that σ should be small (in the range of 2 to 4). The idea is to relax the stop condition, but not to introduce an extra parameter. Larger values of σ should be avoided in favor of other parameters.

The second index contribution is named APG*-R (Algorithm 11). The idea is that APG and APG* can be rapidly get stuck on a local minimum since they are limited to use always the best known candidate. APG*-R has similar structure than APG (and APG*); however, it is different in concept. On the one hand, APG*-R uses random starting points, not the best known in APG, and also, $res*$ is local to each step. A local memory in $evaluated$ was added to avoid unnecessary evaluations. This contrasts with where the $candidates$ set has a global scope, and the improvement policy is also global.

The final index is BS (Algorithm 12), a novel index using beam search over S and $\mathcal{N}$. As in the previous approaches, at least σ steps are made to take the decision to stop the search. So, if after some minimization steps, d_q does not change, then it has been achieved a (good) local minimum. The main parameter is the size of the beam b, which can be adjusted at any time. Notice that beam search does not need to restart; however, it needs to know b. As will be seen in the experimental section, the use of beam search as an alternative to other schemes has implied significant improvements in both recall and speed performances.

Algorithm 11 The search algorithm for APG*-R. As APG*, it tries to adjust the number of restarts automatically.

Name: Search algorithm APG*-R
Input: A transition function $\mathcal{N}$, the database $\Omega = S$, the query $k - \mathsf{nn}(q)$, a minimum number of repeats σ (pretty small, e.g., 4),
Output: The set res of near neighbor of q.

Let res be an empty min-queue of fixed size k
Let $visited \leftarrow \emptyset$
repeat
  Let $cov* \leftarrow \mathsf{cov}(res)$
  for $i = 1$ to σ **do**
    Let $res*$ be an empty min-queue of fixed size k
    Randomly select $sp \in S \setminus visited$
    Append sp into $visited$ and $(d(q, sp), sp)$ into $res*$
    Let $evaluated \leftarrow \emptyset$
    repeat
      for $v \in \mathcal{N}(sp)$ **do**
        if $v \notin evaluated$ **then**
          Add $(d(q, v), v)$ into $res*$
        end if
      end for
      Update sp to be the closer item in $res*$ not in $visited$. If there is no item satisfying the condition, then sp is set as *undefined*.
      Add sp into $visited$
    until sp is *undefined*
    $res \leftarrow res \cup res*$ {The properties of the min-queue of fixed size should be preserved}
  end for
until $cov* = \mathsf{cov}(res)$ {Stops when d_q does not improve, i.e., $\mathsf{cov}(res)$ stays unchanged}

Algorithm 12 Beam Search over Ω and $\mathcal{N}$.

Name: Beam Search
Input: A transition function $\mathcal{N}$, the database S, the query q, the size of the beam b
Output: The set res of near neighbor of q.

1: Let res be an empty min-queue of fixed size k
2: Let $beam$ be an empty min-queue of fixed size b
3: Set $visited \leftarrow \{\}$
4: Set $beam \leftarrow \{\}$
5: **for** $i = 1$ to b **do**
6: Randomly select u from S
7: Add u to $visited$ and $(d(u, q), u)$ to res and $beam$
8: **end for**
9: **repeat**
10: Let $cov* \leftarrow \texttt{COV}(res)$
11: **for** $i = 1$ to σ **do**
12: Let $beam* \leftarrow \{\}$ {Fixed sized priority min-queue of size b as $beam$}
13: **for** $c \in beam$ **do**
14: **for** $u \in \mathcal{N}(c)$ **do**
15: **if** $u \notin visited$ **then**
16: Add u to $visited$ and $(d(q, u), u)$ to res and $beam*$
17: **end if**
18: **end for**
19: **end for**
20: $beam \leftarrow beam*$
21: **end for**
22: **until** $cov* = \texttt{COV}(res)$ {Stops when there is no improvement over d_q, i.e., $\texttt{COV}(res)$}

4.4 Experimental Results

In this section a comparison of the experimental performance of the proposed technique against APG, KNR and the RCT is presented. The attention is going to be focused in the recall and the search speed. This time, the experiments were performed in a newer machine with a 24-core Intel Xeon 2.60 GHz workstation with 256GB of RAM, running CentOS 7. The multiprocessing capabilities in the search process were not used. Both, the index and the database were maintained in memory.

The following synthetic databases were selected: Nasa, Colors, Wiktionary, Documents, and BigANN-1M. The Documents database is a collection of $25{,}157$ short news articles from TREC-3 collection of the Wall Street Journal 1987-1989. It is composed by tf-idf vectors, taken from the SISAP[1] project [Figueroa09]. The angle between the vectors was used as the distance measure [Baeza-Yates99]. 100 random documents were removed from the collection and use them as queries (thus these 100 documents are not indexed in the database). The objects are vectors of hundred thousands coordinates. This dataset has a very high intrinsic dimension in the sense of Chavez et al. [Chavez01]. Even finding the nearest neighbor of a query requires reviewing the entire database in most exact metric indexes. As a reference, a sequential scan needs 0.185 seconds. The BigANN-1M database is a one million subset of the TEXMEX corpus.[2] The original dataset contains a billion SIFT points (128 coordinates). Also 256 queries were extracted from the standard query set. In average, a sequential search of the nn under L_2 needs 0.441 seconds to be solved.

The real databases were RVEC of $16, 32, 64, 128$ and 256 dimensions, each one with 10^5 randomly generated items.

In order to compare against LSH, the E^{2}LSH tool[3] by Alex Andoni was used. It automatically optimizes an LSH index for the available memory. The index memory was fixed to 1, 3, and 5GB, which is many times higher than the necessary by other indexes. In order to be fair, since LSH offers distance-based rather than recall-based guarantees, the search by radius was used (i.e., return any element within distance r to the query), setting r as the average radius of the 30-th nearest neighbor. To avoid extreme or uninteresting cases, the queries yielding empty answers were

[1] `www.sisap.org`
[2] Available at `http://http://corpus-texmex.irisa.fr/`
[3] `http://www.mit.edu/~andoni/LSH/`

removed and those with more than 1000 answers. For the RCT [Houle13] the C++ implementation, kindly shared by authors, was used. Notice that LSH, RCT and FLANN do not offer an implementation for Levenshtein and the cosine similarity, so they only are compared on datasets with the L_2 metric. Therefore, the LSH and RCT were included only on experiments over vector datasets fulfilling this requirement. Also, LSH is not present in RVEC benchmarks since its performance is not preserved under this kind of datasets [Indyk04a, Andoni08b].

While the implementations are written in C# and run under the Mono virtual machine, E^{2}LSH, RCT, and FLANN are written in C++, and run in native form. For example, their sequential search has an advantage of around 2 to 3X in the case of E^{2}LSH and RCT, and 5X times in the case of FLANN due to their code's optimizations. In order to give a full picture of the performance, both the absolute times and speedup were reported. The speedup is computed with respect to the sequential scan in its own implementation.

4.4.1 Performance per Database

In the first batch of experiments, the indexes APG,* APG*-R, BS are presented, with different values for t (the number of neighbors in the construction) for the Colors, Nasa, Wikionary, Documents, and BigANN-1M databases.

The APG, and BS are labeled with $t = 8, 16$, and 32, for each value of t, the parameter m varies from $8, 16$, and 32. The APG* and APG*-R takes $t = 8, 16$, and 32. Each KNR uses $k = 7$ and 12 near references among 2048 references (larger values of k will produce better recalls, but at cost of larger memory consumption); for each k the index is allowd to review at most 3% and 10% of the database (these will be the two points in each curve). For the RCT a covering factor of $32, 64$, and 128 (larger values will produce better recalls but slower searches) was used.

Figure 4.4.1 shows the recall and speedup of the methods for the Colors database. The APG's for $t = 8$ are the fastest but with less recall, as increasing the number of neighbors in the construction, the recall is getting better but the speed is affected. Compare this with the line of the APG*. The APG* is very close to the corresponding APG's. Now, the APG*-R achieves a higher recall but at the cost of losing speed. Next, we have the BS, it has the best overall performance, that is, for a given recall, it has generally the best speed. Note how it get the most recall. The RCT, KNR and LSH are left behind.

The results of the Nasa database are shown on Figure 4.4.1. This is the smallest database and all the indexes reach a very high recall. The results for the APG are similar to the Colors database: the recall rises with the amount of memory but there are faster alternatives. The APG* is still very close the the corresponding APG. The APG*-R gets a very good performance, being small the database, the random starts become very effective. All the BS have a very high recall at a good speed. Again, the KNR fails to match the others. The RCT has more appeal here than on Colors being the index that reaches the faster speed. The LSH is nowhere near the others.

The Wiktionary database is more complicated. It is big and its distance is more expensive. It is on this type of databases where the indexes are more convenient. On Figure 4.4.1 almost every index is hundreds of times faster than a sequential scan. The performance of the APG is again below the others and the APG* is again essentially the same. On the other hand, the APG*-R is above all others. For the correct amount of neighbors, the APG*-R is an absolute improvement of the APG*. The BS has a very good performance in general and it reaches the most recall.

The Figure 4.4.1 shows the Documents database. Note how most of the indexes have the same curve of tendency. Remember that this database has very high intrinsic dimension, so, we can have high recall or high speed, not both. The APG* is again practically the same as the APG, and the APG*-R reaches better recall values. The BS has very good recalls at a competitive speed. The KNR is left behind again.

The results for the BigANN-1M are shown on Figure 4.4.1. Look how similar these are to the Documents figure. The principal difference is the higher values for the speedup because this database is much bigger. Apart from that, the APG are the same as the APG*; the APG*-R have higher recalls values; the BS instances have higher recalls at a cost of the speedup; the KNR, RCT, and the LSH have below the average speed.

From those results we can see that the APG*-R is better in general than the APG* (which is very similar to APG). Also, all the BS have a very good recall and speed.

The best setups on vector datasets with L_2 are summarized in Table 4.1.

4.4.2 The effect of the dimension on the Search Performance

On the next figures, we are going to compare how the indexes handle the dimensionality of datasets. Here, the RVEC datasets were selected with a wide range of dimensions, specifically,

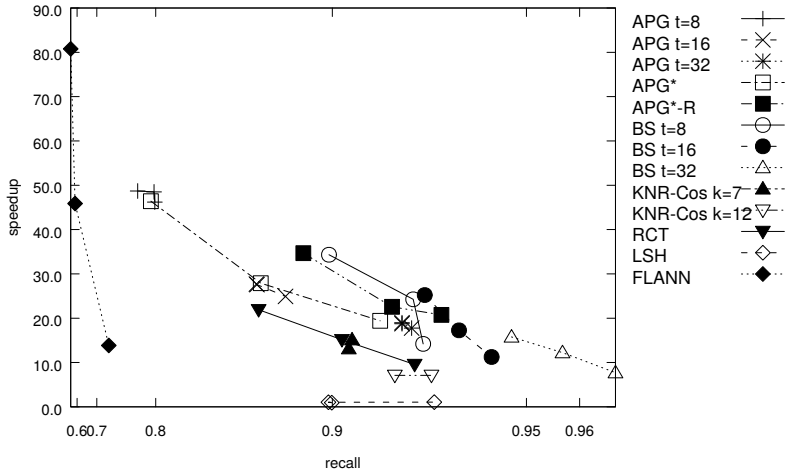


Figure 4.2: Recall and speedup performance for Colors.

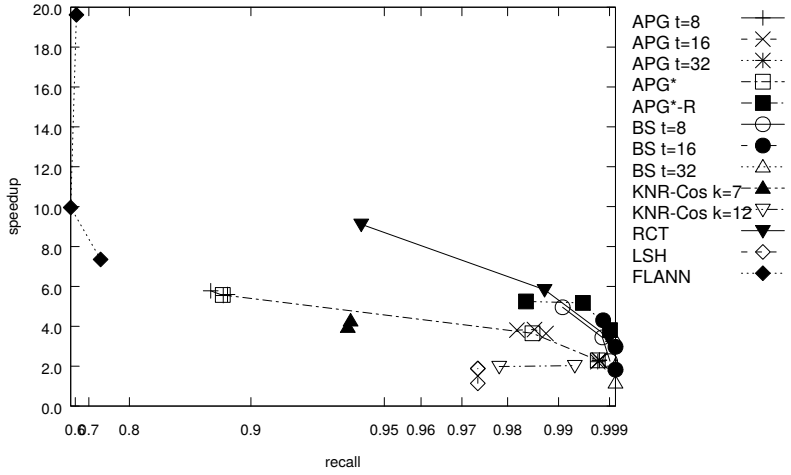


Figure 4.3: Recall and speedup performance for Nasa.

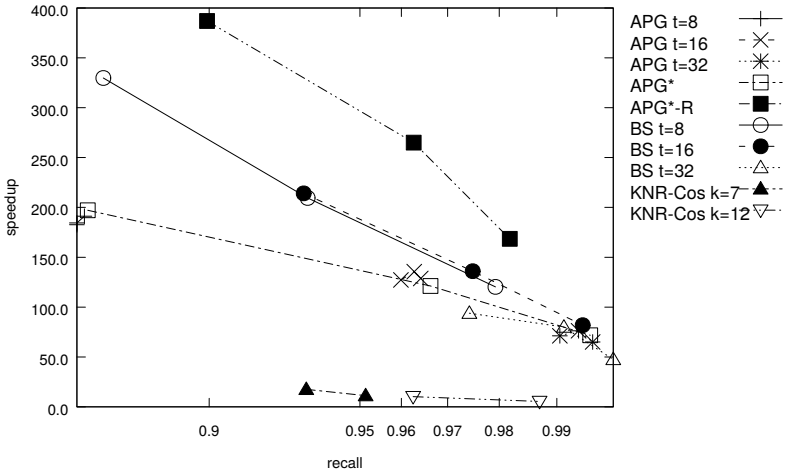


Figure 4.4: Recall and speedup performance for Wiktionary.

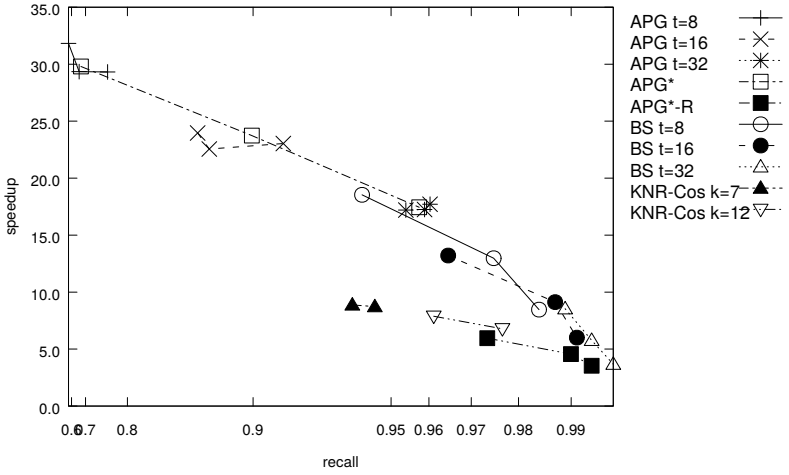


Figure 4.5: Recall and speedup performance for the Documents.

Table 4.1: Summary of the best performing indexes and setups over vector based datasets with L_2. The performance of E^{2}LSH was added to improve the comparison coverage.

COLORS					
Index	Search time (s)	Speedup	Search cost (distances / n)	Recall	Memory
RCT c=64	0.0046	15.1356	0.0534	0.9033	2.41MB
RCT c=128	0.0071	9.5628	0.0847	0.9246	2.41MB
BS t=16,b=8	0.0021	25.2166	0.0160	0.9272	14.61MB
BS t=16,b=16	0.0031	17.2586	0.0242	0.9355	14.61MB
BS t=16,b=32	0.0047	11.2444	0.0366	0.9428	14.61MB
LSH	0.0139	0.9225	-	0.8998	1GB
LSH	0.0123	1.0288	-	0.8986	3GB
LSH	0.0122	1.0697	-	0.9296	5GB
NASA					
Index	Search time (s)	Speedup	Search cost (distances / n)	Recall	Memory
RCT c=32	0.00065	9.1144	0.0632	0.9431	873KB
RCT c=64	0.00101	5.8348	0.0534	0.9874	873KB
RCT c=128	0.00611	3.4134	0.0847	0.9990	873KB
LSH	0.00077	1.1393	-	0.9736	1GB
LSH	0.00044	1.8779	-	0.9736	3GB
LSH	0.00044	1.8872	-	0.9736	5GB
BigANN-1M					
Index	Search time (s)	Speedup	Search cost (distances / n)	Recall	Memory
RCT c=32	0.0096	78.2874	0.008	0.7736	24.8MB
RCT c=64	0.0153	48.5415	0.015	0.8747	24.8MB
RCT c=128	0.0260	29.1784	0.027	0.9418	24.8MB
LSH	0.1713	0.8340	-	0.9638	1GB
LSH	0.0746	1.7985	-	0.9654	3GB
LSH	0.0492	2.6401	-	0.9613	5GB

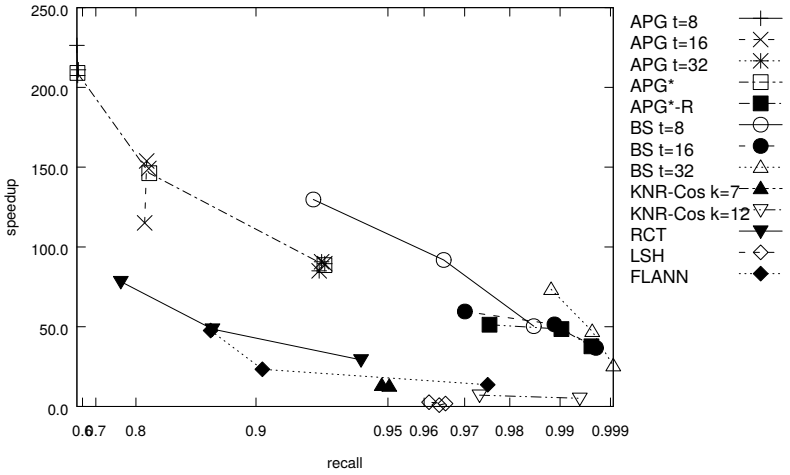


Figure 4.6: Recall and speedup performance for BigANN-1M.

$16, 32, 64, 128,$ and 256. Performance is measured in terms of recall, cost, and speedup. Each line of points represents the results for each of the dimensions.

