UNIVERSIDAD MICHOACANA DE SAN NICOLÁS DE HIDALGO

DIVISIÓN DE ESTUDIOS DE POSGRADO DE LA FACULTAD DE INGENIERÍA ELÉCTRICA

BÚSQUEDA PRÁCTICA POR PROXIMIDAD EN BASES DE DATOS DE GRAN TAMAÑO

TESIS DE DOCTORADO

QUE PARA OBTENER EL TÍTULO DE
DOCTOR EN CIENCIAS EN INGENIERÍA ELÉCTRICA
OPCIÓN EN SISTEMAS COMPUTACIONALES

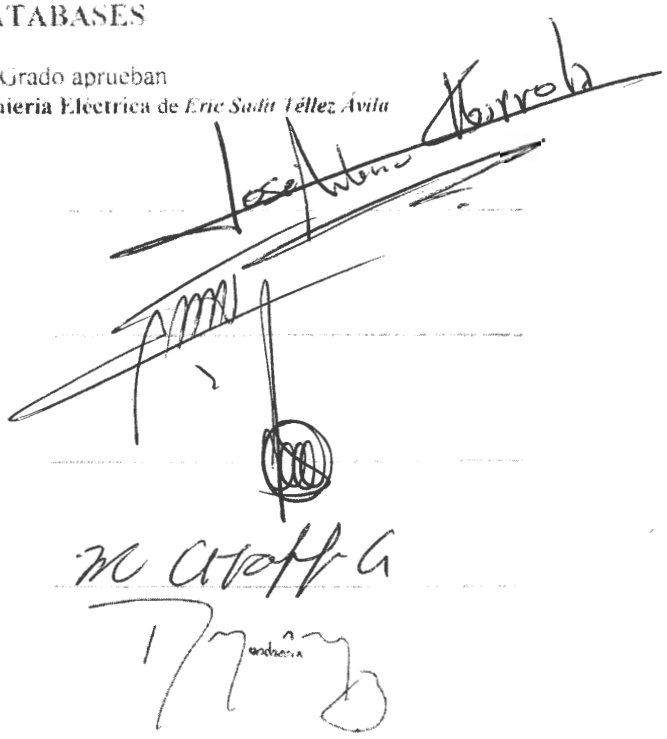
PRESENTA
MC. ERIC SADIT TÉLLEZ AVILA

ASESOR
DR. EDGAR LEONEL CHÁVEZ GONZÁLEZ

MORELIA, MICHOACÁN, JULIO DE 2012

PRACTICAL PROXIMITY SEARCHING IN LARGE METRIC DATABASES

Los Miembros del Jurado de Examen de Grado aprueban
la **Tesis de Doctorado en Ciencias en Ingeniería Eléctrica** de *Eric Sadit Téllez Ávila*



Dr. José Antonio Camarena Ibarrola
Presidente

Dr. Edgar Leonel Chávez González
Director de Tesis

Dr. Félix Calderón Solorio
Vocal

Dr. Mario Graff Guerrero
Vocal

Dr. Rolando Menchaca Méndez
Revisor Externo
Instituto Politécnico Nacional

Dr. J. Aurelio Medina Ríos
Jefe de la División de Estudios de Posgrado
En Ingeniería Eléctrica.

UNIVERSIDAD MICHOACANA DE SAN NICOLÁS DE HIDALGO
Junio 2012

Contents

List of Figures

List of Tables

List of Algorithms

Publications

Journal articles

1. Eric Sadit Tellez, Edgar Chavez, and Gonzalo Navarro. *Succinct Nearest Neighbor Search.* Submitted to Information Systems, Elsevier. A pre-liminar version appeared at "Proc. 4th International Conference on Similarity Search and Applications (SISAP). ACM Press, 2011."

International conferences

1. Eric Sadit Tellez, Edgar Chavez, and Karina Figueroa. *Polyphasic Metric Index: Through Practical Limits of Proximity Search on Metric Spaces.* To appear in Proc. 5th International Conference on Similarity Search and Applications, SISAP 2012. Springer Verlag, Lecture Notes In Computer Science, 2012.

2. Eric Sadit Tellez, Edgar Chavez. *The List of Clusters Revisited* . To appear in Proc. 4th Mexican Congress on Pattern Recognition, MCPR 2012. Springer Verlag, Lecture Notes In Computer Science, 2012.

3. Eric Sadit Tellez, Edgar Chavez, and Gonzalo Navarro. *Succinct Nearest Neighbor Search.* In Proc. 4th International Conference on Similarity Search and Applications (SISAP). ACM Press, 2011.

4. Eric Sadit Tellez, Edgar Chavez, and Mario Graff. *Scalable Pattern Search Analysis.* In 3rd Mexican Congress on Pattern Recognition, MCPR 2011. Springer Verlag, Lecture Notes in Computer Science, 2011.

5. Eric Sadit Tellez and Edgar Chavez. *On Locality Sensitive Hashing in Metric Spaces.* In 3rd Conference on Similarity Search and Applications, SISAP 2010. ACM SIGSPATIAL, 2010.

6. Edgar Chavez and Eric Sadit Tellez. *Navigating k Nearest Neighbor Graphs to Solve Nearest Neighbor Queries*. In 2nd Mexican Congress in Pattern Recognition, MCPR 2010. Springer Lecture Notes In Computer Science, 2010.

7. Carlos Bedregal and Eric Sadit Tellez. *Operators over Compressed Integer Encodings*. In XXIX International Conference of the Chilean Computer Science Society (SCCC), pages 290-297. IEEE CS, 2010.

8. Eric Sadit Tellez, Edgar Chavez, and Antonio Camarena-Ibarrola. *A Brief Index for Proximity Searching*. In Progress in Pattern Recognition, Image Analysis, Computer Vision, and Applications. CIARP 2009, pages 529-536. Springer Verlag, Lecture Notes In Computer Science, 2009.

9. Antonio Camarena-Ibarrola, Edgar Chavez, and Eric Sadit Tellez. *Robust Radio Broadcast Monitoring using a Multi-band Spectral Entropy Signature*. In Progress in Pattern Recognition, Image Analysis, Computer Vision, and Applications. CIARP 2009, pages 587-594. Springer Verlag, Lecture Notes In Computer Science, 2009.

National conferences

1. Edgar Chavez, Eric Sadit Téllez. "Consulta por contenido en colecciones multimedia con natix". 5^{to} Congreso Estatal de Ciencia y Tecnología de COECYT. Nov 12-13, 2009.

List of Symbols

symbol		description
U		The universe of objects.
d		The metric distance function, i.e. $d : U \times U \to \Re$.
(U, d)		A metric space tuple.
S		The database to be indexed $S \subseteq U$.
n		The size of the database, i.e. $n = \|S\|$.
$\mathsf{nn}_{U,S,d}(q)$		The nearest neighbor of q in the database $S \subseteq U$ under d, $q \in U$
$\mathsf{k\text{-}nn}_{U,S,d}(q)$		The k nearest neighbor of q in the database $S \subseteq U$ under d, $q \in U$
R		The set of references $R \subset S$.
σ		The size of R, i.e. $\sigma = \|R\|$.
sim		A similarity function $\mathsf{sim} : \cdot \times \cdot \to \Re$.
f_d		The number of bits required to represent a value as returned by d.
α		A value between 0 and 1, inclusive, meaning the proportion between n and the number of centers in the List of Clusters
β		The exponent determining the number of expected distances of the in the list of clusters searching algorithm, i.e. n^β
t		The threshold for t-threshold algorithms, Chapter 9
t	Appendix A	A parameter fixing the performance of XLB indexes
Σ		An alphabet of size σ, $\Sigma = [1, \cdots, \sigma]$
σ		The size of Σ, $\sigma = \|\Sigma\|$
m		The number of centers of a *compact partition index*
Π_T	Chapter 5	The partition of the database induced by the metric index T
Π_u	Chapters 8 and 9	The permutation of a set of references representing the point of view of the object u

Resumen

En este trabajo presentamos una serie de índices alcanzando excelentes compromisos entre eficiencia en tiempo y memoria. Presentamos índices *exactos*, es decir, aquellos que recuperan el conjunto que satisface completamente la consulta data. Adicionalmente, creamos índices *aproximados*, es decir, algoritmos y estructuras que a costa de la exactitud de la consulta, alcanzan mayores velocidades de búsqueda. En ambas áreas, nuestros índices son rápidos y pequeños, y notablemente, algunos de ellos alcanzan un tamaño cercano al mínimo posible para cada representación.