The figures 4.7 and 4.8 show the cost for the indexes. The recall was compared with the cost of the searches. Figure 4.7(a) shows the indexes that take from 1 to 10 integers per element of the database, that is, on the APG the number of neighbors is small. All the APG and the APG* are very similar and have a very poor recall. This contrasts with the APG*-R that shows a huge improvement on it. This shows how effective is this small variant. The BS has a wide range of action, note how the recall is much better than the APG at the cost of computing more distances. Here, the KNR has a good performance with one of the configurations reaching the most recall and showing an almost constant cost to the change of the dimension. The RCT, like the BS is spread over the figure but with more distances computed.

Now, Figure 4.9(a) shows the speedup for those indexes. The APG and APG* are very fast because of the small number of distances they have to compute but the small recall makes them useless. The APG*-R appears on the middle of the other indexes. The different instances of the BS shows how it gains recall but losses speed when we increase the beam. The KNR gets good recalls but is slower than the average of the others. Finally, the RCT is fast and with good recalls for these smaller indexes. Note how the recall for the APG's and BS increases when adding more memory.

In the Figure 4.7(b) the methods take more neighbors for the elements, this makes them

compute more distances but they reach higher recall values. The APG and APG* can not get a good recall but the improve of the APG*-R lets them have one comparable to the BS. The Figure 4.9(b) contains the speeds of those indexes. Because of the small recall, the APG and APG* are very fast. The APG*-R and BS with $b = 8$ are very similar, one a little faster and the other with better recall. As expected, the other configurations of BS have higher recalls.

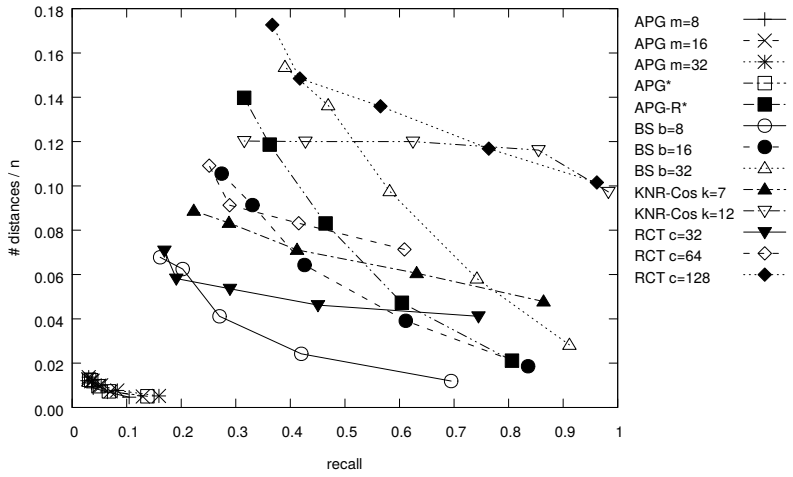
The Figures 4.8(a) and 4.8(b) are very similar, they show how the APG and APG* become to get a good recall for the first dimensions but continue to have a low value for the higher ones. With the APG*-R and BS we see that they can have very high recalls even for the big dimensions.

The Figures 4.10(a) and 4.10(b) are again very similar. The APG and APG* are not very fast. The APG*-R is faster for the dimension $16, 32,$ and 64 but the BS with $b = 8$ becomes faster for the big dimensions.

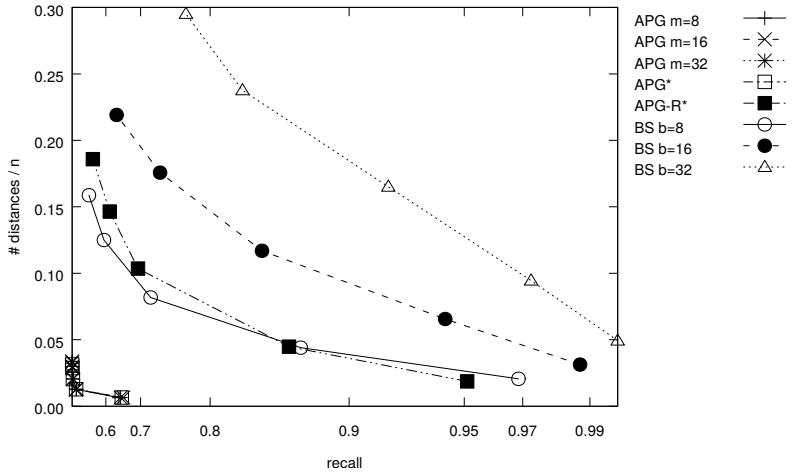
4.4.3 Scalability

A central property of any search index is its capacity to handle the size of the database, with special attention on large values. If an index cannot grow by design, then the index should be created every time the database grows; this high indexing cost is reduced using clever arrangements of $\log n$ smaller indexes (Bentley and Saxe [Bentley80], and its adaptation to metric indexes [Naidan12]). This transformation comes with a small speed overhead at search time, in addition to a constant factor introduced in the construction. This technique is beyond the scope of this document, and static indexes are a single large index without additional search overheads. Notice that the APG's and the BS have an incremental construction, hence they handle the growth of the database naturally. In any case, most indexes will be affected by database growth such that expected performance may vary. The randomly generated databases of dimensions 16, 32, and 64 were used, each with sizes $3 \times 10^5, 10^6$, and 3×10^6. Each point in a curve is produced by a different database size; speedups are larger as n increases. As the database grows, so does the size of the indexes and it could not fit on main memory.Because of that, the number of initial neighbors was set to 8 for the APG's and BS.

Figure 4.11(a) shows the recall and the search speed for the different sizes of the database for 16 dimension. The APG and APG* scored a recall below 0.7 on all the sizes, even when using 32 searches for the query. As shown in the previous experiments, the APG*-R gets a boost on the

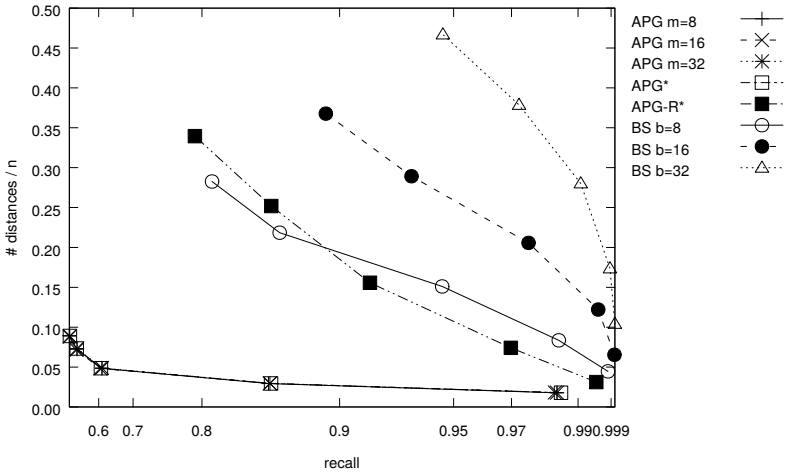


(a) ***1 to 10 integers***

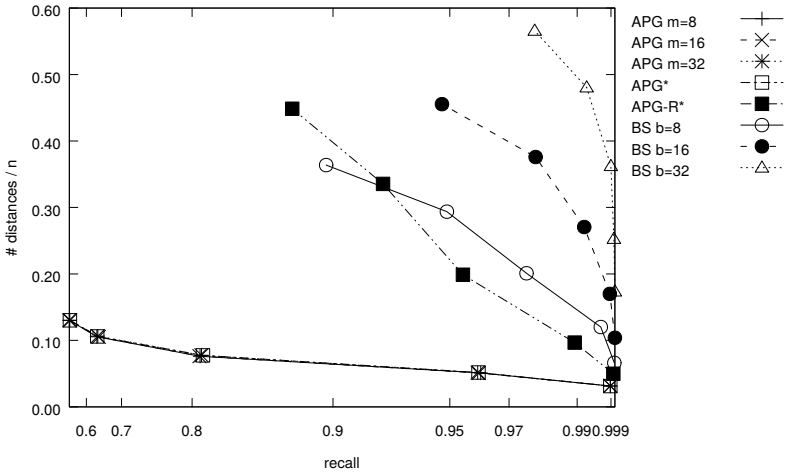


(b) ***10 to 30 integers***

Figure 4.7: Performance comparison (recall vs search cost) among the proposed indexes and several state of the art techniques for fixed $n = 10^5$ over several dimensions,with low to medium memory requirements (i.e., 1 to 30 integers per item). Each curve corresponds to a different dimension $16, 32, 64, 128$, and 256; as a hint to review figures, large dimensions correspond to smaller recalls and scale is exponential.

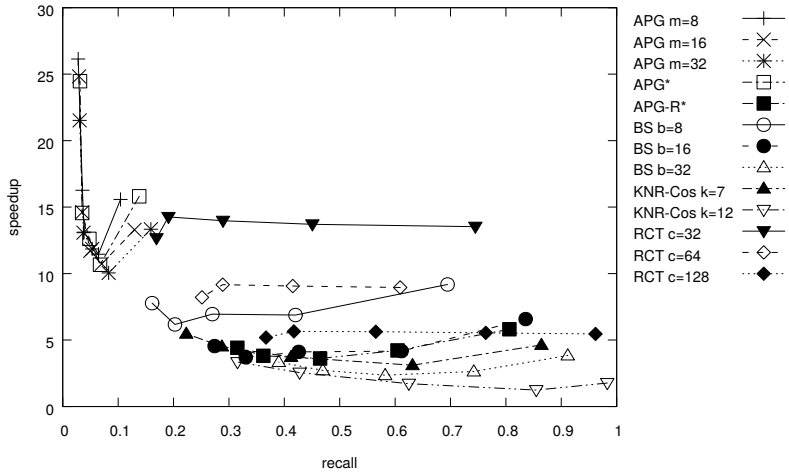


(a) ***30 to 100***

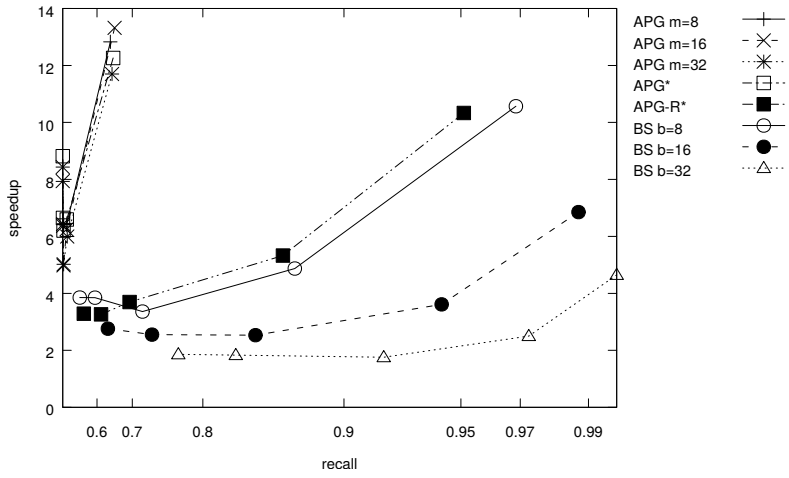


(b) ***100 to 300 integers***

Figure 4.8: Performance comparison (recall vs search cost) among the proposed indexes and several state of the art techniques for fixed $n = 10^5$ over several dimensions. Each curve corresponds to a different dimension $16, 32, 64, 128$, and 256; as a hint to review figures, large dimensions correspond to smaller recalls and scale is exponential.

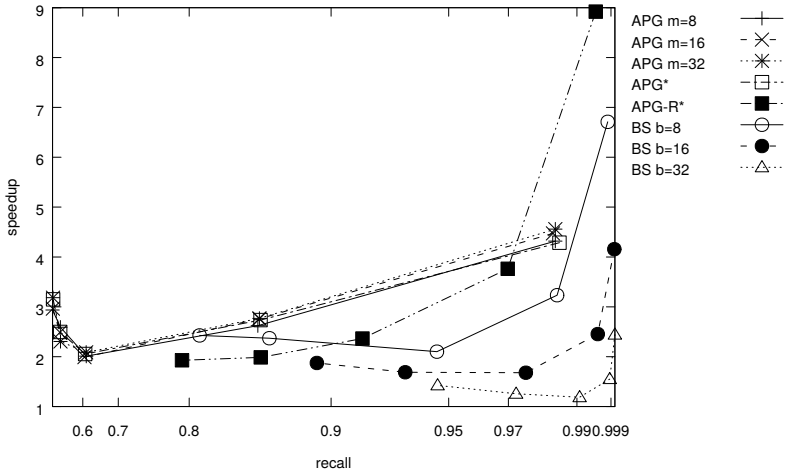


(a) ***1 to 10 integers***

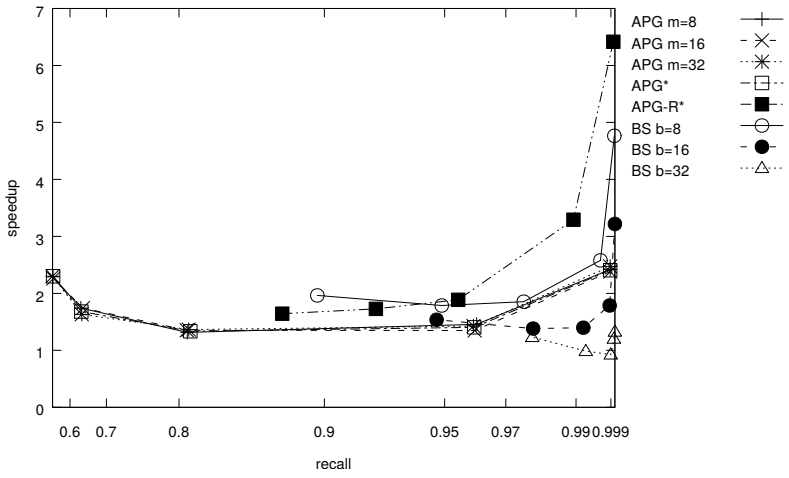


(b) ***10 to 30 integers***

Figure 4.9: Recall and speedup comparison in four different classes of memory among the proposed indexes and several state of the art techniques for fixed $n = 10^5$ over several dimensions. Each curve corresponds to a different dimension $16, 32, 64, 128$, and 256; as a hint to review figures, large dimensions correspond to smaller recalls and scale is exponential.



(a) ***30 to 100 integers***



(b) ***100 to 300 integers***

Figure 4.10: Recall and speedup comparison in four different classes of memory among the proposed indexes and several state of the art techniques for fixed $n = 10^5$ over several dimensions. Each curve corresponds to a different dimension $16, 32, 64, 128$, and 256; as a hint to review figures, large dimensions correspond to smaller recalls and scale is exponential.

Table 4.2: Values for the search cost and recall for the different sizes of the random database of dimension 16.

index	10^5		3×10^5		10^6		3×10^6	
	Cost	Recall	Cost	Recall	Cost	Recall	Cost	Recall
APG*-R	1868.84	0.95	3561.40	0.97	4002.21	0.96	4208.77	0.96
BS b=8	2063.57	0.96	2297.48	0.95	2473.05	0.95	2705.92	0.94
BS b=16	3137.97	0.98	3566.05	0.98	3925.69	0.98	4217.09	0.97
BS b=32	4937.93	0.99	5621.50	0.99	6200.69	0.99	6637.78	0.98
KNR-Cos k=7	4553.08	0.85	9998.99	0.92	28686.36	0.95	83004.65	0.97
KNR-Cos k=12	4950.76	0.92	10948.57	0.96	31941.04	0.98	91945.43	0.99
RCT			15836.4	0.95	26863.9	0.94	42947	0.94

Table 4.3: Values for the search cost and recall for the different sizes of the random database of dimension 32.

index	10^5		3×10^5		10^6		3×10^6	
	Cost	Recall	Cost	Recall	Cost	Recall	Cost	Recall
APG*-R	4484.40	0.86	9276.67	0.88	12865.79	0.82	14168.41	0.82
BS b=8	4405.64	0.87	5529.78	0.83	6480.63	0.78	7309.23	0.73
BS b=16	6565.40	0.94	8105.05	0.91	10078.76	0.88	11446.03	0.84
BS b=32	9450.75	0.97	12122.14	0.95	15067.28	0.94	17256.64	0.91
KNR-Cos k=12	5105.69	0.65	11100.57	0.71	32088.94	0.78	92088.18	0.83
RCT			19229.8	0.71	34085.0	0.67	52872.1	0.63

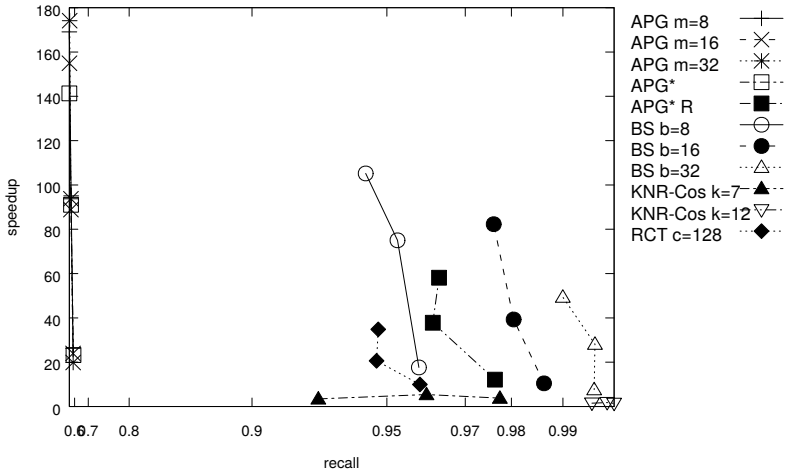
recall compared to the APG*. Note how the lines of the BS are practically vertical meaning that the index gains in speed much more than it loses in the recall. The RCT does not get a recall as good as the BS and is considerable slower. On this and the other dimensions, the sizes corresponding to the lines of each method are $3 \times 10^5, 10^6$, and 3×10^6 from right to left except for the KNR. For this method, the recall is getting better as the size of the database increases. On Figure 4.11(b) we see the recall and the number of distances for the queries. This general behavior is the same in the next figures.

The Figure 4.12(a) shows the same tendency: the APG*-R been an improvement over the APG's, the BS having very good recalls and speeds, the KNR been slow, and the RCT behind the BS. The Figure 4.12(b) shows the same but with the distances computed. The same can be said to the figures 4.13(a) and 4.13(b) but the interesting part is to note how the change of dimension affects more the indexes than the size of the database for reaching high recalls. For 16 dimensions, good recalls can be achieve for all the sizes; for 32 and 64 dimension, the BS always gets better recalls.