Nuestros índices *exactos* realizan cerca de la mitad de cálculos de distancia que el estado del arte (usando memoria dentro de los limites prácticos). Así mismo, el tiempo de construcción se reduce de manera sustancial de $O(n^2)$ a $O(\lambda n^{1+\alpha})$, donde n es el tamaño de la base de datos, λ es valor pequeño que depende de la complejidad de los datos (comúnmente entre 1 y 12), y $\alpha < 1$. También introducimos una técnica para compresión de índices métricos que para reduce los requerimientos de almacenamiento hasta la mitad de la versión descomprimida.

En la parte de algoritmos métricos *aproximados* creamos varios índices, todos ellos alcanzando excelentes tiempos y costo en memoria. La calidad de los resultados es muy alta en términos de recall y *proximity ratio* (definidos en el siguiente capítulo).

En particular, creamos una nueva representación para el índice Locality Sensitive Hashing (LSH), basada en una secuencia de símbolos. El índice necesita un espacio cercano al mínimo requerido (éste límite inferior también es parte de nuestra contribución), esta representación es de vital importancia ya que LSH requiere múltiples instancias para mejorar la calidad de los resultados. Es de remarcar que la velocidad no se ve afectada debido a esta nueva representación. También, estudiamos y caracterizamos una nueva familia de índices aproximados para búsqueda por proximidad. La hemos llamado Aproximación por Vecindad (NAPP, por sus nombre en inglés) y una simplificación utilizable en la práctica la llamada las K referencias cer-

canas (K-nr). Los métodos K-nr producen secuencias que al ser comparadas inducen una noción de proximidad. Variando los métodos de comparación y generación de secuencias K-nr creamos una gran cantidad de métodos de búsqueda por proximidad. Todos ellos con un excelente equilibrio entre tiempo de búsqueda, requerimientos de memoria, y la calidad en los resultados. Creamos una representación común a la mayoría de esos métodos, que además engloban a varios índices en la literatura. Además, como resultado de nuestro estudio y caracterización, hemos reducido los requerimientos en memoria de algunos de los índices existentes. Finalmente, a partir de K-nr creamos un nuevo índice llamado *NAPP compressed inverted index*, éste índice es aproximado y comprimido, así mismo, describimos e implementamos varias heurísticas para acelerar las búsquedas y aumentar la compresión de los índices.

Hacemos notar que todos nuestros índices y técnicas incluyen extensos estudios experimentales validando nuestras pretensiones y comentarios. Finalmente, todos nuestros algoritmos, estructuras, y bases de datos se encuentran disponibles como software de libre distribución en la biblioteca natix, `www.natix.org`.

Abstract

This work discusses a serie of proximity searching methods reaching excellent tradeoffs between time and memory. We present *exact* indexes, i.e., those retrieving the exact set of meeting the query's constraints. Also, we create *approximate* indexes, i.e., algorithms and structures trading time and memory performances with the quality of the result. In both fields, our indexes are fast and small. Remarkably, some of them achieve storage sizes really close to the minimum of its representation.

Our *exact* indexes perform half the distance computations of the state of the art methods (using practical amounts of memory). Furthermore, we experimentally demonstrate that the construction cost is reduced from $O(n^2)$ to $O(\lambda n^{1+\alpha})$, where n is the size of the database, λ is a small value dependent of the data's complexity (commonly between 1 and 12), and $\alpha < 1$. Also, we introduce a metric compression technique saving half the storage requirements of the uncompressed index.

In the part of *approximate* methods, we propose several indexes. All of these new indexes achieve excellent tradeoffs between time and memory costs. Also, the quality of the results is quite high, in both *recall* and *proximity ratio*. For instance, we created a new representation of the Locality Sensitive Hashing (LSH), based on sequences of symbols. This index needs a storage space close to the minimum (this lower bound is part of our contribution). The new representation is important because LSH requires several instances to improve the quality of its results. Furthermore, we study and characterize a new family of approximate metric indexes. This new family is called as Neighborhood Approximation (NAPP). Also, a simplification allowing to bound several parameters of NAPP, being of use in practice, is presented and dubbed as K-nr. The point of K-nr is to create sequences with a proximity semantic. Sequences are compared to hint about proximity. Varying the comparison methods we create several approximate searching methods, all of them with an excellent equilibrium between searching time, memory requirements, and result's quality. Moreover, we create a common

representation for the majority of these methods, and some mentioned in the literature. As a result of our study, we reduced the memory storage of some of the methods already available on the literature. Finally, based on a K-nr method, the K-nr-Jaccard index, we created a new representation called the *NAPP compressed inverted index*. This new index is approximate and compressed, and it implements a set of new heuristics to accelerate and improve the compression ratio of these indexes.

Please note that all of our indexes and techniques are extensively tested, validating our claims. Finally, our algorithms, data structures, and databases are freely available as open source software as the natix library, `www.natix.org`.

Part I

Introduction

Chapter 1

Introduction

Proximity or similarity searching is a pervasive problem in computer science, from pattern recognition to textual and multimedia information retrieval, machine learning, streaming compression, lossless and lossy compression, biometric identification and authentification, bioinformatics.

A direct application of proximity search is the searching by content in large multimedia databases. Here, the basic operation consist of retrieving a set of similar objects of the database to a query object. Searching by content in multimedia databases (containing either audio, images, or video objects) requires to represent each item in the database in a way that it can be easily searched. This representation is independent of the original multimedia data, yet, it contains attributes describing the content of the objects. These attributes are commonly stored as vectors of real numbers, sets, strings of symbols, etc. Searching on these data requires the use of techniques like metric searching, where a metric distance function is defined between any two items. The complication comes in two folds, (i) on the complexity of the representation, and (ii), on the size of the dataset. The complexity of the representation produces high dimensional datasets, and in general, proximity searching is exponentially difficult on this parameter (Chavez et al. [Chávez et al., 2001], and Samet [Samet, 2006]). On the other side, large time and memory overheads per object are induced on the majority of metric searching methods, since these are traditionally optimized to reduce the number of distance computations performed to solve a query, regardless of both memory and real time costs. It is important to notice that many techniques literately abuse of these tradition (like AESA [Vidal Ruiz, 1986] or the Permutation based index [Chavez et al., 2008]), inducing large overheads on both time and memory per object.

Our approach is practical, in the sense that our techniques are ready to use on large datasets with high dimensions on real world systems. We care about the number of computed distances, memory cost, and real time cost on both preprocessing and searching steps.

As previously commented, our chosen paradigm for proximity searching is the metric space model, where objects are treated as a *black box*. So, there is not additional knowledge of the internal composition of the item (i.e. the object's structure is not available), and there is a pairwise metric distance function. From a general perspective, given a (large) database S of objects and a metric $d : S \times S \to \Re$, the problem is to retrieve from S the items closer to a given query q. Traditionally, d is considered to be expensive. As a result, every searching method should try to reduce the number of evaluations of d.[1] The common solution consists in preprocessing the database to create an index providing the necessary machinery to efficiently solve queries [Chávez et al., 2001].

The objects can be on any data model, for example, vectors, sets, XML data, trees, graphs, vertices or edges in a graph, text documents, strings, etc. The sole restriction is the existence of a well defined metric function measuring the distance between any pair of objects. In the particular case of vectors, each object is composed of δ numerical coordinates, this is the *explicit* dimension of the vector. In datasets produced by real world processes, some coordinates are highly related, such that the minimum necessary coordinates are commonly less than δ. This value is known in the literature as the *intrinsic* dimension [Chávez et al., 2001]. Contrary to the explicit notion, this complexity measure of the dataset is extended for any metric space, regardless of the data model of the object. Proximity searching techniques based on the object's structure, like KD-Trees or R-trees [Samet, 2006], are tightly dependent of the *explicit* dimension. On the other hand, metric access methods depend on the *intrinsic* dimension. As commented, the intrinsic dimension is embedded in a larger explicit dimension data, hence this is the reason that expensive cost of d, since d works directly with the explicit representation of the objects. Our present work is dedicated to methods being directly dependent of the intrinsic dimension.

Chavez et al. [Chávez et al., 2001] show a simple way to measure the intrinsic dimension using the histogram of distances of a query object to a set of objects. This complexity is a function of the mean μ, and the standard deviation ρ. The precise quantification of the intrinsic dimension of

[1] Our work shows more than this criterion of performance, as will be stated in next sections.

a dataset, as shown by Chavez et al., is $\frac{\mu}{2\rho^2}$. The idea behind this formula is to capture the effect of the intrinsic dimension on the histogram of distances, that is, high intrinsic dimension dataset produces a large μ and small ρ, while small dimensions produce small μ and large ρ values.

This work is devoted to create indexes for proximity searching in high intrinsic dimensional objects and large datasets. The goal is to maintain the computing and storage requirements in practical terms, in the sense that they remain useful on the current power of computing and storage. Naturally, these constraints lead us to work with very robust exact metric indexes (since the majority of them rapidly degrades to an exhaustive review of the dataset), and ultimately, to approximation techniques. Let us define what we mean with *exact* and *approximate* proximity searching algorithms.

- An *exact* method retrieves the exact set of objects meeting the query's constraint.

- In order to speed up the searching process and to reduce the storage requirements, it is a common practice to change the paradigm from exact methods to *approximate* ones. In such methods, we afford either to lose some relevant objects, or it allows to retrieve some not (too) relevant objects to the exact query's constraint.

Exact indexes are of use on databases with a small and moderated intrinsic dimension. On high intrinsic dimensions, it is impossible to avoid the linear scanning of S. This non scalability issue is dubbed as the *Curse of Dimensionality* (CoD) or the *concentration effect*, Chavez et al. [Chávez et al., 2001]. Basically, the CoD implies that any exact index based on the triangle inequality degrades to an exhaustive search as the intrinsic dimension grows. On the next chapter, we will detail our discussion about CoD.

This work is divided on three parts. The first one is dedicated to introduce to the problem, state our methodology, list and describe our datasets, and briefly review the state of the art. The second part describes our contribution to exact proximity searching methods. The last part, introduces our approximate techniques solving similarity queries. The work is finished with Chapter 10, that is dedicated to summarize and conclude our contributions. Also, this document contains in Appendix A a review of indexed sequences, introducing some new structures, being of use in the majority of our metric indexes. Below, we list our contributions in a *roadmap* format.

1.1 Contributions to Exact Techniques

Chapter 3 partially relieves the complexity issues of the List of Cluster (LC) (Chavez and Navarro [Chávez and Navarro, 2005]). Here we introduce a variation of it with smaller preprocessing time. We dubbed as the Reverse Nearest Neighbor List of Clusters (Rev-LC), partially reported in [Tellez and Chavez, 2012]. The Rev-LC is based on a completely different perspective than LC, but using the same data structure. The payment for the preprocessing speed up is a small increment on the searching complexity, particularly noticeable on low dimension datasets. Also, this chapter show us how to parallelize the List of Clusters, and the resulting techniques are developed and applied in Chapter 4. The parallelization of the LC is of great help on moderated sized datasets and modern multi-core architectures.

In Chapter 5 we introduce a new metric index overcoming many of the limitations of LC. This contribution has been dubbed as Polyphasic Metric Index (PMI), partially reported in [Tellez et al., 2012]. The improvement comes in several folds:

- PMI produces indexes that are more robusts to the intrinsic dimension than LC, yielding to faster searching times.

- The preprocessing time is dramatically reduced, even on high intrinsic dimension datasets. It goes from $O(n^2)$ distance computations of the LC to $O(\lambda n^{1+\alpha})$, where $\alpha < 1$ and λ is a small integer dependent of the dataset.

As will be shown, the PMI improves by far all known tradeoffs preprocessing time, memory space and searching time complexities.

Finally, Chapter 6 introduces a new compact representation of metric indexes, particularly, we focus on those based on LC (e.g. Rev-LC and PMI). We create indexes using close to optimal space, that is $nH_0 + n^\alpha f_d$ bits. Here H_0 (smaller or equal than $\alpha n \log n$) is the zero order entropy of an alternative representation of LC, and f_d is the number of bits required to store any value of d. Notice that $n/n^\alpha = O(1)$ for LC, but it is expressed on this detail for technical convenience that will be evident in next paragraphs. Also, we introduce two variations for special cases, using only nH_0 bits or $n^\alpha f_d$ bits. Remarkably, the compressed LC, has a small searching time overhead. For compression, we use compact and compressed indexes for sequences [Navarro and Mäkinen, 2007; Claude and Navarro, 2008; Golynski et al., 2006].

Using the compressed LC technique, the PMI becomes a metric index using $\lambda(nH_0+n^\alpha f_d)$ bits, where f_d is the number of bits required to represent a distance value.

As the intrinsic dimension increases, sooner or later all exact techniques are condemned to degrade to exhaustive review of the database. For those cases, approximation techniques are the only affordable option.

1.2 Contribution to Approximate Techniques

On the approximate proximity searching problem, we found several indexes performing excellent tradeoffs between preprocessing time, searching time, memory cost, and result's quality.

A new implementation of the well known Locality Sensitive Hashing (Gionis et al. [Gionis et al., 1999]) is introduced in Chapter 7. Our LSH implementation is represented in close to optimal space. This new implementation is particularly efficient in situations requiring multiple indexes; which is quite common when high quality results are necessary. Also, we introduce two new primitives (SuccCtx and PredCtx, retrieving the context of an object) supported by our representation without requiring additional memory.

Chapter 8 shows a novel technique called *Neighborhood Approximation* (NAPP). Also it is simplified in a simple framework named K nearest references (K-nr). The last simplification is used to create several indexes, all of them sharing interesting properties in quality's results, preprocessing and searching time, and compression capabilities. At the best of our knowledge, our indexes are the very first metric indexes on the literature to be primary designed to be compressed. Those indexes were partially reported in [Tellez et al., 2009; Tellez and Chavez, 2010; Tellez et al., 2011a,b]. Even when our indexes are designed to work on general metric spaces, the majority of them do not use the metric properties of d, such that similarity spaces can be used as well.

Knr indexes allow us to solve searches efficiently, while achieves recalls bigger than 90% reviewing less than a 1% of the database. On recall terms, our approximated similarity search techniques achieves performances reaching the state of the art methods [Chavez et al., 2008; Amato and Savino, 2008; Esuli, 2009; Gionis et al., 1999; Andoni and Indyk, 2008], and better than the majority in searching time. The construction cost of our indexes require σn computations of d, and at most $Kn \log \sigma$ bits of storage; where K is a small constant (e.g. 7), and σ is a parameter denoting the cardinality

of a set of objects called *references*. Notice that $\sigma \ll n$.

Finally, Chapter 9 shows an alternative representation of a K-nr index. Based on compressed inverted indexes, where each sorted list is in fact a compressed bitmap, this new approach achieves better compression ratios while introduce several new speed enhancements being times faster than the original NAPP indexes.

1.3 Methodology

Due to the diversity of solutions addressed in this work, and our practical perspective, we show the behaviors of our techniques *in situ*, i.e., we introduce a vast number of experimental evidence in the presenting chapter. So, we describe here the datasets that will be used along chapters, and the methodology followed by our experiments.

1.3.1 Cost Measures and Our Practical Perspective

In Chavez et al.[Chávez et al., 2001] the performance of indexes is measured in terms of distance computations, since it is considered to be the most expensive operation. The *inner* distances are computed navigating the index to obtain a set of candidate objects. This set of candidates is sequentially scanned to obtain the set of relevant objects. The distances computed to obtain the last set are named *outer* distances. From this perspective the CoD comes in the form of having a very large set of candidate objects for a given query, i.e. it shows a low filtering power.

For most practical applications the distance is not too expensive to be considered as the unique countable operation [Skopal, 2010]. For example, real world metric databases/applications require relatively low-cost distance functions like edit distance for dictionaries (with small words), hamming spaces, vector spaces with some Minkowski's norm, etc.