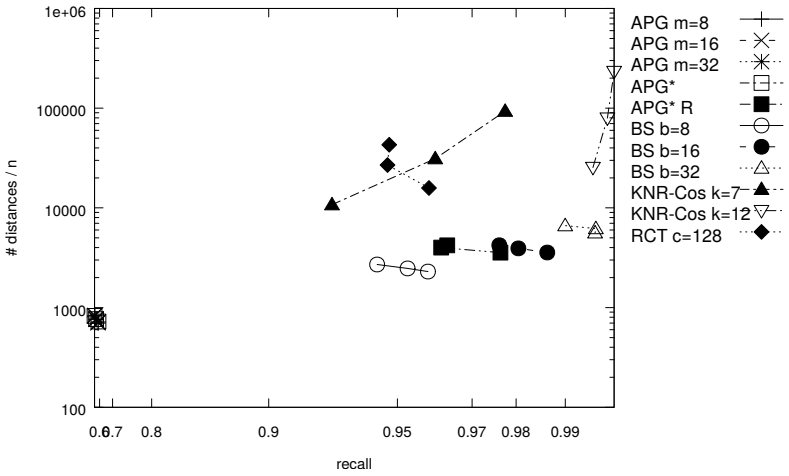
As a summary, the results for the methods with best recall values are shown on Tables 4.2, 4.3, and 4.4 for 16, 32, and 64 dimensions respectively.

Table 4.4: Values for the search cost and recall for the different sizes of the random database of dimension 64.

index	10^5		3×10^5		10^6		3×10^6	
	Cost	Recall	Cost	Recall	Cost	Recall	Cost	Recall
APG*-R	10357.14	0.69	21958.30	0.74	35579.05	0.61	51994.29	0.57
BS b=8	8181.72	0.71	11582.28	0.62	16384.80	0.50	22398.66	0.40
BS b=16	11692.90	0.84	16907.20	0.76	24275.12	0.66	32656.85	0.56
BS b=32	16510.76	0.91	24060.58	0.86	35590.51	0.79	49443.82	0.71

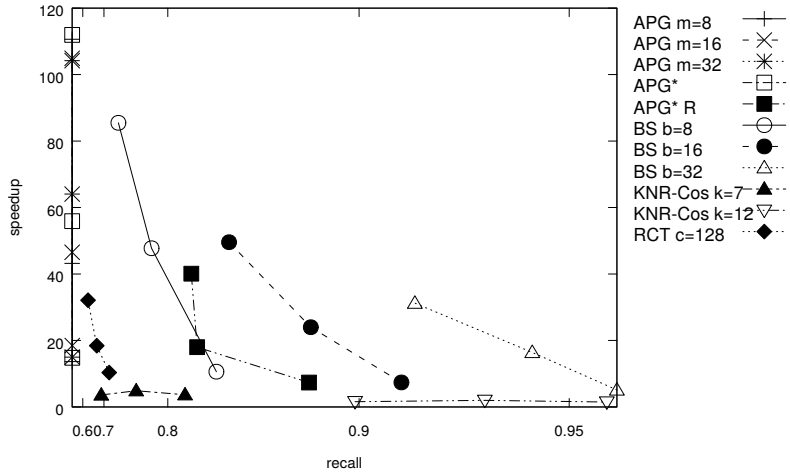


(a) ***Search speedup / 16 dimensions***

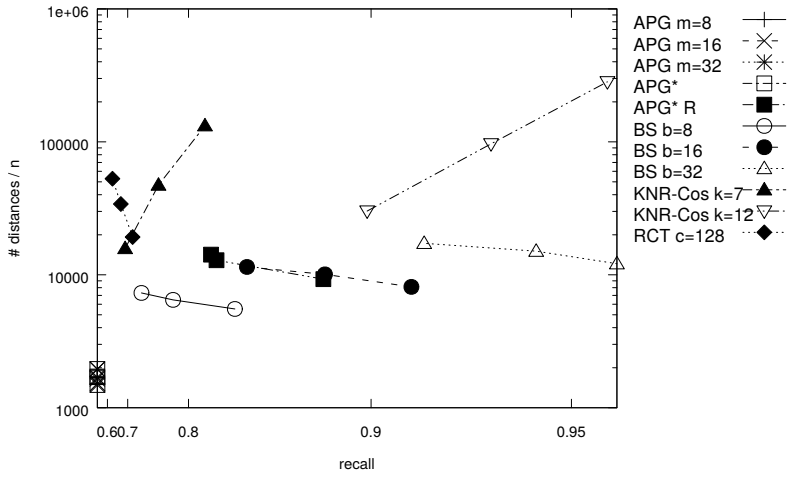


(b) ***Search cost / 16 dimensions***

Figure 4.11: Speedup and search cost as function of the expected recall for the presented indexes and the state of the art. The datasets used had dimensions 16. Each dataset has three instances with sizes of 3×10^5, 10^6, and 3×10^6. the memory was limited to be under 8 direct neighbors per item, i.e. 16 undirected neighbors in average.

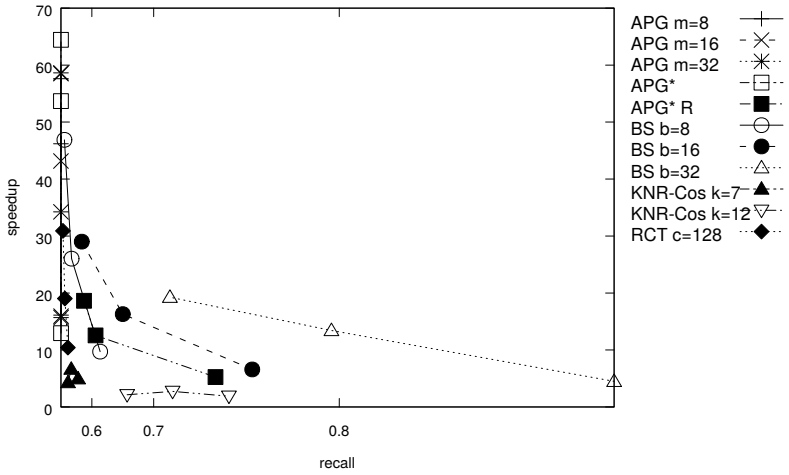


(a) ***Search speedup / 32 dimensions***

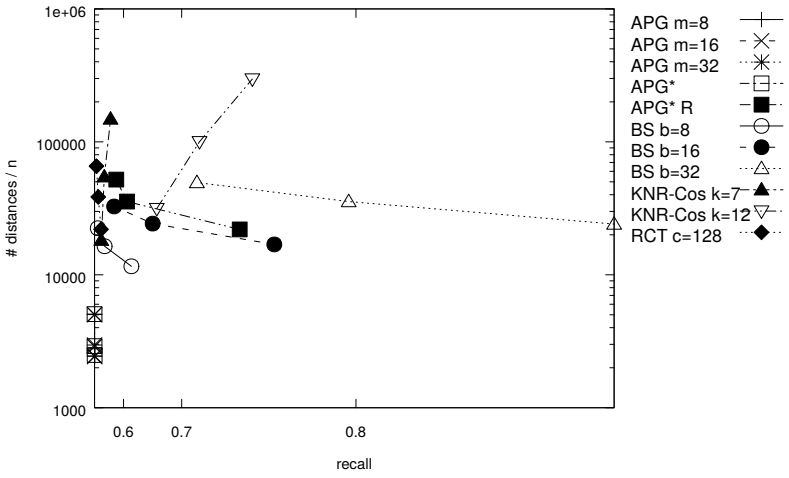


(b) ***Search cost / 32 dimensions***

Figure 4.12: Speedup and search cost as function of the expected recall for the presented indexes and the state of the art. The datasets used had dimensions 32. Each dataset has three instances with sizes of $3 \times 10^5, 10^6$, and 3×10^6. the memory was limited to be under 8 direct neighbors per item, i.e. 16 undirected neighbors in average.



(a) ***Search speedup / 64 dimensions***



(b) ***Search cost / 64 dimensions***

Figure 4.13: Speedup and search cost as function of the expected recall for the presented indexes and the state of the art. The datasets used had dimensions 64. Each dataset has three instances with sizes of $3 \times 10^5, 10^6$, and 3×10^6. the memory was limited to be under 8 direct neighbors per item, i.e. 16 undirected neighbors in average.

4.5 Tuning APG*, APG*-R, and BS

The diversity of the combinatorial optimization applied for proximity search may be problematic for an end-user. In this section, a simple guide is provided for finding a competitive index.

The first step is the selection of the index. Even when APG* performs as good as APG, APG*-R and BS perform better. Both APG*-R and BS need to know the number of direct near-neighbors t. APG*-R has no more explicit parameters since it adjusts the number of repetitions based on a simple iterative optimization; then the robustness is entirely based on t. For this reason, t should be relatively large, i.e., bigger than 16 lesser than 64. Since the underlying graph is undirected, which means that the memory usage will be $n(2t + 1)$ integers, the right selection of t should consider this amount of main memory for the adequate performance. For the case of BS, t can be smaller, i.e., 8 or 16; larger values of t should be used for very high intrinsic dimensions, as described in the experimental section. After fixing t, the free parameter b (the size of the beam) should be adjusted to obtain the desired performance. A large value of b will produce high-quality search results while search time will increase. In contrast, faster searches will be achieved for small b values, yet the quality of the result set could decrease. As a rule of thumb, the recommendation is to fix b to the largest value producing an affordable search time. Nonetheless, good recall values are achieved for most of the datasets for b values between 16 to 64. A more precise setup of b could be accomplished with the help of a query set Q. The idea is to adjust b to obtain a minimum recall on Q; this fitting should be applied after a fixed number of insertions.

4.6 Summary and Future Work

The principal contribution in this chapter was the presentation of the proximity search as a combinatorial optimization problem. For this, the problem was segmented to allow the use of combinatorial optimization algorithms largely studied in the literature. In particular, this contribution focuses on local search algorithms.

Using this new perspective, three near neighbor searching indexes were introduced, called APG*, APG*-R, and BS. These indexes use an underlying graph and several variants of local search to navigate them. As part of this contribution, the indexes were compared among the current state

of the art indexes with excellent results. In particular, the presented indexes are of practical use due to their very good trade-off among speed, recall, and index size.

In this thesis, the $\mathcal{N}$ function was fixed to be that defined by APG while the focus was on finding better algorithms for navigating it, i.e., $\mathcal{A}$. However, new $\mathcal{N}$ definitions with improved or additional properties should be welcome.

Since the presented indexes are incrementally constructed, the insertion operation is native. However, the deletion algorithm is not yet studied. The literature in metric indexes used to solve the deletion operation marking those items as unavailable for most operations [Chavez01]. While this is enough for workloads with just a few deletions, a high rate of removals could yield to a significant unnecessary extra memory.

In other approaches, the metric indexes can afford real deletions. A recent approach uses the strategy of Bentley and Saxe ([Bentley80]) to produce dynamic structures using a list of $\log n$ static structures. This approach gives the possibility of use any static indexes, yet at the cost of a $\log n$ factor in the search cost. The delete operation deserves attention in the future research on the area.

Chapter 5

Spanners from Proximity

5.1 Introduction

In the previous chapter, the main problem is, given a set of points, how to generate a graph over them and how to navigate this graph to go from one point to another without examining the full graph. The routing process should be obtained only with local information. Note that in the past chapter, the start of the search was a point in the graph meanwhile the destination point was not. Here we restrict the problem to the case where both points are on the graph.

This variant has applications in *ad-hoc* networks which consist of transmitters, often called hosts and the network is established as needed, typically without any assistance from a fixed infrastructure. It is assumed that each host can communicate with all the hosts within its transmission range with a single transmission, called a hop. Typically, not all hosts are within the transmission range of each other and the transmission ranges of all hosts are identical. We will additionally assume that each host knows its location, its coordinates in the plane, obtained by a low energy GPS device or by other means.

Since not all the hosts are connected, a message from one host to another must be passed from host to host until it reaches its destination. Each communication between hosts consumes energy that depends on the distance of the source and the receptor. So, the problem is to send a message trying to minimize the energy cost.

The network defines a graph and it must be traverse to get a path between any two points.

If this graph has many edges, the energy efficiency will be high but the finding the path will be slow. We just saw this behavior in the previous chapter, if the graph has many edges, the search will be slow. So, we need to extract a subgraph with less edges but with the property to go from one point to another with just local information.

5.2 Proximity Navigation Graphs

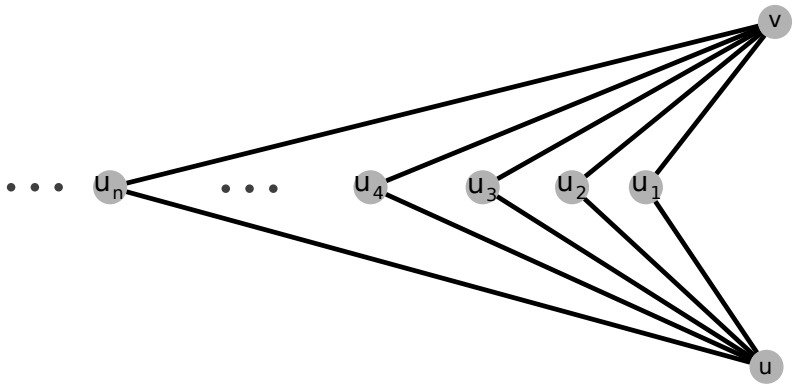
We start with some definitions to establish the problem we want to study. Let $(X, \mathbf{d})$ be a metric space, $V \subseteq X$ a finite set, and $E \subseteq V \times V$. The graph $G(E, V)$ is a *metric* graph if each edge $(u, v) \in E$ has weight $\mathbf{d}(u, v)$. In particular, the edge (u, u) is in the graph and have distance 0. A graph $G(E, V)$ is said to be *strongly connected* if there is a path from u to v for any $u, v \in V$. All the graphs considered in this chapter will be strongly connected. Define G a *proximal navigation graph* or PN-graph if for each $u, v \in V$ either $(u, v) \in E$ or there is a node u_1 such that $(u, u_1) \in E$ and $\mathbf{d}(u, v) > \mathbf{d}(u_1, v)$. This means that there is always a path connecting arbitrary nodes u, v such that the distance of the nodes in the path to v decrease monotonically. Let $\mathbf{d}_G(u, v)$ be the length of the shortest path from u to v in G. A graph G is a *t-spanner* if for every pair of points u and v in G, we have $\mathbf{d}_G(u, v) \leq t\mathbf{d}(u, v)$ where $\mathbf{d}_G(u, v)$ is the length of the smallest path between u and v. The parameter t is called the stretch factor of G.

We say that a family of strongly connected graphs is a *t-spanner* if every graph G in the family is a t-spanner. Note that each graph on the family has the same t as stretch factor.

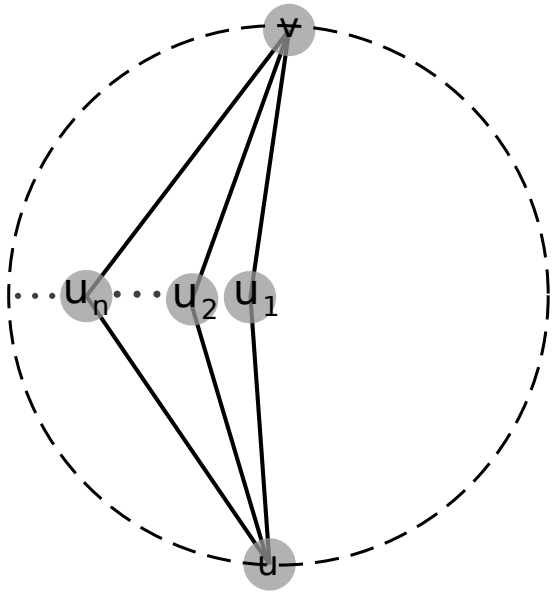
Figure 5.1(a) shows an example of a family of strongly connected graphs that is not a t-spanner. The members of the family are the graphs defined by the vertices u, u_i, and v for $i = 1, 2, \ldots$, so, for example the graph of vertices u, u_1, and v is a t_1 spanner, but the graph of vertices u, u_2, and v is a t_2 spanner for $t_2 > t_1$ and so on. There is not a t that will made all the members of the family t-spanners. The Figure 5.1(b) shows the same family but this time the u_i vertices are bounded so all the graphs are $\sqrt{2}$-spanners.

In a PN-graph G, the *proximity path* from u to v, $\mathbf{PP}(u, v)$, is constructed as follows: We start in u and continue with the neighbor of u closer to v and repeat until we reach v. In general, the proximity path of two points is not the shortest path between them.

Remark 1 *The path $\mathbf{PP}(u, v)$ is always contained in the ball center at v with radius $\mathbf{d}(u, v)$,*



(a) ***An example of a non t-spanner family.***



(b) ***A t-spanner family.***

Figure 5.1: The members of the family are the graphs defined by the vertices u, u_i, and v for $i = 1, 2, \ldots$

$B_{\mathbf{d}(u,v)}(v)$.

The Complete Graph (where every point is connected) is trivially both a 1-spanner and a PN-graph. Another more interesting example is the *Delaunay triangulation* (DT), defined for a set V of points in the plane, as the triangulation such that no point in V is inside the circumcircle of a triangle in DT. It is well known that DT is a $\frac{4\pi}{3\sqrt{3}}$-spanner. It is also known that DT is a PN-graph, see for example [Bose04].

5.2.1 Are Proximal Navigation Graphs t-Spanners?

PN-graphs have the property that they can be navigated using only local information, if we want to go from one point to any other, we can just follow the proximity path and we have the certain that we will reach the desired point. This makes the PN-graphs useful for communication networks. In this domain, it is also important that the length of the communication path is as short as possible. The t-spanner graphs with small t guarantee this last desired property. It is then important to know if the PN-graphs are t-spanners.

Conjecture 1 *Every family of proximity navigation graphs is a t-spanner.*

Let us assume this conjecture 1 is false, then there must exist a family of proximity navigation graphs such that for all $t > 0$ there is a graph H in the family and two points u and v in H such that $\mathbf{d}_H(u,v) > t\mathbf{d}(u,v)$. Note that the length of every path from u to v is also greater than $t\mathbf{d}(u,v)$.

If we have two points u and v, let $\mathbf{R}_r(x)$ be the ring of points such that their distance to v is greater than $\mathbf{d}(v,x) - r$ and smaller than $\mathbf{d}(v,x) + r$.

Let P be the proximity path between u and v, $P = \mathbf{PP}(u,v)$, the *length inside* the set A in P will be the length of the path P from the farthest point to v in $A \cap P$ to the closest point to v in $A \cap P$.