Counting only distance computations is specially problematic when n is sufficiently large such that the linear scanning of the database and the (at least) super-linear space requirements of a table of pivots is prohibitive, even when the number of final candidates is very small. For example, the most efficient algorithm in terms of distance computations (i.e. AESA [Micó et al., 1994]) needs $O(1)^{\delta}$ distance computations for a δ-dimensional space with n objects [Chávez et al., 2001; Navarro, 2009]. Nevertheless, it requires $O(n^2 f_d)$ bits of space, where f_d is the number of bits required to represent a d's computed value; and $O(n^2)$ numeric and logical operations, [Chávez

et al., 2001; Micó et al., 1994]. Hence, it is unpractical, even for medium sized databases.

The main product of our work is a set of indexes for proximity searching, that successfully deal with real world scenarios allowing to be used in practice. Thus, we show the cost of our indexes with several indicators:

— The number of *distances computed* to perform an operation (e.g. construct and index or solve a query).

— The *real time* is shown, even when it is not always a fair comparison, since it depends on many parameters (e.g. hardware, programming language / compilers, operating system, etc.). In contrast, the real time is an excellent criterion to judge if some technique is practical. This is the main indicator for practitioners, and in equality of testing conditions, it can be used as an important comparison among different techniques.

— *Memory* usage is given. The storage requirement is an important factor to considere in practice. It is important to notice that many metric indexes are not aware of the space requirements, using a few integers per object (i.e. pointers plus satellite data) which can be considered a high overhead per object. In contrast, most of our indexes are compressed and its average cost is of just a few bits per object.

— For our approximate indexes, the *recall* is an important quality measure. We define *recall* as the ratio of retrieved items that are part of the exact result, i.e., $|\mathcal{A} \cap \mathcal{E}|/|\mathcal{E}|$ where $\mathcal{A}$ is the set of approximate results and $\mathcal{E}$ the exact result.

— In the same way, the *proximity ratio* is presented. We define the *proximity ratio* as the quotient of the approximate covering radius versus the exact covering radius of result, i.e., $\frac{\max_{u \in \mathcal{A}} d(u,q)}{\max_{v \in \mathcal{E}} d(v,q)}$. This is a relaxed measure of the quality that is quite popular in both practical and theoretical approaches. For example, in most multimedia information retrieval applications, a result is considered good if retrieved objects are simply close to the query, and they do not need to be part of the exact set of closer objects .

1.3.2 Developing and Running Environment

All the algorithms were written in C#, with the Mono framework (`http://www.mono-project.org`). Algorithms and indexes are available as open source software in the *natix* library (`http://www.natix.org`, and `http:`

//github.com/sadit/natix/). Unless another setup is indicated, all experimentations were executed in a 16 core Intel Xeon 2.40 GHz workstation with 32GiB of RAM, running CentOS Linux. The entire databases and indexes were maintained in main memory and without exploiting multiprocessing capabilities of the workstation, except for parallel algorithms (explicitly mentioned).

1.3.3 Description of the Datasets

Metric access methods are independent of the underlaying representation of the objects. The same is true for particular metric functions. However, in the end, objects must be represented in some way, and distances need to be well defined too.

Vectors are tuples of δ numbers. They are commonly measured with the Minkowski's L_p family of distances, defined as:

$$L_p(u, v) = \left(\sum_{i=1}^{\delta} |u_i - v_i|^p \right)^{1/p} \tag{1.1}$$

An L_p evaluation requires $O(\delta)$ basic operations. For instance, the L_1 distance corresponds to the *Manhattan* distance, since in two dimensions resembles the distance necessary to travel between points in a city of rectangular blocks. L_2 is better known as the *Euclidean* distance, and it corresponds to our notion of spatial distance. Finally, another prominent Minkowski's distance is $L_\infty = \max_{1 \leq i \leq \delta} |u_i - v_i|$.

In information retrieval, *TFIDF vectors* are representations of documents in the measured with the cosine similarity,

$$\mathsf{sim}_C(u, v) = \frac{\sum_{i=1}^{\delta} u_i v_i}{\sqrt{\sum_{i=1}^{\delta} u_i^2} \cdot \sqrt{\sum_{i=1}^{\delta} v_i^2}} \tag{1.2}$$

Metric methods should use $d_C(u, v) = \arccos \mathsf{sim}_C$, that is, the angle between vectors.

The Hamming's distance is applied to strings of fixed size ℓ. The strings are aligned and the distance counts how many corresponding symbols are different.

$$d_H(u, v) = \sum_{i=1}^{\ell} \begin{cases} u_i = v_i & 0 \\ u_i \neq v_i & 1 \end{cases} \tag{1.3}$$

An evaluation requires a $O(\ell)$ operations. For instance, binary Hamming's distance simply counts the number of differences between two bit strings.

However, the binary case can be efficiently computing defining x as the bit vector resulting of bitwise xor-ing u and v, processing $\log n$ items at once. Then, $d_C(u, v)$ is computed summing up the number of enabled bits (popcount) of all non overlapping substrings of x of size $w \leq \frac{\log n}{2}$. So, popcount is a table of 2^w entries. It stores the number of enabled bits for each bit-string of w bits, such that it requires $2^w \log(w+1)$ bits (at most $\sqrt{n} \log(1 + \log n)$ bits). Since w bits can be processed in constant time in a RAM machine, the time cost of $d_H(u, v)$ is of $O(\ell/w)$.

The Jaccard distance for any two sets u, v requires $O(|u|+|v|)$ operations in the worst case. Is defined as:

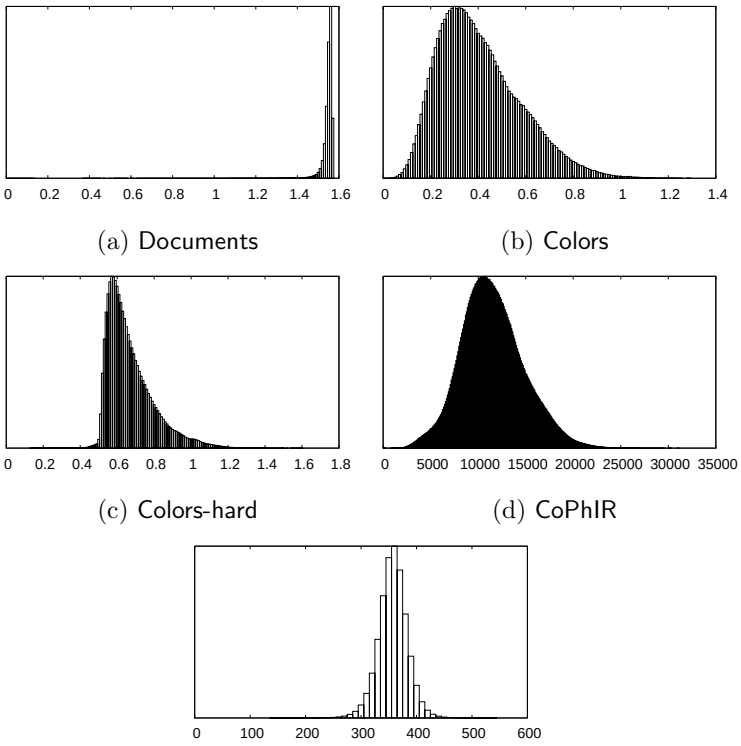
$$d_J(u, v) = 1 - \frac{|u \cap v|}{|u \cup v|} \tag{1.4}$$

List of Datasets

In order to give a rich description of the behavior of our techniques, we select several real world databases and generate synthetic ones, as detailed below. It is worth noticing that even if our datasets are vector spaces, we are not using the coordinates to discard elements. We use the distance as a black box. This allows to work with the data disregarding its representation, all we need is a distance function to index the data.

Name	$\frac{\mu}{2\rho^2}$	d_{max}	μ	ρ
Documents	982.99	1.571	0.985	0.022
Colors	8.59	1.38	0.302	0.032
Colors-hard	36.32	1.73	0.39	0.073
CoPhIR	19.31	32682	0.357	0.096
Audio	142.04	560.0	0.637	0.047
RVEC-4-*	11.22	1.80	0.426	0.138
RVEC-8-*	20.21	2.19	0.509	0.112
RVEC-12-*	29.78	2.55	0.546	0.096
RVEC-16-*	38.21	2.81	0.580	0.087
RVEC-20-*	45.39	2.95	0.607	0.082
RVEC-24-*	55.17	3.21	0.615	0.075

Table 1.1: Statistics of our datasets. The mean and the standard deviation, μ and ρ respectively, are relatives to d_{max}. $\frac{\mu}{2\rho^2}$ is the intrinsic dimension as described by Chavez et al. [Chávez et al., 2001].



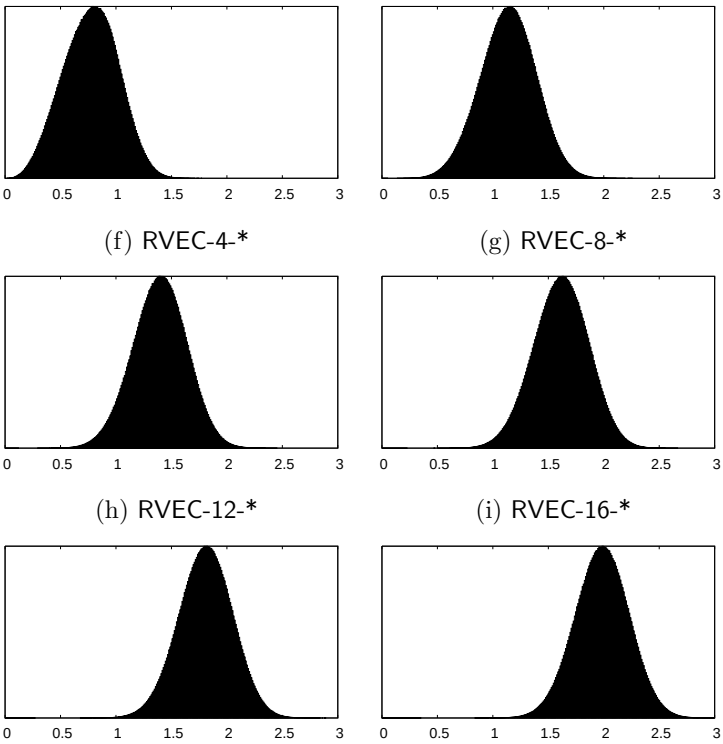
(a) Documents (b) Colors

(c) Colors-hard (d) CoPhIR

(e) Audio

Figure 1.1: Histograms of distances of our datasets.

— **Documents** This database is a collection of 25157 short news articles in the $TFIDF$ format from TREC-3 collection of the Wall Street Journal 1987 – 1989 (taken from the SISAP project [Figueroa et al., 2009], `http://www.sisap.org`). We use the angle between vectors as distance measure [Baeza-Yates and Ribeiro-Neto, 1999] and extracted 100 random documents from the collection as queries (note: these documents were not indexed). The objects are vectors of hundred of thousand of coordinates. Figure 1.1a shows the histogram of distances, we must remark that this dataset has a high intrinsic dimension in the sense described by Chavez et al. [Chávez et al., 2001], i.e., $\frac{\mu}{2\rho^2}$ where μ is the mean and ρ is for the standard deviation. The intrinsic dimension of our datasets is shown in Table 1.1. Even single nearest



(f) RVEC-4-* (g) RVEC-8-*

(h) RVEC-12-* (i) RVEC-16-*

(j) RVEC-20-* (k) RVEC-24-*

Figure 1.1: Histograms of distances of our datasets.

neighbor queries require to review the entire database in most exact metric indexes. As a reference, a sequential scan needs 0.23 seconds.

— **Colors** The second benchmark is a set of 112682 color histograms (112-dimensional vectors) from sisap, under the L_2 distance. This dataset is distributed by the SISAP project [Figueroa et al., 2009], available at `http://www.sisap.org`.

Each query q is created with a simple linear composition of two randomly selected objects of the database. For example, let u, v be in the dataset such that $q_i = (u_i + v_i)/2$.

An alternative set of queries is used for approximate algorithms. We choose randomly 200 histogram vectors and applied a *perturbation* of ± 0.5 on one random coordinate. Figure 1.1c shows the histogram of distances. In this case, the difficulty comes from our perturbed query set, since the perturbation is a third of the maximum distance. Also, the intrinsic dimension grows four times, Table 1.1. We dubbed this mixture (data and query set) as Colors-hard.

— **CoPhIR** We use two subsets of the CoPhIR database, of 1 and 10 million objects selected from the CoPhIR project [Bolettieri et al., 2009], dubbed as CoPhIR-1M and CoPhIR-10M respectively. Each object is a 208-dimensional vector and we use the L_1 distance. Each vector is a linear combination of five different MPEG7 vectors as described in [Bolettieri et al., 2009]. We choose 200 vectors (not indexed) as queries. Figure 1.1d shows the histogram of distances. The complexity of this database comes from its size, since the concentration around the mean is not so large. A sequential scan takes approximately 60 seconds. The intrinsic dimension as described by Chavez et al. [Chávez et al., 2001] is of 19.31.

— **Audio** A set of more than 55 million bit-strings obtained from the audio fingerprinting process of Camarena and Chavez [Ibarrola and Chávez, 2006]. The intrinsic dimension is of approximately 142, Table 1.1, yet each object is composed of 720 boolean attributes. The closeness is measured with the Hamming distance. We randomly choose 256 bit-streams from the database as queries. In order to prepare the queries we flip 3% of its bits such that the expected distance is of at most 21. The distance function is quite fast, a sequential scan on the 55 million objects takes 120 seconds. The intrinsic dimension is high, $\frac{\mu}{2\sigma^2} = 142$.

— **RVEC** In order to show behaviors parametrized by the dimensionality of the datasets, we generate three vector databases of 250000, 500000, and 1000000 objects, over six different dimensions; i.e. 4, 8, 12, 16, 20, and 24 coordinates. These databases are dubbed with the pattern RVEC-dimension-n. We allow us to use the symbol * as *wildcard*, in order to refer to sets of datasets. Each vector was generated randomly in the unitary hypercube of the appropriate dimension. In our table of dataset's statistics, the intrinsic dimension (as shown in Table 1.1) is larger than two times the number of explicit coordinates. The query set contains 200 random vectors. Query sets and databases are disjoint sets with high probability.

1.3.4 Computation Model

As integral part of our approach, we also analyze most of our metric indexes and data structures. Unless another premise is given, we suppose that our algorithms are running on a RAM machine, that is, we can access $\log n$ bits in $O(1)$ time, on a memory that can represent n values per cell. Also, all basic arithmetic and logic operations have a constant time cost.

When metric indexes are analyzed, we suppose the distance evaluation as the unitary cost. Other operations can be ignored. Commonly, in this work, this paradigm is switched to comment a more realistic cost, since this is historically overexploited in the literature. This is particularly self-evident as distance computations time are overwhelm by other basic operations.

As commented before, our approach is quite practical, so, our experimental sections will address both computation measures counting distances and real time.

1.4 Summary

In this chapter we introduces the proximity searching problem, and lists our contributions to the area. Furthermore, we state the methodology that will be followed in the rest of this thesis, and briefly describe the properties of our datasets, like size, source, intrinsic dimensionality, query sets, and their data models. Also, we give a small discussion of the major distance functions used in this work.

The next chapter discuss the state of the art in the field, and introduces the basic tools to create our new metric indexes.

Chapter 2

Foundations

This work is aimed to data-structures and algorithms performing proximity searches, on large and high intrinsic-dimensional datasets. These requirements set a complicated scenario for most traditional techniques, thus we try to find adequate tradeoffs among memory, preprocessing time and searching time, such that our techniques can be used on current systems. This is a very simple (yet vague) definition of our goals, but it is enough to say that we are interested on proximity searching techniques performing well with the current power of computers.

In this work, we review a set of proximity searching algorithms and techniques achieving and surpassing the current state of the art. This chapter discuss the foundations on proximity searching, and review the current state of the art on the field. Also, since our algorithms and data structures can be stated as indexed sequences (and its operations), so, we include a short review of this kind of indexes and list its properties and operations at the end of this chapter.

2.1 Searching in Metric Spaces

A metric space is a tuple (U, d) where U is a domain, $S \subseteq U$ is a finite subset (the *database*) of U with size $n = |S|$, and $d : U \times U \to \Re$ is a *distance function*. Formally, d obeys the following properties $\forall u, v, w \in U$.