Lemma 2 *Let $\mathcal{G}$ be a family of proximity navigation graphs that is not a t-spanners. Then for all $M > 0$ and $\epsilon > 0$ there exist a proximity navigation graph $G \in \mathcal{G}$ and points x, u, v in G such that the length inside the ring $\mathbf{R}_{\epsilon\mathbf{d}(u,v)}(x)$ in P is greater than $M\mathbf{d}(u,v)$. We will say that $\mathcal{G}$ has an* unbounded approximation point.

Proof 5.2.1 *Suppose this lemma is false, then there exist $M > 0$ and $\epsilon > 0$ such that for all proximity navigation graphs $G \in \mathcal{G}$ and all x, u, v in G, the length inside $\mathbf{R}_{\epsilon \mathbf{d}(u,v)}(x)$ of $P = \mathbf{PP}(u, v)$ is smaller than $M\mathbf{d}(u, v)$.*

Let y_1 be the last point from P in that ring, following P, y_1 is connected to x_2, the length inside $\mathbf{R}_{\epsilon \mathbf{d}(u,v)}(x_2)$ of P is less than $M\mathbf{d}(u, v)$. Let y_2 be the last point from P in that last ball, following P, y_2 is connected to x_3. We continue in this way.

Note that when we go from x_i to x_{i+1} we get closer to v at least $\epsilon\mathbf{d}(u, v)$, hence the number of points in the sequence $x_1, x_2, \cdots, x_i, \cdots$ is at most $1/\epsilon$. Then $\mathbf{d}(u, v) \leq \frac{M\mathbf{d}(u,v)+2\mathbf{d}(u,v)}{\epsilon} = \frac{M+2}{\epsilon}\mathbf{d}(u, v)$ for all $G \in \mathcal{G}$ and all x, u, v in G, that is, $\mathcal{G}$ is t-spanner, which is a contradiction.

Theorem 5.2.1 *Let $\mathcal{G}$ be a family of proximal navigation graphs. If $\mathcal{G}$ is not a t-spanner, then $\mathcal{G}$ has an unbounded approximation point. If $\mathcal{G}$ does not have an unbounded approximation point, then $\mathcal{G}$ is a t-spanner.*

Proof 5.2.2 *The first implication is the last lemma. For the last implication let us suppose that $\mathcal{G}$ is not a t-spanner, then for the first implication we have that $\mathcal{G}$ should have an unbounded approximation point which is a contradiction.*

Theorem 5.2.2 *Let $\mathcal{G}$ be a family of proximal navigation graphs over $X = \mathbb{R}^n$. If the distance $\mathbf{d}$ is bounded, i.e. $0 < d < \mathbf{d} < D$ for all points in X, then $\mathcal{G}$ is a t-spanner.*

Proof 5.2.3 *We will prove it for $n = 2$, the other cases are similar. For every $H \in \mathcal{G}$ and every u and v in H, we have that the $R_{d/2}(u)$ has at most $\pi D/d$ points, since the distance is at most D, the inside length is at most $\pi D^2/d$. Let y_1 be the last point from $\mathbf{PP}(u, v)$ in that ring, following the path, y_1 is connected to u_2, the length inside $\mathbf{R}_{d/2}(u_2)$ is less than $\pi D^2/d$. Let y_2 be the last point from $\mathbf{PP}(u, v)$ in that ring, following the path, y_2 is connected to u_3. We continue in this way. Then the $\mathbf{PP}(u, v)$ has length at most*

$$\left(\frac{\pi D^2}{d + D}\right) \frac{\mathbf{d}(u, v)}{\frac{d}{2}}$$

Then, $\mathcal{G}$ is a t-spanner.

Corollary 5.2.3 *If the distance function $\mathbf{d}$ is discrete and bounded, then every family of PN-graphs $\mathcal{G}$ in $(\mathbb{R}^n, \mathbf{d})$ is a t-spanner.*

An example of the corollary is that a family of PN-graphs in the metric space of bitmaps of length m using the hamming distance is a t-spanner: if we have a family in this metric space, then for every two bitmaps u and v, we have $\mathbf{d}(u, v) \leq m$ and each time we move from u to v the distance decreases at least by 1, then we reach v in at most m points. Then this family cannot have an unbounded approximation point.

5.3 HSP

A special example of a PN-graph is the Half Space Proximal (HSP)[Chávez05a]. To obtain the HSP of a finite point set V each $u \in V$ computes their HSP neighbors $HSP(u)$ in a distributed manner. The *allowed region* is initially all the points V, let $A = V$ be that set. The Nearest Neighbor (NN) of u in A is computed and added to $HSP(u)$. All points in A closer to $NN(u)$ than to u are removed from A, this removed set is called the *forbidden region* and the process is repeated until A, the allowed region is empty. Below we prove that the HSP is a PN-graph.

Lemma 3 *If there is not an edge from the point u to v in the HSP, then there exist a point z such that it connects to u and z is in the intersection of the two circles centered at u and v with radius $\mathbf{d}(u, v)$.*

Proof 5.3.1 *If there is not an edge from the point u to v, then, v is in a forbidden region of u and some point z. The point z connects to u and is closer to u than from v, so, z is in the circle centered at u with radius $\mathbf{d}(u, v)$. As v is in the forbidden region of u with respect to z, then v is closer to z than to u, so, z is in the circle centered at v with radius $\mathbf{d}(u, v)$.*

Lemma 4 *If there is an edge from the point u to v in the HSP, then there is no point in the circle centered at v with radius $\mathbf{d}(u, v)$ connected to u.*

Proof 5.3.2 *If there is an edge from the point u to v, then there is no point in $B_{\mathbf{d}(u,v)}(u) \cap B_{\mathbf{d}(u,v)}(v)$ because if there was, then it would leave v in its forbidden region. There is also not a point in the forbidden region of v, so the lemma is proved.*

Lemma 5 *The HSP is a proximal navigation graph.*

Proof 5.3.3 *From lemma 3, if (u, v) is not an edge in the HSP, then there exist a point z connected to u, with z in the interior of the intersection of the two circles centered at u and v respectively, and with radius $\mathbf{d}(u, v)$. This implies, $\mathbf{d}(z, v) < \mathbf{d}(u, v)$. The argument can be repeated on the pair z and v until we reach v to get a path from u to v as the path in the definition of a PN-graph.*

5.3.1 A More General HSP

The HSP can be generalized if we see the algorithm as follows:

1. Take a point p in the set.
2. Sort all the other points in a list with respect with the distance to p.
3. Take the first element e off the list.
4. Add e to the neighbors of p.
5. Eliminate all the points from the list that are closer to e than to p.
6. Repeat from 3 until the list is empty.

The sorting order of step 2 can be changed for the inverse order. With this change we will call *Half Space Distal* (HSD) to the resulting graph. Note that we could put any order in step 2, even a random order. We will call *Half Space Random* (HSR) to that graph.

With these variations the properties change, for example, the HSR and HSD may have out-degree bigger than 6 in the plane. But some others are still unaffected.

Lemma 6 *The HSD and HSR are strongly connected.*

Proof 5.3.4 *Let $u, v \in V$. If $(u, v) \in E$, they are connected. If (u, v) is not in E, there is a point connected to u that left v in a forbidden region, i.e., there should be a $z \in V$ such that $(u, z) \in E$ and $\mathbf{d}(v, z) < \mathbf{d}(v, u)$. We repeat the process with z and v, eventually we will get v because V is finite and all the points we are finding are different.*

Lemma 7 *The HSD and HSR are PN-Graphs.*

Proof 5.3.5 *This proof is very similar to the last one. Let $u, v \in V$. We will show that we could go from u to v greedy. If $(u, v) \in E$ there is nothing else to do. If (u, v) is not in E, there is a point connected to u that left v in a forbidden region, i.e., there should be a $z \in V$ such that $(u, z) \in E$ and $\mathbf{d}(v, z) < \mathbf{d}(v, u)$.*

We will say that a PN-graph (E, V) is minimal if for every $e \in E$, $(E \setminus e, V)$ is not a PN-graph.

Lemma 8 *The HSP is minimal.*

Proof 5.3.6 *Let the HSP=(E,V) and let $e = (u, v) \in E$. By the lemma 4 there is not a point in $\mathbf{B}_{\mathbf{d}(u,v)}(v)$ connected with u, so if we remove the edge e, we could not go from u to v greedy, so the graph would not be a PN-Graph.*

The last lemma is not true for the HSR or the HSD.

A PN-graph is *distributed* if for every edge (u, v), u is not connected to someone in $\mathbf{d}(u, v)$. The sense of distributed PN-graphs is that the points have a neighbor *in each each direction*.

Lemma 9 *The HSP is distributed.*

Proof 5.3.7 *Immediate from the lemma 4.*

Lemma 10 *For every distributed PN-graph $G = (E, V)$, there is a $G' = HSR(V)$ such that $G = G'$.*

Proof 5.3.8 *Let $u \in V$ and $N(u) = \{u_1, u_2, \ldots, u_n\}$ be the neighbors of u in G. The point u is not connected to anyone in $\mathbf{B}_{\mathbf{d}(u,u_i)}(u_i)$ for every i since G is distributed. Then there is no point connected to u in the forbidden region of u_i. This implies that if we take the order $\{u_1, u_2, \ldots, u_n, \ldots\}$ for u, its neighbors in the HSR will be $\{u_1, u_2, \ldots, u_n\}$, we only need to verify that there will not be any more, but that is true because for every point in $v \in V$, we can go from u to v greedy through a u_i because G is a PN-graph and then v will be in the forbidden region of u_i. Then, for every point in V, we can find an order such that G and G' have the same edges.*

Then, all the distributed PN-graphs are in HSR, the inverse is not true since there are HRS graphs that are not distributed. The important conclusion is that the HSP has the best properties and arises naturally among the PN-graphs.

5.4 A Family of Proximal Navigation Graphs that it is not a t-spanner

For $\epsilon > 0$ let X be the following set of functions from [0,1] to [0,1]

$$f_0(x) = 0$$

$$f_1(x) = \begin{cases} 0 & \text{if } x \leq 1/2 - \epsilon \\ 1 & \text{if } 1/2 - \epsilon < x \leq 1/2 \\ 1/2 & \text{if } x > 1/2 \end{cases}$$

$$f_2(x) = \begin{cases} 0 & \text{if } x \leq 1/2 - 2\epsilon \\ 1 & \text{if } 1/2 - 2\epsilon < x \leq 1/2 \\ 1/3 & \text{if } x > 2/3 \\ f_1(x) & \text{elsewhere} \end{cases}$$

$$f_3(x) = \begin{cases} 0 & \text{if } x \leq 1/2 - 3\epsilon \\ 1 & \text{if } 1/2 - 3\epsilon < x \leq 1/2 \\ 1/4 & \text{if } x > 3/4 \\ f_2(x) & \text{elsewhere} \end{cases}$$

$$\vdots$$

$$f_i(x) = \begin{cases} 0 & \text{if } x \leq 1/2 - i\epsilon \\ 1 & \text{if } 1/2 - i\epsilon < x \leq 1/2 \\ 1/(i+1) & \text{if } x > i/(i+1) \\ f_{i-1}(x) & \text{elsewhere} \end{cases}$$

$$\vdots$$

$$f_\infty(x) = 1$$

Let $\mathbf{d}_X \colon X \times X \to \mathbb{R}$ be such that $\mathbf{d}_X(f, g) = \mu(\{x \in [0, 1] | f(x) \neq g(x)\})$ where μ is the Lebesgue measure. Table 5.1 show some example distances between some of the functions.

We will see that the function $\mathbf{d}_X$ is a distance over the set X. The complicated part is the

	f_0	f_1	f_2	f_3	$\cdots$	f_∞
f_0	0	$1/2+\epsilon$	$1/2+2\epsilon$	$1/2+3\epsilon$		1
f_1	$1/2+\epsilon$	0	$1/3+\epsilon$	$1/3+2\epsilon$		$1-\epsilon$
f_2	$1/2+2\epsilon$	$1/3+\epsilon$	0	$1/4+\epsilon$		$1-2\epsilon$
f_3	$1/2+3\epsilon$	$1/3+2\epsilon$	$1/4+\epsilon$	0		$1-3\epsilon$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$		$\vdots$
f_∞	1	$1-\epsilon$	$1-2\epsilon$	$1-3\epsilon$		0

Table 5.1: Distances between functions

triangle inequality. We want to prove that $\mathbf{d}_X(f_x, f_y) + \mathbf{d}_X(f_y, f_z) \geq \mathbf{d}_X(f_x, f_z)$. Is easy to prove it if two of the tree functions are equal or if $z = \infty$. Without lose of generality, $x < z$, and we have tree cases.

Case $y < x$ We have

$$\mathbf{d}_X(f_x, f_z) = \frac{1}{x+2} + (z-x)\epsilon$$

but, since $z - y > z - x$ then $z - x < z - y + x - y$. Also we have $\frac{1}{x+2} < \frac{1}{y+2} < \frac{2}{y+2}$.

$$\frac{1}{x+2} + (z-x)\epsilon < \frac{2}{y+2} + (z-y+x-y)\epsilon$$

$$= \frac{1}{y+2} + (x-y)\epsilon + \frac{1}{y+2}(z-y)\epsilon = \mathbf{d}_X(f_x, f_y) + \mathbf{d}_X(f_y, f_z)$$

Case $x < y < z$ Here we have $\frac{1}{y+2} < \frac{1}{x+2}$ and

$$\mathbf{d}_X(f_x, f_y) + \mathbf{d}_X(f_y, f_z) = \frac{1}{x+2} + (y-x)\epsilon + \frac{1}{y+2}(z-y)\epsilon$$

$$> \frac{1}{x+2} + (y-x)\epsilon + \frac{1}{x+2}(z-y)\epsilon = \frac{2}{x+2}(z-x)\epsilon > \frac{1}{x+2}(z-x)\epsilon = \mathbf{d}_X(f_x, f_z)$$

Case $z < y$ In this case $z - x < y - x$ and

$$\mathbf{d}_X(f_x, f_z) = \frac{1}{x+2} + (z-x)\epsilon < \frac{1}{x+2} + (y-x)\epsilon$$

$$< \frac{1}{x+2} + (y-x)\epsilon + \frac{1}{z+2} + (y-z)\epsilon = \mathbf{d}_X(f_x, f_y) + \mathbf{d}_X(f_y, f_z).$$

We have shown that $\mathbf{d}_X$ is a distance function.

Let $\mathcal{F}$ be the family of graphs $(E_1, V_1), (E_2, V_2), \ldots, (E_k, V_k), \ldots$ where

$$E_i = \{\{f_0, f_1\}, \{f_1, f_2\}, \ldots, \{f_k, f_{k+1}\}, \ldots, \{f_i, f_\infty\}\},$$

$$V_i = \{f_\infty, f_0, f_1, \ldots, f_i\}.$$

We will show that every graph in the family $\mathcal{F}$ is a proximity navigation graph for a suitable ϵ. We need to show that for every pair of points in the graph, u and v, there is a path from u to v that is getting closer to v. From the table of distances the result is clear if one of the points is f_0 or $f\infty$. If one point is f_i the result is also clear if we see the distances from f_i to the other points:

$$\frac{1}{2} + i\epsilon, \frac{1}{3} + (i-1)\epsilon, \frac{1}{4} + (i-2)\epsilon, \ldots, \frac{1}{i+1} + \epsilon, 0, \frac{1}{i+2} + \epsilon, \frac{1}{i+2} + 2\epsilon, \frac{1}{i+2} + 3\epsilon, \ldots, 1 - i\epsilon$$

in this list, we have the distances from f_i to $f_0, f_1, \ldots, f\infty$.

The family $\mathcal{F}$ is not a t-spanner because for every $t > 0$ we can find a graph G in $\mathcal{F}$ such that the $\mathbf{d}_G(f_0, f\infty) > t\mathbf{d}_X(f_0, f\infty)$=t. That is because the distance in the graph G from f_0 to f_∞ is

$$\frac{1}{2} + \epsilon + \frac{1}{3} + \epsilon + \frac{1}{4} + \epsilon + \cdots$$

which is a non convergent series. The Conjecture 1 is false.

Remark 2 *If we apply the HSP to this family, we get the same graphs, hence the HSP is not always a t-spanner.*

5.5 Summary

The HSP and its variants have the potential of being of use in many applications because what they do is to take a point on every direction of the space and works for abstract metric spaces. The number of neighbors of a point depends on the intrinsic dimension of the data and on the position of the point in the cloud. A point in the center of the dataset will have a higher number of neighbors than a point in the boundary. This information could be used to classify the points by their position. But that is another story.

Chapter 6

Conclusions and Future Work

This work presents indexes that auto-adjust some parameters in order to simplify their use. Some indexes have parameters not easily to tune and many times, the only viable option is to make a guess. Some other times, many instances of an index should be constructed to find the one with the best performance. Some other indexes do not have parameters at all, they are easy to use but their space and speed can not be changed. The best of this two worlds is to have a tunable index with easy parameters.

The main part of this work is the introduction of the EPT, ANNI, APG*, APG*-R, and BS indexes. They improved the state of the art alternatives in multiple occasions. We cover the cases of exact and approximate searches. Analysis of their theoretical performance are included and used to make the construction algorithms able to find the best parameters.

The EPT shifts the traditional way to create indexes. A lot of effort has been put in selecting a good set of pivots for a given dataset, the EPT assigns the best points to a given set of pivots. This change has proved to be very effective. The EPT assumes some distribution of the data, pivots and queries; for the cases where that can not be true was designed the ANNI. Here, the construction of the index consists of improvements based on real searches. In this new schema a time-optimization is possible. This is very difficult to do with other indexes because the real time of the search can depend on many things beside the index itself. Those indexes are good but, for high intrinsic dimension sets, an approximate index is needed. The APG is simple and fast, but it has some parameters difficult to tune. The APG* and APG*-R are simpler versions with the same or

best performance than the APG. The main idea behind these indexes is to turn the proximity search problem into an optimization problem. With that, we can use all the techniques from optimization, in particular we used the beam search to produce the BS variation index with excellent results.

None of the above indexes guaranties a faster search than sequential scan on every metric index. The course of the dimension is unavoidable. Besides that, some improvements can be done. For example, a version to work on disk instead of main memory. The difficulty here is the rearrangement of the index structure to avoid random accesses. This study is left for future works. Also, the EPT and ANNI are static which means that the full database must be known at the time of construction and no modification can be made afterwards. A dynamic version of these require further research.

Appendix A

Databases

A.1 Introduction

The use of a database is becoming more and more important due to the high amount of information that we have about almost everything. The appropriate management of this information is crucial to make it useful. On this, we want to find the data we are looking for, and, most importantly, we want to do it fast.

The databases are everywhere, some times we do not see them. An application could use a database to store some information and to retrieve and analyse it. The database could be hidden from the user where the application takes care of the administration. This kind of databases, where the user does not work directly on them, are known as *embedded databases* [Nori07].

There is a growing amount of computers in our lives. The new phones can run a large variety of applications. They can manage small databases and the conventional database systems tend to ignore them.

The form to find information on a text and on a database had been studied in a separated way. But the interest in doing these two types of queries on the same model is growing.

This appendix, describes a method to enable keyword and text searches on relational databases using succinct structures to efficiently use the space needed.

The common way to represent a relational database is a tabular structure, where each table represents a relation [Garcia-Molina09]. Each table is a set of records, and each record contains an

ordered set of fixed attributes called columns or fields. Traditional Database Management Systems (DBMS) use the relational model to store and retrieve the data. The relational model describes the database as a collection of predicates with some variables. It provides a declarative method for the queries.

The Structured Query Language (SQL) is the standard way to specify the queries on this model. A table in an SQL database schema corresponds to a predicate variable; the contents of a table to a relation; and SQL queries correspond to predicates. This model has proved been effective to most cases. However, an operation to efficiently search and locate plain text occurrences is not part of the relational model, hence this operation is not efficiently handled by most DBMS.

In the *full text search*, each word of every document in the database is examined to find a match of the query. Traditional DBMS does not support full text searches, an add-on is needed to perform them. The database model is optimized to find exact matches of fields and searches inside them were not popular until recent years.

A full text search can be done using a sequential scan of all the text in the database. This approach is only practical for small amounts of text. When the sequential scan is too slow, a pre-process of the data is done to build a structure that permits the searches to be executed in much less time. This structure is known as an *index*. In general, an index is a way to organize the data to easily find what we are looking for.

Full-text searches can be done using an *inverted index*. An inverted index is a data structure that maps a search key (like a word) to a set of positions where that key appears in a text or a in a collection of documents. This set is called the *post* of the key. An inverted index can be implemented using a relational database as described in [Putz91].

To support this operation the use of bitmaps is the preferred choice [Stockinger09]. A *bitmap* B is an array with values in $\{0, 1\}$. Moreover, bitmaps do not just help full text searches but can even be faster than trees for standard SQL queries, when the data is static and queries are not very selective [Jürgens99]. An example of this situation is in data warehousing applications. A data warehouse is a database where each data introduced is kept forever and can not be modified. We aim not to implement a fully relational DBMS, but provide the necessary tools to represent efficiently a table while supporting many searching operations using the compressed representation.

One obvious interest on compressing the tables in a database is to reduce the space needed

in storage, but this also improves the query times as well [Lemire12]. A simple technique is to reorder the records on a table to improve Run-Length encoding. Run-Length replaces a single symbol substring (called a *run*) by that symbol and its multiplicity. The purpose of reordering records is the induction of large runs, which naturally compresses the table. Research shows that spending extra time in reordering records yields to a query speed improvement, due to better memory access patterns. So, reordering is justified even on small compression ratios [Lemire12].

Due to the increasing amount of digital information generated everywhere, and the need to represent, store, process and retrieve data, extra effort has been placed in squeezing speed from a variety of sources. Recently a lot of research has been made in compressing and compacting data. The advantage of compressing is not only due to savings in storage space [Garber12]; the holy grail in gaining speed out of compression is to avoid the use of disk even when dealing with relatively large amounts of data. It is well known that main memory is orders of magnitude faster than disk, hence working on RAM yields a higher throughput than using disk. This is true even if the number of disk accesses is relatively small.

Making the database fit in RAM using compression is not enough if we want to process the data or to perform queries on it. This is because, for each transaction, we would need to decompress it. An alternative idea is to compact the information; the difference between compression and compaction is that, compact data does not require decompression for accessing the data. Noticing the difference between compressing and compacting data, both terms will be interchangeable, knowing that these data structures do not need an explicit decompression.

The presented method compresses both, the data and the indexes into the same structure using less memory than the plain representation. Moreover, several alternative configurations were experimented with selective compression trading effectively speed for space. The objective is to use the higher levels of the memory hierarchy present on modern computing devices. The contribution is on gaining speed out of compression by letting more data, and the index, to fit in the same space.

A.2 Basic concepts

In order to understand the basics of this technique, a small introduction to succinct structures is presented.

A.2.1 Entropy

An alphabet Δ of size n is an ordered set of n elements. Let S be a sequence over an alphabet Δ of size n and n_c be the number of occurrences of character c on S for each $c \in \Delta$. The *order zero entropy* is defined as

$$H_0(S) = \sum_{c \in \Delta} \frac{n_c}{n} \log \frac{n}{n_c}. \tag{A.1}$$

H_0 is a lower bound of any codification that assigns a unique code to a character c.

Let A be a substring of S and S_A be the concatenation of the characters right after each occurrence of A in S. The *order k entropy* is

$$H_k(S) = \frac{1}{n} \sum_{A \in \Delta^k} |S_A| H_0(S_A). \tag{A.2}$$

This H_k is a lower bound of a codification which depends of the k preceding symbols [Manzini01].

A.2.2 Bitmaps

A *bitmap* B of length n is an array of n elements with values in $\{0, 1\}$. The function $\mathsf{Rank}_i(B, j)$ is the number of elements in the bitmap B from position 1 to j which are equal to i, where $i \in \{0, 1\}$. $\mathsf{Select}_i(B, j)$ is the smallest position k of the bitmap B such that $\mathsf{Rank}_i(B, k) = j$ with $i \in \{0, 1\}$. The function $\mathsf{Access}(B, j)$ returns the element of B at position j.

Some properties of the Access, Rank and Select functions are

1. $\mathsf{Rank}_i(B, j) = j - \mathsf{Rank}_{1-i}(B, j)$
2. $\mathsf{Access}(B, j) = \mathsf{Rank}_1(B, j) - \mathsf{Rank}_1(B, j-1)$
3. $\mathsf{Rank}_i(B, \mathsf{Select}_i(B, j)) = j$
4. $\mathsf{Select}_i(B, \mathsf{Rank}_i(B, j)) \leq j$

The functions Rank_i, Select_i, and Access can be computed in constant time using an extra space of $o(n)$ bits for a bitmap B of length n [Raman02]. To do this, B is splitted into superblocks of size $\log^2 n$ bits. Now, each superblock is divided in blocks of size $(\log n)/2$ bits.

Each block is divided in classes. A block is on class c if it contains c bits set. For each class, there is a universal table with one row for every possible block on the class. On this table we store all the Rank answers for that block. All the tables need $\sqrt{n}$ rows and $\mathbf{O}(\sqrt{n}\mathbf{polylog}(n))$ bits. Each superblock stores the value of Rank at its beginning. Each block stores the value of Rank at the beginning of the block relative to value of the superblock where the block is.

To solve $\mathsf{Rank}(B, i)$ we find the superblock where the position i is and take its Rank value, then we find the block where i is and add its value to the one of the superblock. Finally, we find the number of ones on the block, from the beginning to i, using the universal table. We add up these three numbers and get $\mathsf{Rank}(B, i)$ in constant time.

The function Select can be computed using binary search and Rank but could be done in constant time using an analogue technique of the Rank function.

This method is only for static bitmaps. There are some theoretical models to compute Rank, Select, and Access for dynamic bitmaps [Mäkinen08] but there are no useful implementations.

A.2.3 Sequence Indexes

Now the previous operations are extended over larger alphabets. Let S be a sequence of symbols of size N (i.e., $S \in \Delta^N$ for some alphabet Δ, of size $\sigma = |\Delta|$). There exist efficient representations of sequences closely following the entropy lower bound and providing a rich set of efficient operations; they are called Sequence Indexes. A Sequence Index provides three basic operations over a sequence S.

- $\mathsf{Rank}_c(S, pos)$ counts how many c's occur in S until position pos.
- $\mathsf{Select}_c(S, r)$ returns the smaller position pos such that $\mathsf{Rank}_c(S, pos) = r$.
- $\mathsf{Access}(S, pos)$ retrieves the symbol stored at the position pos in S.

The interested reader is referred to [Navarro07, Tellez12, Golynski06, Claude08].

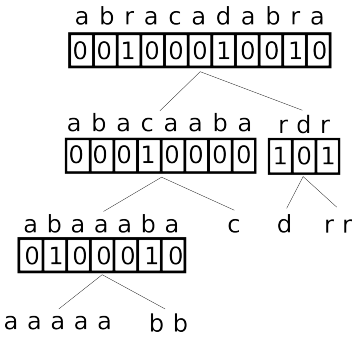


Figure A.1: Wavelet Tree for the string abracadabra

A.2.4 Wavelet Trees

Let S be a string over an alphabet of length σ. A binary *wavelet tree* for S is a balanced binary tree whose leaves are the symbols of the alphabet. The root is S. The left child of a node ν is formed by the elements of ν that are on the first half of the alphabet of ν. The right child is formed by the rest of the elements of ν. The sequences at each node are represented by bitmaps of length equal to the string at the node. A zero at position i means that the i^{th} character of the node is on the first half of its alphabet. A one means the opposite.

For example, on Figure A.1 we have a wavelet tree for the string abracadabra. On the root of the tree we have the whole string and a bitmap, a 0 means that the corresponding symbol is in the first half of the alphabet ($\{a, b, c\}$). A 1 means that the character is one of the set $\{r, d\}$. Then, all the 0's goes to the left to form the string abacaaba and the 1's form rdr. This process is repeated until the generated strings are formed by one symbol.

With Rank and Select operations over the bitmaps on a wavelet tree, we can generate the original string S on the root. Even more, we can compute the functions $\mathsf{Rank}_c(S, i)$ that returns the number of occurrences of character c on the first i elements of S; and the function $\mathsf{Select}_c(S, j)$ that returns the position on S where is the j^{th} occurrence of character c. This operations can be done using $\mathbf{O}(\log \sigma)$ operations on bitmaps.

A.2.5 Permutations

Permutations are fundamental in computer science and have been extensively studied. A *permutation* is a bijection of a set of n elements with itself. Given a set of n elements, the identity

permutation is $I(x) = x$ for all x. Given a permutation P, its inverse P^{-1} is the unique permutation such that $P \circ P^{-1} = I$. The objective is to compute a permutation and its inverse using a small amount of space and time. Next is shown how to represent a permutation of n numbers using $(1+\epsilon)n \log n + o(n)$ bits of space that computes the permutation in $O(1)$ time and its inverse in $O(1/\epsilon)$ time for any $\epsilon > 0$ [Munro03].

Let P be a permutation of $S = \{0, 1, \ldots, n-1\}$ and $t \geq 2$ be a parameter. A *cycle* C of P is a subset of S such that $P(x) \in C$ for every $x \in C$. Every permutation can be written as the product of disjoint cycles. For each cycle of P with length at least t, we store a shortcut pointer for the elements of the cycle that are at positions which are multiples of t. The shortcut pointer points to the position that is t elements before it in the cycle.

If we want to know $P^{-1}(x)$, we check whether x has a pointer or not. If it has one, we follow the pointer and then the cycle until we go back to x, the element before x is $P^{-1}(x)$. If x does not have a pointer, we follow the cycle until we find one, we follow it and then the cycle until we find x. The previous element is $P^{-1}(x)$.

Taking $\epsilon = t/2$, we get the time and space mentioned [Munro03].

A.2.6 Elias-δ encoding

Let $A = 0^{g_0}10^{g_1}1 \cdots 10^{g_{l-1}}10^{g_l}$ where 0^{g_i} represents a sequence of g_i zeros. A will be represented as $\delta(g_0)\delta(g_1)\ldots\delta(g_l)$, where $\delta(x)$ is an encoding for the non-negative integer x. Let $l = \lceil \log(x+1) \rceil$ be the number of bits necessary to encode x, and let $ll = \lceil \log(l+1) \rceil$ be the number of bits necessary to code l. Then $\delta(x)$ is formed by three parts: ll zeros followed by a 1; the $ll - 1$ least significant bits of the binary representation of l; and the $l - 1$ least significant bits of the binary representation of x. This compression is knowns as Elias-δ [Elias75].

A.2.7 Full-Text Indexes

The full-text indexes represent a text in a compressed structure and still can answer queries efficiently. A well known example is the *inverted index* which is a list of the words on a database together with their positions in the text. This structure is very popular but has some limitations as it needs the existence of words in the text, the set of different words should not be too large, and we

can just search for full words, not for sub-strings. In order to improve these limitations, the full-text indexes were developed. The problem of these indexes is the large amount of space they need. Later appeared the self-indexes that could reproduce any portion of the text so they could replace the text.

Given a text and a pattern, full-text indexes can count the number of occurrences of the pattern in the text and their positions. If the text is not stored separately, the operation of recovering any substring of it will be needed.

A requirement will be that the text T ends with a special character called \$ that appears only one time on the text and is greater than any other character of text in the lexicographic order.

Some of the basic components of many full-text indexes are described below.

Tries

A *trie* is a tree data structure that stores a set of strings. It can search a string in time proportional to its length. A trie for the strings $S_1, S_2, \ldots, S_n$ is a tree where each node represents a distinct prefix in the set. The root node represents the empty prefix ϵ. A node v representing the prefix Y is a child of a node u representing the prefix X if and only if $Y = Xc$ for some character c, which will be the label of the tree edge from v to u.

The character \$ is added at the end of every prefix. The trie has exactly n leaves. The trie can be built in time proportional to the sum of the lengths of all its strings. We can find all the prefixes in the collection of strings using a trie, simply follow the trie from the root following a string, then scan all the leaves of that subtree to find all the strings that start with the given string.

Suffix Tries and Suffix Trees

Given a text $T_{1,n}$ (ending with \$), T defines n suffixes $T_{1,n}, T_{2,n}, \ldots, T_{n,n}$. The *suffix trie* of T is a trie build over all the suffixes of T.

The suffix trie makes fast string matching posible. Given a pattern P, every occurrence of P in T is a substring of T, that is, the prefix of a sufix of T. This permits counting the occurrences of P in T in time proportional to the length of P. To find all the occurrences, the subtree is traversed.

The *suffix tree* of a text T is a suffix trie where each unary path is converted into a single edge. Those edges are labelled by strings obtained by concatenating the characters of the replaced path. The leaves indicate the text position where the suffixes start.

The suffix array

A suffix array A of the text T is an array of pointers to all the suffixes of T in lexicographic order [Manber90]. $T_{i,j}$ will denote the substring of T starting and ending in i and j respectively. T_i will denote the i^{th} character of T. $A[i]$ points to the text suffix $T_{A[i],n}$.

The suffix array plus the text can be used to search for patterns. Just like the result of a pattern search in a suffix tree is a subtree, the result of a suffix array search is an interval. It is possible to search for the interval of A containing the suffixes prefixed by the pattern via two binary searches on A.

The Burrows-Wheeler Transform

Given a text $T_{1,n}$ and its suffix array A, the *Burrows-Wheeler Transformation* (BWT) T^{bwt} of T has size n and is defined as $T^{bwt}_i = T_{A[i]-1}$ except when $A[i] = 1$, where $T^{bwt}_1 = T_n$.

Another way of seeing the BWT is as follows. A *cyclic shift* of $T_{1,n}$ is any string of the form $T_{i,n}T_{1,i-1}$. Let M be a matrix containing all the cyclic shifts of T in lexicographical order. Let F be the first and L be the last column of M. M is the suffix array A of T. L is the BWT.

Given strings F and L resulting from the BWT of T, the *LF-mapping* is a function $LF\colon [1,n] \rightarrow [1,n]$ such that $LF(i)$ is the position in F where character L_i occurs.

Let $C(c)$ be the number of occurrences in T of characters alphabetically smaller than c, and $Occ(c,i)$ be the number of occurrences of character c in $L_{1,i}$. Then $LF(i) = C(L_i) + Occ(L_i, i)$.

The BWT by itself does not compress T, is is just a permutation of its characters. However, this permutation is more compressible than the original T.

A.3 Similar systems

BANKS [Bhalotia02] is a system that represents a relational database with graphs where a node is a tuple and the edges are foreign-key–primary-key links. BANKS enables keyword search over relational databases. The user does not need to know the scheme to do the queries. The system uses an inverted index to find the nodes where the given keywords are, and then uses graph algorithms to find the more relevant records. BANKS creates the joins automatically.

Kite is presented on [Sayyadian07],it is a system that enables keyword searches over multiples databases without the need to know the databases scheme. Kite makes the joins over the databases automatically. It uses inverted indexes to find the location of the keywords, then it ranks the records to find the most relevant ones and finally, uses the DBMS to display the results. The best part of this system is that it can be used to tens of databases and the queries are independent of the schemes of the databases.

The authors of [Stockinger09] take a DBMS and attach an inverted index implemented using compressed bitmaps. They called this structure FastBit. To compress the bitmaps they use the WAH encoding [Wu06]. The inverted index is built over the text fields of the database using Lucine. Here, the results show that a DBMS with a FastBit is like 50 times faster than using MySQL alone.

DBXplorer [S.02] is a system that is used with a DBMS to enable keyword searches. It uses an inverted index to find the records of the database that contain the keywords in the query. The system permits the search over a database and the user does not need to know the database scheme. DBXplorer uses the inverted index to find the records and an SQL query for showing them. The changes on the data are saved in a table of the database and a server updates the inverted index with the new modifications. The update is done offline and the users do not see it.

On [Hristidis03], the authors made a model to incorporate to a DBMS keyword searches over a database, even on records that have information on different tables. The results of the queries are the k most relevant records. The model does searches using AND and OR operators over the keywords and the scheme of the database is not needed to do the queries. This system represents the records of the database as nodes of a graph and the edges are the relations between the tables. Analysing these graphs, the system finds the relations of the data and performs the joins with the most relevant answers. The keywords can appear on any field of a record to be considered as a result.