— It is positive, $d(u, v) \geq 0$ and $d(u, v) = 0 \iff u = v$.

— It is symmetric, i.e. $d(u, v) = d(v, u)$.

— It holds the triangle inequality, $d(u, w) + d(w, v) \geq d(u, v)$.

In this work we are interested in two proximity operations over the database:

— Searching for the k nearest neighbors $\mathsf{k\text{-}nn}_{U,S,d}(q)$. Retrieves (at most) k closest objects to q in S, for $q \in U$ and $S \subseteq U$. Formally, it retrieves the set $\mathsf{k\text{-}nn}_{U,S,d}(q) = \{u \mid d(u,q) \leq d(v,q)\ \forall u, v \in S\}$ where $|\mathsf{k\text{-}nn}_{U,S,d}(q)| = k$, since $k \leq n$ for our purposes.

  For technical reasons, to be used later on this work, we define a version of the $\mathsf{k\text{-}nn}$ problem slightly more restrictive than the one usually found in the literature. Let $\mathsf{k\text{-}nn}_{U,S,d}(q) = [u_1, u_2, \cdots, u_k]$ such that $d(u_1,q) \leq d(u_2,q) \leq \cdots \leq d(u_k,q)$ and $d(u_i,q) \leq d(v,q)\ \forall v \in S \setminus \mathsf{k\text{-}nn}_{U,S,d}(q)$. In other words, we care about the order of the elements.

— Range query $(q,r)_{U,S,d}$. 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. $(q,r)_{U,S,d} = \{u \in S \mid d(q,u) \leq r\}$.

Our exact techniques are attending both problems, while our approximate techniques are focused on the $\mathsf{k\text{-}nn}$ problem.

In order to simplify our notation, we allow us to write $\mathsf{k\text{-}nn}_{S,d}$, $(q,r)_{S,d}$; $\mathsf{k\text{-}nn}_d$, $(q,r)_d$; and $\mathsf{k\text{-}nn}$, (q,r) whenever the context provide enough information to avoid confusions, or when a generic reference to the operation is required. With the same purpose, we define $\mathsf{nn}_{U,S,d}(q)$ as $1\text{-}\mathsf{nn}_{U,S,d}(q)$.

2.2 Related Work

There exists two main classes of indexes for general metric spaces: pivot based and compact partition indexes.

2.2.1 Pivot Based Indexes

The purpose of an index is to avoid a sequential scan. The pivot trick consist in filtering the database S by using repeatedly the triangle inequality to bound the distance from an object to the query. A set of distinguished points $P = \{p_1, p_2, \cdots, p_m\} \subseteq U$ (the pivots) are used to define a filtering distance, always bounded from above by the original distance d. Let $D(u,v) = \max_{1 \leq i \leq m} |d(u,p_i) - d(v,p_i)|$. Using the triangle inequality, it is inmediate $D(u,v) \leq d(u,v)$ and hence it implies $(q,r)_d \subseteq (q,r)_D$ as detailed by Chavez et al. [Chávez et al., 2001].

The index retrieve $(q,r)_D$ using only m distance computations, just distances to the pivots. When the intrinsic dimension of the data set is high,

the CoD implies that even a significant increase in the number of pivots (say to the limits of the available memory to store the distance matrix), barely decreases $(q,r)_D \setminus (q,r)_d$. For easier instances of metric spaces the decrease may be significant, implying that a pivot based index will have a single parameter for the end user, i.e., if the time to get an answer is not satisfactory, the number of pivots should be increased.

The above simple rule can be used as long as the cost of obtaining $(q,r)_D$ and the amount of memory used to maintain the distances is bounded. A plain table of m pivots is neither efficient for processing $(q,r)_D$ nor efficient in space usage.

In a tree data structure as the Fixed Height Fixed Queries tree by Baeza-Yates and Navarro [Baeza-Yates and Navarro, 1998] the time is sublinear but this comes with the overhead of maintaining pointers in addition to the mn distances. Other pivot based tree data structures like that presented by Ullman [Uhlmann, 1991], and Burkhard and Keller [Burkhard and Keller, 1973] cannot use more pivots because they can only represent as much distances as the path length (the sum of the paths from every node to the root). One interesting alternative is the Fixed Queries Array (FQA) of Chavez et al. [Chávez et al., 2001] which uses a few bits per pivot, and a logarithmic penalty over a tree data structure.

The limit in the number of pivots usable for indexing is the size of the database (unless pivots outside the database are used for indexing). Vidal [Vidal Ruiz, 1986] introduces AESA, an index using all objects in the database as pivots. The index consists of a $n(n+1)/2$ array of distance values. Obviously, the preprocessing time has the same order (counting computed distances). AESA exposes experimental evidence that the number of computed distances to solve a query is independent of the size of the database, but with an exponential dependency on the intrinsic dimension. However, it requires a quadratic number of arithmetic and logical operations. Linear-AESA (LAESA) [Micó et al., 1994] is a memory improvement over the n pivots of AESA, it uses m fixed number of pivots (dependent on the intrinsic dimension). AESA and LAESA store $n^2 f_d$ bits and mnf_d bits, respectively, these spaces are unpractical, and they are only of use for small databases.

2.2.2 Compact Partition Indexes

Another approach to proximity searching consist in partitioning the database in compact regions. Most of the compact partitioning indexes in the literature are hierarchical, with a recursive rule as follows: A set of centers

$c_1, c_2, \cdots, c_m \in S$ is selected per node, such that every c_i is the center of a subtree T_i. The set of centers are used to partition the database such that each T_i is spatially compact. For example, $u \in S$ is linked to the subtree T_i such that $i = \arg\min_{1 \leq i \leq m} d(c_i, u)$. The covering radius $\mathsf{cov}(c_i) = \max_{u \in T_i} d(c_i, v)$ is stored for each node. This construction is applied recursively. A query $(q, r)_d$ is solved recursively starting from the root node. If $d(q, c_i) \leq r$ then $c_i \in (q, r)_d$, and T_i must be explored if $|d(q, c_i) - \mathsf{cov}(c_i)| \leq r$.

In general the recursion can be stopped at any level, and objects below that level in each node are stored together in a bucket. In this type of indexes there is not a simple recipe to increase the searching performance as in the case of pivot indexing. Below, a list of seminal work is presented.

Kalantari and McDonald [Kalantari and McDonald, 1983] propose a binary tree called Bisector Tree (BST). Let w be the root node, the left child is associated with the center c_1, and the right child with c_2. For each u in $S \setminus \{c_1, c_2\}$ if $d(u, c_1) \leq d(u, c_2)$ then u is inserted into the left subtree recursively, otherwise it is inserted into the right subtree in the same way. Each child stores its covering radius, so $\mathsf{cov}(\cdot)$ gives the pruning information. The procedure is recursively applied. Uhlmann [Uhlmann, 1991] introduces the Generalized Hyperplane Tree (GHT). The construction is identical to BST, but it does not stores the covering radius. At query time, $(q, r)_d$, the left subtree is visited if $d(q, c_1) - r < d(q, c_2) + r$. In the same way, the right tree requires to be reviewed if $d(q, c_2) - r < d(q, c_1) + r$.

Brin [Brin, 1995] introduces the Geometric Near-neighbor Access Tree (GNAT), which is similar to GHT, but with nodes of a large arity m. Notice that a GNAT is quite similar to a recursive Voronoi partition of vector spaces. In addition to the basic structure, a table of $O(m^2)$ entries containing the minimum and maximum distances of each center to all items in all subtrees. This table is used like a pivot filtering table, that is, at query time a c_i is selected and $d(q, c_i)$ is evaluated; this distance is used to filter subtrees with the information of this table and the tringle inequality constraints. The index requires $O(nm^2)$ identifiers and distances. Experiments performed by Chavez et al. [Chávez et al., 2001], suggest that only large arities are of use, this implies a very large memory overhead per object.

Ciaccia et al. [Ciaccia et al., 1997] introduces the M-Tree, a dynamic and disk based metric index. The preprocessing step resembles to GNAT, yet only the covering radius is stored at each node. The searching procedure is similar to BST. However, the dynamic operations are the distinguishing property of the M-Tree, those algorithms take similar decisions to B-Tree or R-Tree [Cormen et al., 2001].