A.4 Representing Tables as Sequences

The main idea proposed in this appendix is to represent a table in a succinct way while supporting efficient searches on it. To achieve these goals, a new representation of a table is detailed in what follows. Let $\Delta = \{1, 2, \ldots, \sigma\}$ be an ordered set called the alphabet. Let the table $\mathcal{T}$ be

the set of records $\{r_1, r_2, \ldots, r_n\}$, where each record $r_i \in C_1 \times C_2 \times \cdots \times C_m$. Here C_j is the set of valid entries in column j, and $C_j \subset \Delta^+$ (strings of symbols in Δ with length greater than zero). The null entries should be marked with a special symbol. Thus, for instance the table can be seen as the $n \times m$ array below

$$\begin{aligned} \mathcal{T} \;=\; & s_{1,1}, s_{1,2}, \ldots, s_{1,m} \\ & s_{2,1}, s_{2,2}, \ldots, s_{2,m} \\ & \vdots \\ & s_{n,1}, s_{n,2}, \ldots, s_{n,m}, \end{aligned}$$

where each $s_{i,j}$ (called the i,j cell) corresponds to the j-th column of the i-th record.

The content of the tables must be possible to be separated in text entities $w_i \in \Delta^+$, e.g. words and word-separators. For simplicity, the term *word* will be used to name any text entity. Let $V(S)$ be the set of all different words on a string S. $V(S)$ will be called the *vocabulary* of S.

In this work, the following operations are presented:

- Search for all occurrences of a given string SubstringMatch $(\mathcal{T}, j, p)$.
  Given a string p and a column j, it retrieves the set of records $\{r_i \in \mathcal{T} \mid p \text{ is a substring of } s_{i,j}\}$
- Cell matching CellMatch$(\mathcal{T}, j, p)$. Given a string p and a column j, it retrieves the set of records $\{r_i \in \mathcal{T} \mid p = s_{i,j}\}$.
- Join query Join$(\mathcal{A}, \mathcal{B}, x, y)$. Given two columns x and y, on tables $\mathcal{A}$ and $\mathcal{B}$ respectively, it retrieves the set $\{(r_i, r_j) \in \mathcal{A} \times \mathcal{B} \mid s_{i,x} = s_{j,y}\}$.

The first attempt, that will be refined later, is to represent a table as the concatenation of the cells as a single string, with some special delimiters. Abusing the notation a bit, the table $\mathcal{T}$ is a string concatenating all the cells. We can write $\mathcal{T} = \rho_1\rho_2\cdots\rho_n$, where ρ_i is the concatenation of $\#s_{i,1}\#s_{i,2}\#\cdots\#s_{i,m}$ and $\#$ is not in the vocabulary. Every cell in a table will be a string in $V(\mathcal{T})$, and the table will be a string in $V(\mathcal{T}) \cup \{\#\}$. If we were doing this, the operation SubstringMatch would be complicated because this representation needs extra effort to distinguish the columns.

A column-wise representation is better fitted for this purposes. $\mathcal{T}$ will be represented as m sequences, one per column, as follows. If we see $\mathcal{T}$ as an $n \times m$ array, the concatenation of the entries of the column j will be the string $T_j = \#s_{1,j}\#s_{2,j}\#\cdots\#s_{m,j}\#$. A Sequence Index on each T_j is used for $1 \leq j \leq m$. Notice that under this approach, T_j is seen as a text over the vocabulary $V(T_j) \subseteq V(\mathcal{T}) \cup \{\#\}$.

The focus will be on both searching procedures and extracting strings. The operations will be divided on the search step and the record recovery step. For a given query, the occurrences set is computed on the *search* step; the *location* step takes the *search* results and retrieves the records identifiers (i.e, where the record is located); and finally, if required, the relevant records can be shown in the *extract* step.

A.5 Algorithms

The *search* step, described above, is the most convoluted part of the solution. Once the results of a query are computed in the column-wise representation, then counting the number of results is easy. On the other side, *extract* consists of Access calls on the index of sequences. The general strategy consists on transforming a relational query subgoals into set operations, mostly union and intersection operations as follows. Let T be a string and let $L_{T,c,sp}$ be the set (represented by a sorted list) of all positions on T where character c occurs shifted to the left in sp positions, i.e., $L_{T,c,sp}[i] = \mathsf{Select}_c(T, i) - sp$ defined for all $1 \leq i \leq \mathsf{Rank}_c(T, |T|)$ (Algorithm 13).

Algorithm 13 Computing $L_{T,c,sp}$.

Input: A string T representing a column of a table, a word c, and a number sp.
Output: The ordered positions on T where c occurs shifted to the left sp positions.

1: Let L be empty.
2: **for** $i = 1$ to $\mathsf{Rank}_c(T, |T|)$ **do**
3: Add $\mathsf{Select}_c(T, i) - sp$ to L.
4: **end for**
5: **return** L

Consider the *substring matching problem* $\mathsf{SubstringMatch}(T, j, p)$. Let T be the sequence representing the j-th column of $\mathcal{T}$, and let the string p be a sequence of words, $p = k_1 k_2 \cdots k_\ell$. First, we need to determine if all $k_i \in V(T)$; in the negative case, p has no occur-

rences on T; on the positive case, we retrieve the corresponding sorted lists of the pattern:

$$P_T = \{L_{T,k_1,0}, L_{T,k_2,1}, \ldots, L_{T,k_\ell,\ell-1}\}.$$

Thus, the set $\mathcal{C} = \bigcap_{L \in P_T} L$, contains all occurrences of p on T. Then, we get the set $\{\mathsf{Rank}_{\#}(T, pos) \mid pos \in \mathcal{C}\}$ to find wanted records. This last operation is called *location*. The substring matching problem is presented in Algorithm 14.

The intersection operation (Line 11) can be implemented using an ad-hoc intersection algorithm working with sorted lists [Demaine01, Baeza-Yates10, Barbay09]. In particular, the Baeza-Yates algorithm was used using a small vs. small (SvS) pairwise set intersection strategy [Barbay09]. The reason to choose this algorithm is that, in the tested datasets, a Zipf distribution was expected (i.e. there should be a very small list on P_T). The searching time is in the worst case $m \log \binom{\min_{L \in P_T} |L| + \max_{L \in P_T} |L|}{\min_{L \in P_T} |L|} \cdot t_s + occs \cdot t_r$, where $occs$ is the number of occurrences in the result set; and t_s, and t_r are the time complexities for Select and Rank operations on T.

Algorithm 14 The search algorithm for SubstringMatch

Input: A string T representing a column of a table and the pattern p.
Output: The record number containing pattern p on the column T.

```
1: Let R be empty.
2: Divide p in words k_1, k_2, ..., k_ℓ.
3: for i = 1 to ℓ do
4:     if k_i is not in the vocabulary of T then
5:         return p not found
6:     end if.
7: end for
8: for i = 1 to ℓ do
9:     Compute L_{T,k_i,i-1}
10: end for
11: Compute L_{T,k_1,0} ∩ L_{T,k_2,1} ∩ ··· ∩ L_{T,k_ℓ,ℓ-1} and store the result on C.
12: for all pos ∈ C do
13:     Add Rank_#(T, pos) to R
14: end for
15: return R
```

It is possible to state the *cell matching problem* $\mathsf{CellMatch}(T, j, p)$ in terms of substring matching, simply modifying the search pattern $p' = \#p\#$ and solving with SubstringMatch. Despite the use of an extended vocabulary to compose the query, this is a transparent modification to the Sequence Index machinery. The same goal can be achieved simply by locating p and

then checking all occurrence positions pos to be preceded and followed for $\#$, by verifying that $\mathsf{Access}(T, pos-1) = \mathsf{Access}(T, pos+m+1) = \#$. This can be specially useful whenever the performance of an intersection algorithm becomes degraded with the existence of large sets.

Finally, the *join query problem* $\mathsf{Join}(\mathcal{A}, \mathcal{B}, i, j)$ is studied. Here, two cases are considered:

(a) The content of each cell is a single word.

(b) The content is a string of symbols larger than 1.

Consider case (a), let A and B be the sequences of the i-th and j-th columns, respectively for tables $\mathcal{A}$ and $\mathcal{B}$. Let $V(A)$ and $V(B)$ be their vocabularies. The first step consists on computing $V_\cap = V(A) \cap V(B)$, then taking the union J of the Cartesian products $\mathsf{CellMatch}(\mathcal{A}, i, w) \times \mathsf{CellMatch}(\mathcal{B}, j, w)$ for each word $w \in V_\cap$. The set J is the searched join. At least $\log \binom{|V(A)|+|V(B)|}{|V(B)|}$ symbol comparisons were required for the intersection of vocabularies plus the necessary Select operations to solve the $\mathsf{CellMatch}$ operation (which depends on the cardinality of the corresponding sorted lists). This simple Join solution has a powerful extension: it is possible to obtain a fast filter for the Jaccard similarity [Xiao11] without the need of a modification. However, this extension is beyond the scope of the work.

In case (b), Join is reduced to search all cells of the i-th column in $\mathcal{A}$ in the j-th column in $\mathcal{B}$ using $\mathsf{CellMatch}$.

Finally, the *extraction* step is performed as follows. Let $\mathcal{T}$ be represented by the set of sequences $\{T_i\}$, the extraction of a single cell on T_i corresponding to record j is reconstructed repeatedly using $\mathsf{Access}(T_i, x)$ for x in the range from $\mathsf{Select}_{\#}(T_i, j)+1$ to $\mathsf{Select}_{\#}(T_i, j+1)-1$, with inclusive bounds.

A.6 Experimental Results

The algorithms were tested against SQlite3 using the extention FTS [1] which allows the user to query the database for all rows that contain one or more words. Even if the presented method and SQlite3 have different objectives, the focus was on compression and algorithms working on the compressed representation while SQlite3 is an ACID-compliant embedded DBMS. SQlite3

[1] `http://www.sqlite.org/fts3.html`

was selected due to its wide usage in the community and its full text search capabilities through FTS. Also, SQlite3+FTS provides transparent access to the working databases, and the resource management can me measured easily.

The Sequence Index XLB was used with DiffSet and SArray bitmaps [Tellez12]. The XLB-DiffSet can achieve local H_0 entropy while XLB-SArray is a good worst case structure. Both XLB indexes have efficient Select operations. Parameters B and t control the performance trading speed with memory. Briefly, XLB-DiffSet computes all Select answers storing the differences among values. Also, an absolute value and an offset position to the corresponding difference is placed every B differences. Using this scheme it is possible to offer worst case guarantees. Access is solved using the underlying inverse permutation stored by the sequence index. This structure consists of at most $|T|/t$ pointers for the text T. Access requires at most t Select calls to be solved. The parameters are $t = 1, 6, and 12$, and $B = 15, 31, 63, and 127$. A wavelet tree (WT) [Mäkinen08] was also used with the GGMN [González05] uncompressed bitmap index, this combination requires $O(\log V(T))$ time for all operations. In addition, the structures were tested with the Sequence Index of Golynski et al. [Golynski06], implemented as described by Claude and Navarro [Claude08]. The last structure needs $O(1)$ time for Select and solves Rank in $\log V(T)$ time. Access is solved in $O(t)$ time (with the same values of t than XLB).

A.6.1 Description of the Databases

As stated before, the primary interest is on those databases containing text information and the SubstringMatch operation is of interest. The vocabulary was made separating cells into words. They follow the Law of Zipf, which says that the most common word is twice as frequent as the second most common word, three times more frequent as the third most common word and so on. However, this is not the general case since some columns may contain unique identifiers (like those dedicated to primary keys).

The compact tables were tested on the database Book from the Freebase site[2]. Book is composed of the following fields: `name`, `id`, `editions`, `characters`, `genre`, `first_edition` and `interior_illustrations_by`. It contains more than 2.4 million records and 3 million text entities. The raw text file of the Book table is 250M long.

[2] `http://www.freebase.com`

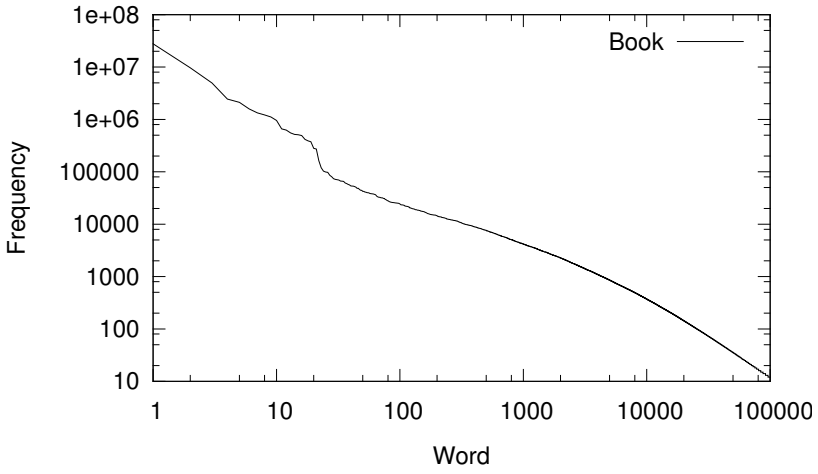


Figure A.2: Frequency of the Book vocabulary

Figure A.2 shows the frequency of the 100000 most common words. Notice that the 30 most common words have a big portion of the database, clearly following a Zipf tendency. 230 phrases were selected (substrings) as queries from table Book in the column `genre`; 104 composed by one single word, 105 having 3 words, and 21 with 5 words or more; the operation SubstringMatch was applied. All experimentations were executed in a 16 core Intel Xeon 2.40 GHz workstation with 32GiB of RAM, running CentOS Linux, without exploiting the parallel capabilities. Independent tests were made and show that the same algorithms running on disk are just 3 times slower than running on main memory. This impressive performance is due to memory cache of the OS, and the high amount of available RAM.

A.6.2 Performance of the Compact Tables

As anticipated in Section A.5, the cost of the searches is a function of the underlying set operations (mostly union and intersection) and the Select, Rank, and Access time costs.

In order to get a better understanding of the costs, the procedure was splitted in three steps: search, locate, and extract. Figure A.3 shows the time of SubstringMatch to *count* the results on the y-axis and the query number in the x-axis. The queries are ordered by the time on XLB-DiffSet(12,127) XLB-SArray has the best performance; also the majority of compact representations are faster than SQlite3. On the other hand, expensive queries for the presented model are not necessarily expensive for SQlite3. If both indexes were run in parallel, we could always give the

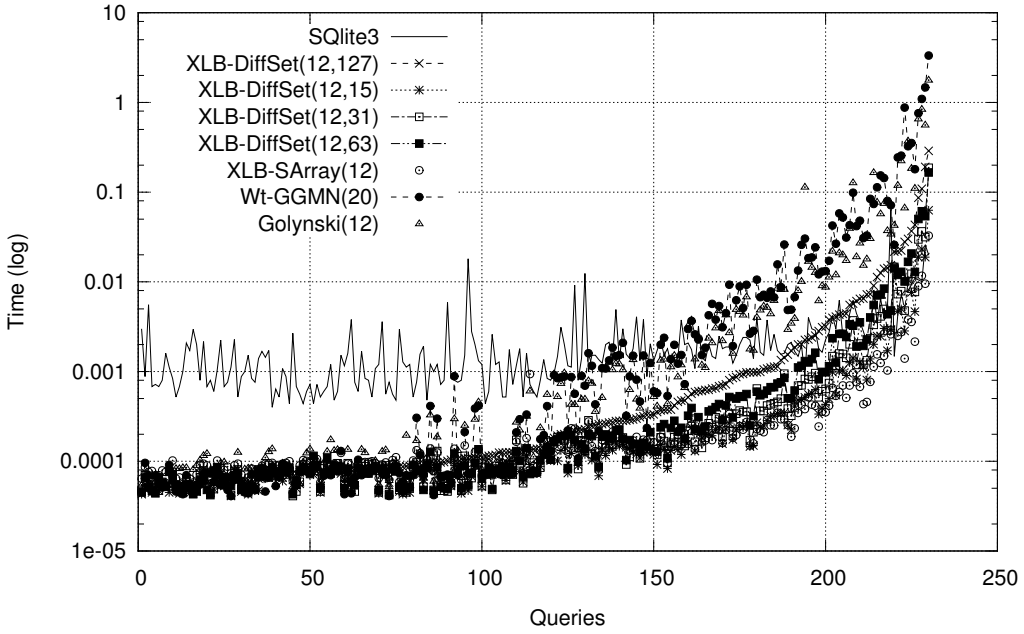


Figure A.3: *Search* time. Query time for queries of different complexity.

faster answer. In this figure we can see the impact that the query complexity has over the time on the model. The fastest queries corresponds to single word queries and as the time becomes greater, also does the number of words in the searched strings.

Figure A.4 shows the time per occurrence. We can see that *locate* is a quite efficient operation, independent of the parameter t; t was fixed to 6 on the image to show its performance. On the other side, Figure A.5(a) shows the average time per character of the *extract* operation, we can see it is fast and highly sensitive to the available memory (specifically on the parameter t).

Figure A.6(a) shows the execution time sorted by query complexity (XLB-DiffSet $B = 127$ was selected for this task). Again, it is important to notice that the cost of searches depends critically on the query itself; this is because the algorithms are based on set operations. All compact representations have a similar performance, in contrast SQlite3, not necessarily has the same unit of complexity. In this experiment, the parameter t was fixed to 12. Comparing Figure A.3 and A.6(a) we see a big performance difference, two orders of magnitude, meaning that the searching operation is quite fast. The total time also contains the *locate* and *extract* operations, being the *extract* time dominating; the domination arises because *locate* is using a single Rank operation per occurrence

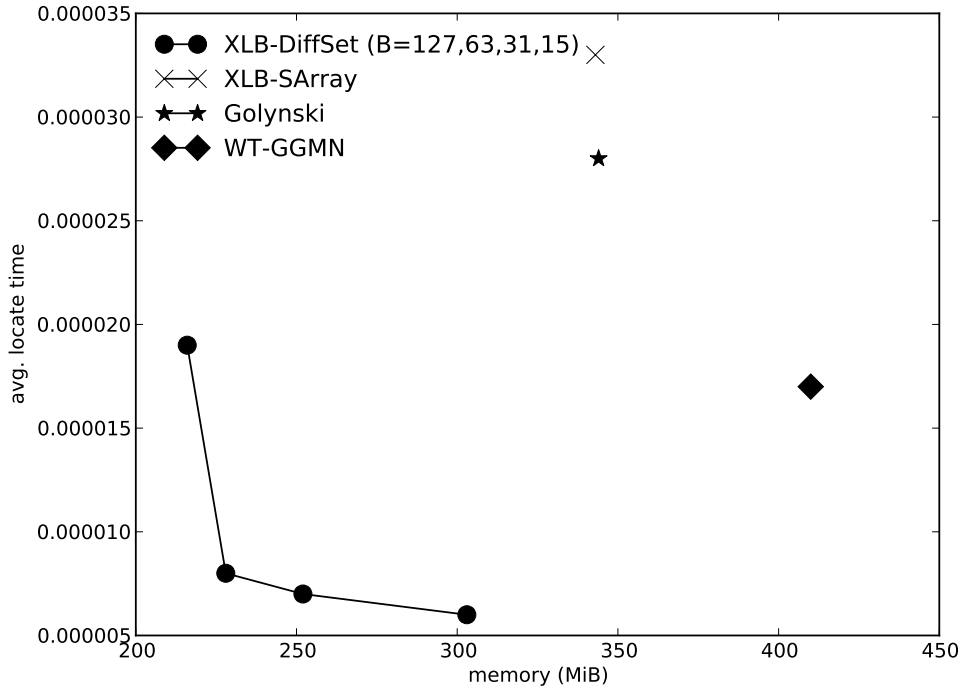


Figure A.4: *Locate* time. Time per occurrence.

while *extract* retrieves the entire set of records containing the occurrences. Since the performance is output-sensitive, the time was divided by the size of the result.