The List of Clusters

Chavez and Navarro [Chávez and Navarro, 2005] present a robust and memory efficient alternative, dubbed as the List of Clusters (LC). The LC needs $O(n)$ integers for the index and mf_d bits to store distance values, where m the number of centers and f_d is the number of bits required to store a distance value. The preprocessing step requires close to $mn/2$ evaluations of the distance function. Nevertheless, as explained by Chavez and Navarro [Chávez and Navarro, 2005], high intrinsic dimensional datasets must follow that $n/m = O(1)$. So, with this setup, the LC computes $O(n^2)$ distances on the preprocessing step. On the other side, the searching time is $O(n^\beta)$, for some $\beta \leq 1$ dependent of the database. At equality of memory, it is unbeatable on datasets with high intrinsic dimension.

The original idea of LC was to unbalance a tree data structure until it becomes a linked list. While this is a very bad idea for exact searching; where achieving balance is a paramount and a myriad of balancing algorithms exist, in the case of approximate searching it has proven to be of use. The drawback of the approach is a quadratic construction time and has the same origin of its unmatched performance.

Let explore with more detail the construction and the searching algorithms of the LC (taken from [Chávez and Navarro, 2005]). Define $I_{S,c,\mathsf{cov}(c)} = \{u \in S \setminus \{c\} \mid d(c,u) \leq \mathsf{cov}(c)\}$ as the bucket of *internal* elements, which lie inside center ball of c, and $E_{S,c,\mathsf{cov}(c)} = \{u \in S \mid d(c,u) > \mathsf{cov}(c)\}$ as the rest of the elements (the *external* ones). Now the process is repeated recursively inside E. The construction procedure returns a list of triples (c_i, r_i, I_i) (center,radius,bucket) and it is shown in Figure 2.1 and formalized in Algorithm 1.

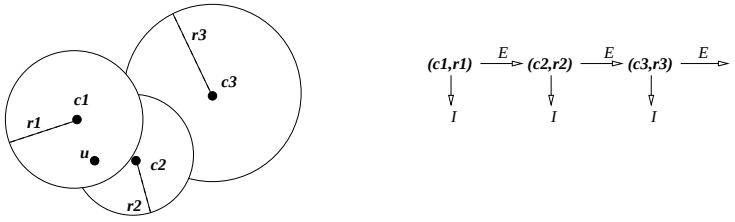


Figure 2.1: The influence zones of three centers taken in this order: c_1, c_2, c_3. On the right, the list arrangement for the data structure. On both figures $\mathsf{cov}(c_i) = r_i$.

Please note that the number of centers in the Algorithm 1 is unknown beforehand. There are two possible parameters, the number of objects inside

algorithm 1: The construction algorithm of the LC. The operator :: is the list constructor. It is not hard to remove the tail recursion to make it iterative.

Input: The set of objects to be indexed, m.
Output: The list of clusters.
Build(S)

```
1: if S = ∅ then
2:    return empty list
3: end if
4: Select c ∈ S
5: Select a radius cov(c)
6: I ← {u ∈ S \ {c}, d(c, u) ≤ cov(c)}
7: E ← S \ I
8: return (c, cov(c), I)::Build(E)
```

a ball, or the radius of the ball. This defines indirectly the number of centers. As proposed in the original paper, we select the number of centers, i.e. n/m, as a simple way to select the required parameters.

algorithm 2: The search algorithm. The main loop (line 2) visits triples in the order specified on L.

Input: The list of clusters L, the query $(q, r)_d$.
Output: The result set R.

```
1: Let R ← ∅
2: for all (c, cov(c), I) ∈ L do
3:    Let d_cq = d(c, q)
4:    R ← R ∪ {c} if d_cq ≤ r
5:    if d_cq ≤ cov(c) + r then
6:       for all u ∈ I do
7:          R ← R ∪ {u} if d(u, q) ≤ r
8:       end for
9:    end if
10:   stop loop if d_cq < cov(c) − r
11: end for
```

The searching procedure is described in Algorithm 2, here the focus is to solve three cases on each node. Let us define $(c, \mathsf{cov}(c))$ as the ball centered at c with radius $\mathsf{cov}(c)$, then there are three possible cases on each center, as follows.

(i) The query ball is intersecting $(c, \mathsf{cov}(c))$, in such case we need to review all objects in I_c.

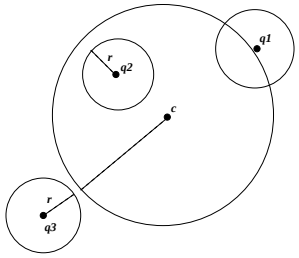


Figure 2.2: An illustration of the three cases of query ball versus center ball. For q_1 we need to consider the current bucket and the rest of centers. For q_2 we can prune the search inside the rest of the partitions. For q_3 we can avoid considering the current bucket.

(ii) The query ball is completely contained by $(c, \mathsf{cov}(c))$, in such case we can stop the searching algorithm.

(iii) The query ball is not intersection $(c, \mathsf{cov}(c))$, so I_c cannot contains any result object.

These cases are illustrated in Figure 2.2. When the intrinsic dimensionality of the data is high, then most of the balls need to be reviewed.

Chavez and Navarro [Chávez and Navarro, 2005] use a set of probabilistic arguments to show the complexity of the searching, showing that it must be $O(n^\beta)$ distance computations, for some $\beta \leq 1$ which depend on the distribution of the data.

2.2.3 Approximate Proximity Searching

As intrinsic dimension grows, the linear scanning of S becomes mandatory. This non scalability issue is dubbed as the *Curse of Dimensionality* (CoD) or the *concentration effect*. This effect has been researched in depth (among others) by Pestov [Pestov, 2010b,a, 2007, 2008], Volnyansky and Pestov [Volnyansky and Pestov, 2009], Shaft and Ramakrishnan [Shaft and Ramakrishnan, 2006], and nicely surveyed by Indyk [Indyk, 2004], Samet [Samet, 2006], Hjaltason and Samet [Hjaltason and Samet, 2003], Chavez et al. [Chávez et al., 2001], Böhm et al. [Böhm et al., 2001], and Skopal and Bustos [Skopal and Bustos, 2011], just to mention some of the most influential literature on the area. The CoD is a statistical condition present in every high intrinsic dimensional data set describing a non desirable characteristic: most of the data looks far and equally distant to any given query, then it is practically

impossible to avoid sequential scanning only using the triangle inequality. A relaxed characterization uses the histogram of distances, here a small standard deviation and a large mean indicate a high intrinsic dimension [Chávez et al., 2001]. Up to the best of our knowledge no exact index can overcome the CoD.

In this case a metric index should be tweaked to support approximation techniques in searches when the users can afford loosing relevant answers, or receive irrelevant ones. A common approach to alleviate the CoD is to use *approximate* proximity search algorithms trading speed for accuracy. One way is to convert an exact algorithm to an approximate one using the procedure described by Chavez and Navarro [Chávez and Navarro, 2003], Zezula et al. [Zezula et al., 2006], and Bustos & Navarro [Bustos and Navarro, 2004] that consists in aggressively reducing the radius by multiplying it by a stretching constant. This technique is specially useful for tree based indexes.

Skopal [Skopal, 2007] shows an alternative model unifying both exact and approximate proximity search algorithms. Also, he introduces the TriGen algorithm, a method to envelop a (dis)similarity function into a metric function with high probability. This technique allows to index general similarity spaces using metric indexes.

Kyselak et al. [Kyselak et al., 2011] observed that most of approximate methods optimize the average accuracy of the indexes, but it is common to find very bad performances on individual queries. So, they propose a simple solution to stabilize the accuracy. The idea is to reduce the probability of a poor-quality result using multiple independent indexes solving a query in parallel, so, with high probability at least one index achieves good quality on a result. Nevertheless, this technique increases both the searching time and memory costs.

Even with these probabilistic techniques, the internal cost of the searches and the number of distance computations is sometimes too high for practical applications. Other techniques are designed to be approximate from the beginning, and commonly scale better in both the size and the intrinsic dimensionality of the database. Recently, a novel branch of approximate algorithms has emerged reaching amazing performance in large databases and high intrinsic dimensions. The following techniques are tightly related to our indexes, and they are considered the state of the art, so we will discuss them below in detail.