Even having a compressed representation of the database, and being able to efficiently solve both *search* and *locate* operations, the *extract* operation is much slower. To improve the *extraction* time using a small memory footprint, the particulars of the database were used.

Caching Symbols to Take Advantage of the Zipf Distribution

The *extract* procedure can be quite expensive if the outcome is large (many records and/or large ones) even considering that the "per character" operation is quite fast. This can be solved by caching, as described below.

The underlying reason of the slow performance is the compact nature of the representation. An interesting option to accelerate *extract* and to kept excellent searching times is using a fast searching Sequence Index and maintaining the entire text uncompressed giving the fastest Access operation, i.e., $O(1)$. Since it is important to avoid memory usage, we will do the next best thing instead: take advantage of the distribution of the tested datasets to improve the *extract* times, while maintain a moderate memory requirements. Please notice that since the Zipf distribution is expected, with high probability, any random word to be extracted is in the set of $K = O(1)$ most frequent terms in the frequency table. In order to enhance the extraction time, all occurrences of the K most frequent symbols on each sequence were cached, that is, for all words c in T at position i, if c is a common word, the symbol was stored on a new array $T_C[i]$, otherwise if c is not common, then the special symbol ε is stored in $T_C[i]$. Then to solve the $\mathsf{Access}(T, pos)$ operation, first lookup T_C. If $T_C[pos] = \varepsilon$ then solve it as the non cached $\mathsf{Access}(T, pos)$ operation, if $T_C[pos] \neq \varepsilon$ then solve the operation returning $T_C[pos]$. This caching scheme adds $|T_C| \log(K + 1)$ bits to the table representation. For the experiments, $K = 15$ was fixed to be able to use 4 bits per symbol in the cache array.

Figure A.5(b) shows the average extraction time per character using the distribution aware caching approach. For each curve, each point represents a different value of the parameter $t = 1, 6, 12$, appearing in this order from right to left in figures. As anticipated, the parameter t can be used to trade time and memory, along the B parameter of XLB-DiffSet. Also, we can see the final improvement achieved using this caching scheme in Figure A.6(b). As we can see, the majority of

the caching methods are faster than non caching ones, Figure A.6(b) and A.6(a), respectively.

A.7 Summary

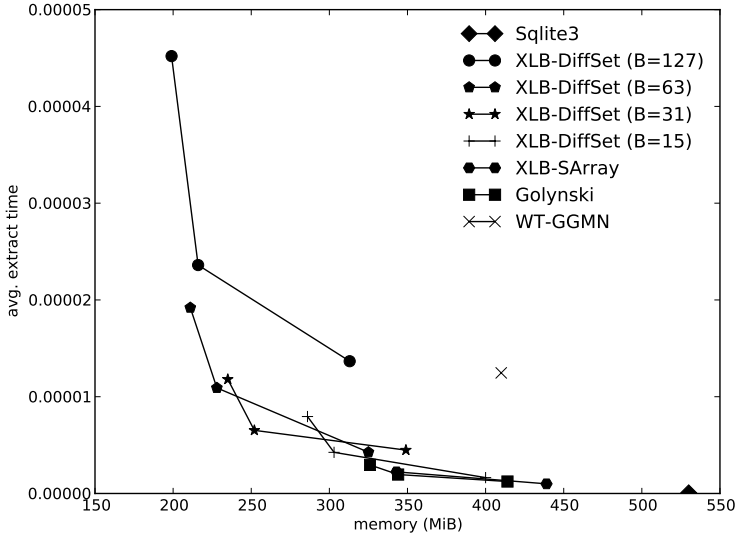
It was presented a model that can compress a database and perform both keyword and phrase queries without the need of decompression. The presented model accepts any Sequence Index, and can be configured to either occupy less space or answer the queries faster. Compared to SQlite3 using FTS, the proposed approach needs less space while improving the searching and locating time.

With some additions, the model could answer numeric queries, similarity joins, and others. Also, it is based on Sequence Indexes, but it can be easily extended to other compact and compressed structures like full text self indexes. They can be used to reach text from databases not containing *natural languages* (e.g., oriental languages, agglutinant languages, DNA, etc.) which can open a new set of interesting applications.

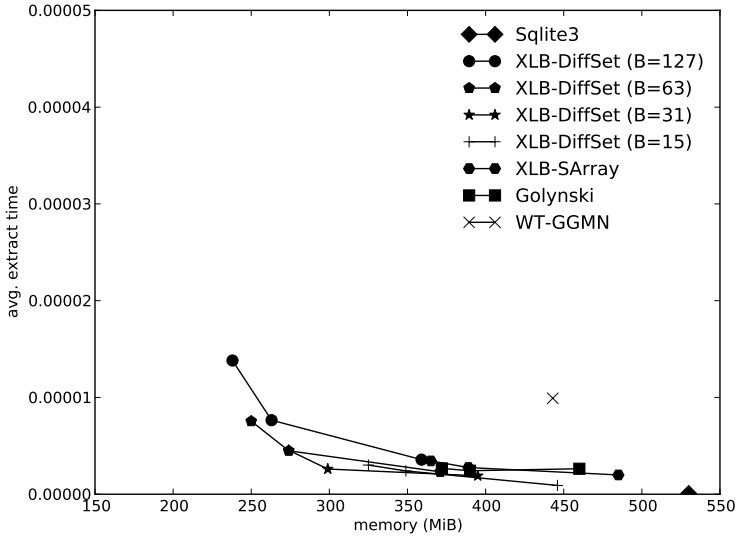
Since the approach is based on Sequence Indexes, it share the limitations they have, namely:

- Most Sequence Indexes are static, however there exists a few theoretical proposals (e.g. [Navarro13]) handling dynamicity.
- Sequence Indexes are designed to work on main memory, there are not good implementations for disk.

Finally, notice that the performance of *extract* needs to be improved, since for small compression ratios this operation becomes a bottleneck on applications with large result sizes. However, applications performing complex searches with small number of records in the outcome can take advantage of this technique.

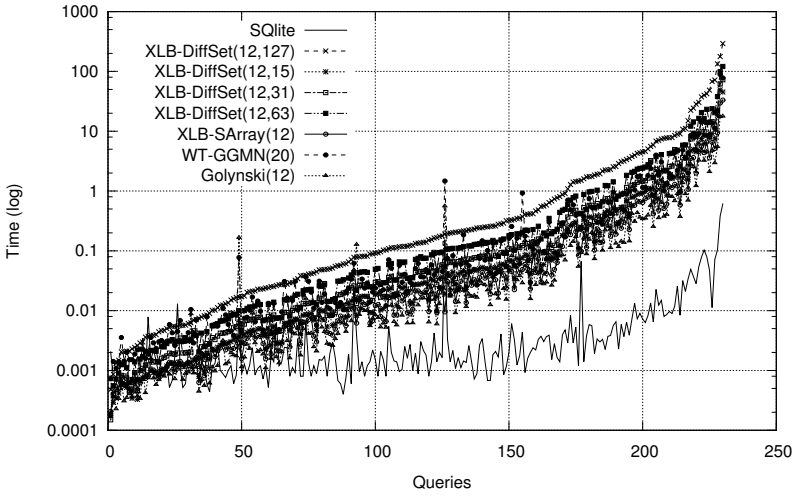


(a) ***The representation without cache.***

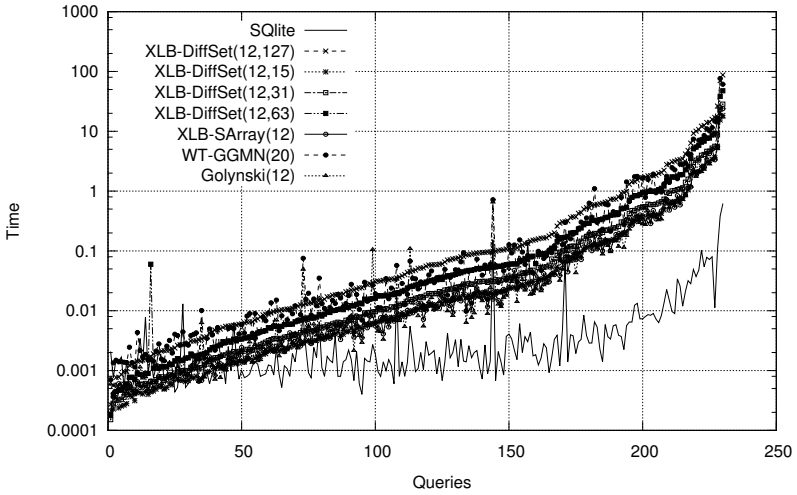


(b) ***Using the Zipf caching scheme.***

Figure A.5: Average time to *extract* a single character



(a) ***Total time without cache.***



(b) ***Total time with the Zipfian caching strategy.***

Figure A.6: The necessary time to search, locate, and extract the results of a set of queries. The size of the text describing the table is of 250M.

References

[Amato08] Amato, G. y Savino, P. Approximate similarity search in metric spaces using inverted files. *En InfoScale '08: Proceedings of the 3rd international conference on Scalable information systems*, págs. 1–10. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), ICST, Brussels, Belgium, Belgium, 2008. ISBN 978-963-9799-28-8.

[Amato14] Amato, G., Gennaro, C., y Savino, P. Mi-file: using inverted files for scalable approximate similarity search. *Multimedia Tools and Applications*, 71(3):1333–1362, 2014. ISSN 1380-7501. doi:10.1007/s11042-012-1271-1.

[Andoni08a] Andoni, A. y Indyk, P. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Communications ACM*, 51:117–122, January 2008. ISSN 0001-0782. doi:http://doi.acm.org/10.1145/1327452.1327494.
URL `http://doi.acm.org/10.1145/1327452.1327494`

[Andoni08b] Andoni, A. y Indyk, P. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Communications of the ACM*, 51:117–122, January 2008. ISSN 0001-0782. doi:http://doi.acm.org/10.1145/1327452.1327494.

[Arya93] Arya, S. y Mount, D. M. Approximate nearest neighbor queries in fixed dimensions. *En SODA*, tomo 93, págs. 271–280. 1993.

[Arya98] Arya, S., Mount, D., Netanyahu, N., Silverman, R., y Wu, Y. An optimal algorithm for approximate nearest neighbor searching in fixed dimensions. *Journal of the ACM*, 45(6):891–923, 1998.

[Baeza-Yates99] Baeza-Yates, R. A. y Ribeiro-Neto, B. A. *Modern Information Retrieval*. ACM Press / Addison-Wesley, 1999. ISBN 0-201-39829-X.
URL `citeseer.ist.psu.edu/baeza-yates99modern.html`

[Baeza-Yates10] Baeza-Yates, R. A. y Salinger, A. Fast intersection algorithms for sorted sequences. *En Algorithms and Applications*, págs. 45–61. 2010.

[Barbay09] Barbay, J., López-Ortiz, A., Lu, T., y Salinger, A. An experimental investigation of set intersection algorithms for text searching. *J. Exp. Algorithmics*, 14:3.7–3.24, 2009. ISSN 1084-6654. doi:http://doi.acm.org/10.1145/1498698.1564507.

[Bentley80] Bentley, J. L. y Saxe, J. B. Decomposable searching problems i. static-to-dynamic transformation. *Journal of Algorithms*, 1(4):301–358, 1980.

[Beygelzimer06] Beygelzimer, A., Kakade, S., y Langford, J. Cover trees for nearest neighbor. *En Proceedings of the 23rd international conference on Machine learning*, págs. 97–104. ACM, 2006.

[Bhalotia02] Bhalotia, G., Hulgeri, A., Nakhe, C., Chakrabarti, S., y Sudarshan, S. Keyword searching and browsing in databases using banks. *En ICDE*, págs. 431–440. 2002.

[Böhm01] Böhm, C., Berchtold, S., y Keim, D. A. Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. *ACM Computing Surveys*, 33(3):322–373, 2001. ISSN 0360-0300. doi:http://doi.acm.org/10.1145/502807.502809.

[Bolettieri09] Bolettieri, P., Esuli, A., Falchi, F., Lucchese, C., Perego, R., Piccioli, T., y Rabitti, F. CoPhIR: a test collection for content-based image retrieval.

CoRR, abs/0905.4627v2, 2009.
URL `http://cophir.isti.cnr.it`

[Bose04] Bose, P. y Morin, P. Online routing in triangulations. *SIAM journal on computing*, 33(4):937–951, 2004.

[Bustos03] Bustos, B., Navarro, G., y Chávez, E. Pivot selection techniques for proximity searching in metric spaces. *Pattern Recognition Letters*, 24(14):2357–2366, 2003.

[Celik02a] Celik, C. Priority vantage points structures for similarity queries in metric spaces. *En EurAsia-ICT '02: Proceedings of the First EurAsian Conference on Information and Communication Technology*, págs. 256–263. Springer-Verlag, London, UK, 2002. ISBN 3-540-00028-3.

[Celik02b] Celik, C. Priority vantage points structures for similarity queries in metric spaces. *En EurAsia-ICT '02: Proceedings of the First EurAsian Conference on Information and Communication Technology*, págs. 256–263. Springer-Verlag, London, UK, 2002. ISBN 3-540-00028-3.

[Celik08] Celik, C. Effective use of space for pivot-based metric indexing structures. *En SISAP '08: Proceedings of the First International Workshop on Similarity Search and Applications (sisap 2008)*, págs. 113–120. IEEE Computer Society, Washington, DC, USA, 2008. ISBN 978-0-7695-3101-4. doi: http://dx.doi.org/10.1109/SISAP.2008.22.

[Chavez01] Chavez, E., Navarro, G., Baeza-Yates, R., y Marroquin, J. L. Searching in metric spaces. *ACM Computing Surveys*, 33(3):273–321, 2001. ISSN 0360-0300. doi:http://doi.acm.org/10.1145/502807.502808.

[Chávez05a] Chávez, E., Dobrev, S., Kranakis, E., Opatrny, J., Stacho, L., Tejeda, H., y Urrutia, J. Half-space proximal: A new local test for extracting a bounded dilation spanner of a unit disk graph. *En* J. H. Anderson, G. Prencipe, y R. Wattenhofer, eds., *OPODIS*, tomo 3974 de *Lecture Notes in Computer*

Science, págs. 235–245. Springer, 2005. ISBN 3-540-36321-1.
URL http://dblp.uni-trier.de/db/conf/opodis/opodis2005.html#ChavezDKOSTU05

[Chávez05b] Chávez, E. y Navarro, G. A compact space decomposition for effective metric indexing. *Pattern Recogn. Lett.*, 26:1363–1376, July 2005. ISSN 0167-8655. doi:http://dx.doi.org/10.1016/j.patrec.2004.11.014.
URL http://dx.doi.org/10.1016/j.patrec.2004.11.014

[Chavez08] Chavez, E., Figueroa, K., y Navarro, G. Effective proximity retrieval by ordering permutations. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 30(9):1647–1658, sep. 2008.

[Chávez14a] Chávez, E., Ludueña, V., Reyes, N., y Roggero, P. Faster proximity searching with the distal sat. *En* A. Traina, J. Traina, Caetano, y R. Cordeiro, eds., *Similarity Search and Applications*, tomo 8821 de *Lecture Notes in Computer Science*, págs. 58–69. Springer International Publishing, 2014. ISBN 978-3-319-11987-8. doi:10.1007/978-3-319-11988-5_6.
URL http://dx.doi.org/10.1007/978-3-319-11988-5_6

[Chavez14b] Chavez, E., Ludueña, V., Reyes, N., y Roggero, P. Faster proximity searching with the Distal SAT. *En Proc. 7th International Conference on Similarity Search and Applications (SISAP)*, LNCS 8821, págs. 58–69. 2014.

[Chávez15] Chávez, E., Graff, M., Navarro, G., y Téllez, E. Near neighbor searching with K nearest references . *Information Systems*, 51(0):43–61, 2015. ISSN 0306-4379. doi:10.1016/j.is.2015.02.001.
URL http://www.sciencedirect.com/science/article/pii/S0306437915000241

[Ciaccia97] Ciaccia, P., Patella, M., y Zezula, P. M-tree: An efficient access method for similarity search in metric spaces. *En Proceedings of the 23rd International Conference on Very Large Data Bases*, VLDB '97, págs. 426–435.

Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997. ISBN 1-55860-470-7.
URL `http://dl.acm.org/citation.cfm?id=645923.671005`

[Clarkson99] Clarkson, K. L. Nearest neighbor queries in metric spaces. *Discrete and Computational Geometry*, 22:63–93, 1999.

[Claude08] Claude, F. y Navarro, G. Practical rank/select queries over arbitrary sequences. *En Proc. 15th SPIRE, LNCS 5280*, págs. 176–187. 2008.

[Cormen01] Cormen, T. H., Leiserson, C., Rivest, R. L., y Stein, C. E. L. C. *Introduction to Algorithms*. McGraw-Hill, Inc., New York, NY, USA, 2ª ed$^{\text{ón}}$., 2001. ISBN 0070131430.

[Demaine01] Demaine, E. D., López-Ortiz, A., y Munro, J. I. Experiments on adaptive set intersections for text retrieval systems. *En ALENEX '01: Revised Papers from the Third International Workshop on Algorithm Engineering and Experimentation*, págs. 91–104. Springer-Verlag, London, UK, 2001. ISBN 3-540-42560-8.

[Edmund K. Burke14] Edmund K. Burke, G. K. *Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques*. Springer US, New York, NY, USA, second edition ed$^{\text{ón}}$., 2014. ISBN 978-1-4614-6940-7. doi:10.1007/978-1-4614-6940-7.

[Elias75] Elias, P. Universal codeword sets and representations of the integers. *Information Theory, IEEE Transactions on*, 21(2):194–203, Mar 1975. ISSN 0018-9448. doi:10.1109/TIT.1975.1055349.

[Esuli09] Esuli, A. Pp-index: Using permutation prefixes for efficient and scalable approximate similarity search. *En Proceedings of the 7th Workshop on Large-Scale Distributed Systems for Information Retrieval (LSDS-IR'09)*, págs. 17–24. Boston, USA, 2009.

[Esuli12] Esuli, A. Use of permutation prefixes for efficient and scalable approximate similarity search. *Information Processing & Management*, 48(5):889–902, 2012. ISSN 0306-4573. doi:http://dx.doi.org/10.1016/j.ipm.2010.11.011.

[Figueroa09] Figueroa, K., Chavez, E., y Navarro, G. Technical description of the metric spaces library. Technical Report. SISAP Project, 2009.

[Gao14] Gao, J., Jagadish, H. V., Lu, W., y Ooi, B. C. DSH: Data sensitive hashing for high-dimensional k-nnsearch. *En Proc. ACM International Conference on Management of Data (SIGMOD)*, págs. 1127–1138. 2014. ISBN 978-1-4503-2376-5. doi:10.1145/2588555.2588565.

[Garber12] Garber, L. Using in-memory analytics to quickly crunch big data. *Computer*, 45:16–18, 2012. ISSN 0018-9162. doi:http://doi.ieeecomputersociety.org/10.1109/MC.2012.358.

[Garcia-Molina09] Garcia-Molina, H., Ullman, J., y Widom, J. *Database Systems: The Complete Book*. Pearson international edition. Pearson Prentice Hall, 2009. ISBN 9780131873254.
URL `https://books.google.com.mx/books?id=pk5GAQAAIAAJ`

[Gionis99] Gionis, A., Indyk, P., y Motwani, R. Similarity search in high dimensions via hashing. *En Proceedings of the 25th International Conference on Very Large Data Bases*, VLDB '99, págs. 518–529. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999. ISBN 1-55860-615-7.
URL `http://portal.acm.org/citation.cfm?id=645925.671516`

[Golynski06] Golynski, A., Munro, J. I., y Rao, S. S. Rank/select operations on large alphabets: a tool for text indexing. *En Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm*, SODA '06, págs. 368–373. ACM, New York, NY, USA, 2006. ISBN 0-89871-605-5. doi:

10.1145/1109557.1109599.
URL http://doi.acm.org/10.1145/1109557.1109599

[González05] González, R., Grabowski, S., Mäkinen, V., y Navarro, G. Practical implementation of rank and select queries. *En Poster Proc. Volume of 4th Workshop on Efficient and Experimental Algorithms (WEA)*, págs. 27–38. CTI Press and Ellinika Grammata, Greece, 2005.

[Goyal08] Goyal, N., Lifshits, Y., y Schütze, H. Disorder inequality: a combinatorial approach to nearest neighbor search. *En Proceedings of the 2008 International Conference on Web Search and Data Mining*, págs. 25–32. ACM, 2008.

[Hjaltason03a] Hjaltason, G. R. y Samet, H. Index-driven similarity search in metric spaces. *ACM Transactions on Database Systems*, 28(4):517–580, 2003. ISSN 0362-5915. doi:http://doi.acm.org/10.1145/958942.958948.

[Hjaltason03b] Hjaltason, G. R. y Samet, H. Index-driven similarity search in metric spaces (survey article). *ACM Transactions on Database Systems (TODS)*, 28(4):517–580, 2003.

[Houle13] Houle, M. E. y Nett, M. Rank cover trees for nearest neighbor search. *En Similarity Search and Applications*, págs. 16–29. Springer, 2013.

[Hristidis03] Hristidis, V., Gravano, L., y Papakonstantinou, Y. Efficient ir-style keyword search over relational databases. *En Proceedings of the 29th international conference on Very large data bases - Volume 29*, VLDB '2003, págs. 850–861. VLDB Endowment, 2003. ISBN 0-12-722442-4.
URL http://dl.acm.org/citation.cfm?id=1315451.1315524

[Indyk04a] Indyk, P. *Handbook of Discrete and Computational Geometry.* CRC press, 2ª ed$^{\underline{ón}}$., 2004. ISBN 1584883014. Chapter 39.

[Indyk04b] Indyk, P. *Handbook of Discrete and Computational Geometry, J.E. Goodman and J. O'Rouke, editors.* CRC press, Boca Raton, FL., 2ª edón., 2004. ISBN 1584883014.

[Jagadish05] Jagadish, H. V., Ooi, B. C., Tan, K.-L., Yu, C., y Zhang, R. idistance: An adaptive b+-tree based indexing method for nearest neighbor search. *ACM Trans. Database Syst.*, 30(2):364–397, jun. 2005. ISSN 0362-5915. doi: 10.1145/1071610.1071612.
URL http://doi.acm.org/10.1145/1071610.1071612

[Jürgens99] Jürgens, M. y Lenz, H.-J. Tree based indexes vs. bitmap indexes: A performance study. *En Proceedings of the Intl. Workshop on Design and Management of Data Warehouses, DMDW'99*, págs. 14–15. 1999.

[Kaplan95] Kaplan, H. Persistent data structures. *En IN HANDBOOK ON DATA STRUCTURES AND APPLICATIONS, CRC PRESS 2001, DINESH MEHTA AND SARTAJ SAHNI (EDITORS) BOROUJERDI, A., AND MORET, B.M.E., PERSISTENCY IN COMPUTATIONAL GEOMETRY; PROC. 7TH CANADIAN CONF. COMP. GEOMETRY, QUEBEC*, págs. 241–246. 1995.

[Lemire12] Lemire, D., Kaser, O., y Gutarra, E. Reordering rows for better compression: Beyond the lexicographic order. *CoRR*, abs/1207.2189, 2012.

[Mäkinen08] Mäkinen, V. y Navarro, G. Dynamic entropy-compressed sequences and full-text indexes. *ACM Trans. Algorithms*, 4:32:1–32:38, July 2008. ISSN 1549-6325. doi:http://doi.acm.org/10.1145/1367064.1367072.
URL http://doi.acm.org/10.1145/1367064.1367072

[Malkov12] Malkov, Y., Ponomarenko, A., Logvinov, A., y Krylov, V. Scalable distributed algorithm for approximate nearest neighbor search problem in high dimensional general metric spaces. *En Proc. 5th International Conference on Similarity Search and Applications (SISAP)*, págs. 132–147. 2012.

[Malkov14] Malkov, Y., Ponomarenko, A., Logvinov, A., y Krylov, V. Approximate nearest neighbor algorithm based on navigable small world graphs. *Information Systems*, 45:61–68, 2014. ISSN 0306-4379. doi:http://dx.doi.org/10.1016/j.is.2013.10.006.

[Manber90] Manber, U. y Myers, G. Suffix arrays: a new method for on-line string searches. *En SODA '90: Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms*, págs. 319–327. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1990. ISBN 0-89871-251-3.

[Manzini01] Manzini, G. An analysis of the burrows-wheeler transform. *Journal of the ACM (JACM)*, 48(3):407–430, 2001.

[Micó94] Micó, M. L., Oncina, J., y Vidal, E. A new version of the nearest-neighbour approximating and eliminating search algorithm (aesa) with linear preprocessing time and memory requirements. *Pattern Recogn. Lett.*, 15:9–17, January 1994. ISSN 0167-8655. doi:10.1016/0167-8655(94)90095-7.
URL `http://portal.acm.org/citation.cfm?id=176626.176628`

[Mohamed13] Mohamed, H. y Marchand-Maillet, S. Quantized ranking for permutation-based indexing. *En* N. Brisaboa, O. Pedreira, y P. Zezula, eds., *Similarity Search and Applications*, tomo 8199 de *Lecture Notes in Computer Science*, págs. 103–114. Springer Berlin Heidelberg, 2013. ISBN 978-3-642-41061-1. doi:10.1007/978-3-642-41062-8_11.
URL `http://dx.doi.org/10.1007/978-3-642-41062-8_11`

[Mohamed14] Mohamed, H. *Scalable approximate k-NN in multidimensional big data. PhD Thesis.* Tesis Doctoral, University of Geneva, 2014.

[Munro03] Munro, J., Raman, R., Raman, V., y Rao, S. Succinct representations of permutations. *En* J. Baeten, J. Lenstra, J. Parrow, y G. Woeginger, eds., *Automata, Languages and Programming*, tomo 2719 de *Lecture Notes in*

Computer Science, págs. 191–191. Springer Berlin / Heidelberg, 2003. ISBN 978-3-540-40493-4. 10.1007/3-540-45061-0_29.
URL http://dx.doi.org/10.1007/3-540-45061-0_29

[Naidan12] Naidan, B. y Hetland, M. L. Static-to-dynamic transformation for metric indexing structures. *En Similarity Search and Applications - 5th International Conference, SISAP 2012, Toronto, ON, Canada, August 9-10, 2012. Proceedings*, págs. 101–115. 2012. doi:10.1007/978-3-642-32153-5_8.
URL http://dx.doi.org/10.1007/978-3-642-32153-5_8

[Navarro02a] Navarro, G. Searching in metric spaces by spatial approximation. *The Very Large Databases Journal (VLDBJ)*, 11(1):28–46, 2002.

[Navarro02b] Navarro, G. Searching in metric spaces by spatial approximation. *The Very Large Databases Journal (VLDBJ)*, 11(1):28–46, 2002.

[Navarro07] Navarro, G. y Mäkinen, V. Compressed full-text indexes. *ACM Comput. Surv.*, 39(1), abr. 2007. ISSN 0360-0300. doi:10.1145/1216370.1216372.
URL http://doi.acm.org/10.1145/1216370.1216372

[Navarro13] Navarro, G. y Nekrich, Y. Optimal dynamic sequence representations. *En Proc. 24th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, págs. 865–876. 2013.

[Nori07] Nori, A. Mobile and embedded databases. *En Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, SIGMOD '07, págs. 1175–1177. ACM, New York, NY, USA, 2007. ISBN 978-1-59593-686-8. doi:http://doi.acm.org/10.1145/1247480.1247648.
URL http://doi.acm.org/10.1145/1247480.1247648

[Novak09] Novak, D. y Batko, M. Metric index: An efficient and scalable solution for similarity search. *En Similarity Search and Applications, 2009. SISAP '09. Second International Workshop on*, págs. 65–73. Aug 2009. doi:10.1109/SISAP.2009.26.

[Patella09] Patella, M. y Ciaccia, P. Approximate similarity search: A multi-faceted problem. *Journal of Discrete Algorithms*, 7(1):36–48, 2009. ISSN 1570-8667.

[Pedreira07] Pedreira, O. y Brisaboa, N. Spatial selection of sparse pivots for similarity search in metric spaces. *En* J. van Leeuwen, G. Italiano, W. van der Hoek, C. Meinel, H. Sack, y F. Plášil, eds., *SOFSEM 2007: Theory and Practice of Computer Science*, tomo 4362 de *Lecture Notes in Computer Science*, págs. 434–445. Springer Berlin Heidelberg, 2007. ISBN 978-3-540-69506-6. doi: 10.1007/978-3-540-69507-3_37.
URL http://dx.doi.org/10.1007/978-3-540-69507-3_37

[Pestov07] Pestov, V. Intrinsic dimension of a dataset: what properties does one expect? *En Proc. 20th International Joint Conference on Neural Networks*, págs. 1775–1780. 2007.

[Pestov08] Pestov, V. An axiomatic approach to intrinsic dimension of a dataset. *Neural Networks*, 21(2-3):204–213, 2008.

[Pestov10a] Pestov, V. Indexability, concentration, and vc theory. *En Proceedings of the Third International Conference on SImilarity Search and APplications*, SISAP '10, págs. 3–12. ACM, New York, NY, USA, 2010. ISBN 978-1-4503-0420-7. doi:http://doi.acm.org/10.1145/1862344.1862346.
URL http://doi.acm.org/10.1145/1862344.1862346

[Pestov10b] Pestov, V. Intrinsic dimensionality. *SIGSPATIAL Special*, 2:8–11, July 2010. ISSN 1946-7729. doi:http://doi.acm.org/10.1145/1862413.1862416.
URL http://doi.acm.org/10.1145/1862413.1862416

[Putz91] Putz, S. Using a relational database for an inverted text index, 1991.

[Raman02] Raman, R., Raman, V., y Rao, S. S. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. *En Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, SODA

'02, págs. 233–242. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2002. ISBN 0-89871-513-X.
URL http://dl.acm.org/citation.cfm?id=545381.545411

[Ruiz13] Ruiz, G., Santoyo, F., Chávez, E., Figueroa, K., y Tellez, E. S. Extreme pivots for faster metric indexes. *En* N. Brisaboa, O. Pedreira, y P. Zezula, eds., *Similarity Search and Applications*, tomo 8199 de *Lecture Notes in Computer Science*, págs. 115–126. Springer Berlin Heidelberg, 2013. ISBN 978-3-642-41061-1. doi:10.1007/978-3-642-41062-8_12.
URL http://dx.doi.org/10.1007/978-3-642-41062-8_12

[S.02] S., A., S., C., y G., D. Dbxplorer: a system for keyword-based search over relational databases. *Data Engineering, 2002. Proceedings. 18th International Conference*, págs. 5–16, 2002.

[Samet06] Samet, H. *Foundations of Multidimensional and Metric Data Structures*. The morgan Kaufman Series in Computer Graphics and Geometic Modeling. Morgan Kaufmann Publishers, University of Maryland at College Park, 1ª ed$^{\underline{\text{ón}}}$., 2006.

[Sayyadian07] Sayyadian, M., LeKhac, H., Doan, A., y Gravano, L. Efficient keyword search across heterogeneous relational databases. *En ICDE*, págs. 346–355. 2007.

[Shaft06] Shaft, U. y Ramakrishnan, R. Theory of nearest neighbors indexability. *ACM Trans. Database Syst.*, 31:814–838, September 2006. ISSN 0362-5915. doi:http://doi.acm.org/10.1145/1166074.1166077.
URL http://doi.acm.org/10.1145/1166074.1166077

[Skopal04] Skopal, T. Pivoting m-tree: A metric access method for efficient similarity search. *En DATESO'04*, págs. 27–37. 2004.

[Skopal10] Skopal, T. Where are you heading, metric access methods?: a provocative

survey. *En Proceedings of the Third International Conference on SImilarity Search and APplications*, SISAP '10, págs. 13–21. ACM, New York, NY, USA, 2010. ISBN 978-1-4503-0420-7. doi:http://doi.acm.org/10.1145/1862344.1862347.
URL `http://doi.acm.org/10.1145/1862344.1862347`

[Skopal11] Skopal, T. y Bustos, B. On nonmetric similarity search problems in complex domains. *ACM Computing Surveys*, 43(4):34:1–34:50, oct. 2011. ISSN 0360-0300. doi:10.1145/1978802.1978813.
URL `http://doi.acm.org/10.1145/1978802.1978813`

[Stockinger09] Stockinger, K., Cieslewicz, J., Wu, K., Rotem, D., y Shoshani, A. Using bitmap index for joint queries on structured and text data. *En New Trends in Data Warehousing and Data Analysis*, págs. 1–23. 2009.

[Tellez09] Tellez, E. S., Chavez, E., y Camarena-Ibarrola, A. A brief index for proximity searching. *En Proceedings of 14th Iberoamerican Congress on Pattern Recognition CIARP 2009*, Lecture Notes in Computer Science, págs. 529–536. Springer Verlag, Berlin, Heidelberg, November 2009.

[Tellez10] Tellez, E. S. y Chavez, E. On locality sensitive hashing in metric spaces. *En Proceedings of the Third International Conference on SImilarity Search and APplications*, SISAP 2010, págs. 67–74. ACM, New York, NY, USA, 2010. ISBN 978-1-4503-0420-7. doi:http://doi.acm.org/10.1145/1862344.1862355.
URL `http://doi.acm.org/10.1145/1862344.1862355`

[Tellez11a] Tellez, E. S., Chavez, E., y Graff, M. Scalable pattern search analysis. *En 3rd Mexican Congress on Pattern Recognition, MCPR 2011*. Springer Verlag, Lecture Notes in Computer Science, 2011.

[Tellez11b] Tellez, E. S., Chávez, E., y Navarro, G. Succinct nearest neighbor search. *En Proc. 4th International Conference on Similarity Search and Applications, SISAP 2011*. ACM Press, 2011.

[Tellez12] Tellez, E. S. *Practical Proximity Searching in Large Metric Databases*. Tesis Doctoral, Universidad Michoacana de San Nicolás de Hidalgo, México, 2012.

[Tellez13] Tellez, E. S., Chavez, E., y Navarro, G. Succinct nearest neighbor search. *Information Systems*, 38(7):1019–1030, 2013.

[Vidal Ruiz86] Vidal Ruiz, E. An algorithm for finding nearest neighbours in (approximately) constant average time. *Pattern Recognition Letters*, 4:145–157, July 1986. ISSN 0167-8655.

[Volnyansky09] Volnyansky, I. y Pestov, V. Curse of dimensionality in pivot based indexes. *En Proceedings of the 2009 Second International Workshop on Similarity Search and Applications*, SISAP '09, págs. 39–46. IEEE Computer Society, Washington, DC, USA, 2009. ISBN 978-0-7695-3765-8. doi:http://dx.doi.org/10.1109/SISAP.2009.9.
URL `http://dx.doi.org/10.1109/SISAP.2009.9`

[Wu06] Wu, K., Otoo, E. J., y Shoshani, A. Optimizing bitmap indices with efficient compression. *ACM Transactions on Database Systems (TODS)*, 31(1):1–38, 2006.

[Xiao11] Xiao, C., Wang, W., Lin, X., Yu, J. X., y Wang, G. Efficient similarity joins for near-duplicate detection. *ACM Trans. Database Syst.*, 36(3):15:1–15:41, ago. 2011. ISSN 0362-5915. doi:10.1145/2000824.2000825.
URL `http://doi.acm.org/10.1145/2000824.2000825`

[Yianilos93] Yianilos, P.Ñ. Data structures and algorithms for nearest neighbor search in general metric spaces. *En Proceedings of the fourth annual ACM-SIAM Symposium on Discrete algorithms*, SODA '93, págs. 311–321. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1993. ISBN 0-89871-313-7.
URL `http://dl.acm.org/citation.cfm?id=313559.313789`

[Zezula06] Zezula, P., Amato, G., Dohnal, V., y Batko, M. *Similarity Search - The Metric Space Approach*, tomo 32. Springer, Series: Advances in Database Systems, 1ª edón., 2006. ISBN 0-387-29146-6.