Comparing Permutations

Chavez et al. [Chavez et al., 2008] shown a new proximity search method based on comparing permutations, it is called the *Permutation Index* (PI). The main idea is to describe objects as its perspective of a set of references (R). Chavez et al. named R as the set of *permutants*. The proximity is computed using the relative movements of the permutants. $|R|$ is very small, this makes it particularly efficient for expensive distance computations, and small databases. The scalability remains as a main problem, even considering the efforts of Figueroa and Fredriksson [Figueroa and Fredriksson, 2009], and Tellez and Chavez [Tellez and Chavez, 2010].

The index uses the inverse of the permutation to create a vector space, using the Minkowski's L_1 or L_2 distances (i.e. Spearman Footrule and Spearman-ρ, respectively) [Chavez et al., 2008].[1]). Please note that each permutation requires $|R|$ distances, a linear sort $O(|R|)$,[2] and a linear pass to find the inverse. In order to find the candidate list, one needs to perform n permutation distance comparisons (L_1 or L_2) each one of them requires $O(|R|)$ basic arithmetic operations. This yields to a total cost of $O(n|R|)$ basic operations. The space complexity is $n|R| \log |R|$ bits. The final number of candidates is specified with the querying parameter γ.

The Brief Permutation Index

Another algorithmic solution to compute an approximate distance between permutations is the *Brief Permutation Index* (BPI), introduced by Tellez et al. [Tellez et al., 2009]. The main idea is to encode the permutation vectors using fixed-size bit-strings and compare them using the Hamming distance. This produces a smaller representation which can be filtered out faster than the original permutations space. Nevertheless the set of candidate objects after filtering with the brief version of the permutations is larger, and this is relevant for expensive distance functions. The advantage against the original algorithm is the reduction of some CPU cost and smaller requirements of memory, yielding faster searches for large databases.

The resulting Hamming space encodes each permutant with a single bit using the information about how much it deviates from the original position. If the permutant is displaced by more than m positions (which is a parameter) the corresponding bit is set to 1, else it is set to 0. The

[1] Another option is Kendall-τ [Chavez et al., 2008], but its usage is limited by the high cost of the computation

[2] In general, if distances cannot be discretized, we need a comparison based sorting, i.e. $O(|R| \log |R|)$.

number of bits then matches the number of permutants. A fair choice for m is $\frac{|R|}{2}$. One observation is that the central positions are assigned mostly 0's because the central permutants have less room for displacement. This is solved using an inline central permutation [Tellez et al., 2009].

Although computing the Hamming distance is faster than computing L_2, a sequential scan can be too high for large databases. The same authors presented later a version indexed with Locality Sensitive Hashing [Tellez and Chavez, 2010]. Unfortunately, the recall dropped as the speed increased.

Metric Inverted File

Amato and Savino [Amato and Savino, 2008] present an scalable proximity search index, based on [Chavez et al., 2008] and the simplification of the Spearman Footrule. The main idea is to use of only the first K references closer to an object and its positions in the full permutation. This information is used to compute approximately the Spearman Footrule (L_1) distance of the permutations. The memory requirement is smaller since $K \ll |R|$.

Since many permutants cannot be found to compute L_1, the blank positions should be filled with a penalization constant ω (e.g. $\omega = |R|/2$). To provide a scalable representation, the set of references is used as the *thesaurus* of an inverted file, and list of tuples $(object, position)$ as *posting lists*. Baeza-Yates and Ribeiro-Neto provide a detailed explanation about inverted indexes [Baeza-Yates and Ribeiro-Neto, 1999].

The computational cost to represent each object is equivalent to the permutation based index; however, in this case the plain mapping (without inverted index representation) requires $Kn \log(K|R|)$ bits i.e., each object is represented with K tuples $(referenceId, position)$. These tuples are sorted by $referenceId$ (adding an additional sort over K items). Using an inverted index, requires $Kn \log(Kn)$ bits of space, and the total cost is driven by the cost to obtain the candidate list plus γ distance computations.

Prefix Permutations Index: PP-Index

The last approach using the permutations idea is the PP-Index [Esuli, 2009]. It stores only the prefixes of the permutations and hints the proximity between two objects with the length of its shared prefix (if any). Longer shared prefixes hint high proximity and short length prefixes reflect low proximity. This strict notion of proximity yields very low recalls. This condition is somewhat alleviated by using several permutations sets, several indexes, and tricks like randomly perturbing the query, which end up increasing the num-

ber of queries to the index and affecting the searching speed, i.e., the main advantage of the index. The index consists in a compact trie [Baeza-Yates and Ribeiro-Neto, 1999] representing the space of permutation prefixes. A plain representation needs $Kn \log \sigma$ bits. The compact trie is usually smaller, and the storage usage depends on the amount of shared prefixes. The first levels in the trie are stored in main memory and the lower levels in secondary memory.

In order to overcome the low recall, several strategies are possible, increasing the searching time and memory usage (see [Esuli, 2009]). For example the PP-Index needs up to eight indexes and expand the query on each to achieve perfect recall on the 106 million MPEG7 vectors of the CoPhIR data set [Bolettieri et al., 2009].

2.3 Indexing Sequences

In this Thesis, we introduce several new metric indexes, the majority of them require a simple organization of the involved information. Fortunately, this simplicity allow us to implement our indexes as sequences of symbols, that can be indexed in close to optimal space, while basic operations are efficiently solved.

In this section we introduce the basic representation, notation, and operations on a sequence.

2.3.1 Statement of the Problem

There exists several Indexes of Sequences (IoS), since each one puts a different tradeoff between the necessary memory, and the complexity of its operations.

The basic problem is as follows. Let $T = s_1 s_2 \cdots s_n$ be a sequence of symbols on the alphabet Σ of size σ, i.e. $s_i \in \Sigma$. Without loose of generality, let Σ be a set of integers, that is $\Sigma = \{1, 2, \cdots, \sigma\}$. The i-th symbol in T is denoted as T_i.

An IoS provides three basic operations:

- $\mathsf{Rank}_c(T, pos)$ counts how many c's occurs in T until pos, $c \in \Sigma$.
- $\mathsf{Select}_c(T, r)$ returns the smaller position pos such that $\mathsf{Rank}_c(T, pos) = r$.
- $\mathsf{Access}(T, pos)$ retrieves the symbol stored at the position pos in T, i.e., T_{pos}.

Notice that an IoS replaces T, since we can reconstruct it using $\mathsf{Access}(T, pos)$, but our notation requires to put T in the arguments even when it is not actually stored.

There exists several indexes solving these operations efficiently. For example, for binary alphabets, it is possible to solve all operations in constant time using $n + o(n)$ bits, as surveyed by Navarro and Mäkinen [Navarro and Mäkinen, 2007].

Gonzalez et al. [González et al., 2005] present a fast practical approach, it solves Rank_c in $O(\log n)$ time, and stores $n + o(n)$ bits. This is one of the fastest bitmap indexes.

Larger alphabets are solved reducing the problem to the binary case. Grossi et al. [Grossi et al., 2003] introduce the Wavelet Tree (WT), it uses $n \log \sigma + O(\sigma \log n)$ bits solving all operations in $O(\log \sigma)$ time. Very large alphabets are problematic with this scheme since all times are dependent on σ. Golinsky et al. [Golynski et al., 2006] introduce a fast index, robust to large σ. It uses $n \log \sigma + o(n \log \sigma)$ bits, it solves Select_c in constant time, and both Rank_c and Access on $O(\log \log \sigma)$ time. Claude and Navarro [Claude and Navarro, 2008] implement this index applying practical decisions, this implementation solves Rank_c and Access on $O(\log \sigma)$ time.

2.3.2 Storage Requirements

Let n_c be the number of symbols c in T, then we require at least

$$\log \binom{n}{n_1, n_2, \cdots, n_\sigma} = \log \frac{n!}{n_1! n_2! \cdots n_\sigma!} \text{ bits}$$

to represent any instance of T with these statistics. From information theory we can obtain the following formulation, using a fixed code word for each symbol, we require at least $nH_0(T) \leq n \log \sigma$ bits, here, $H_0(T)$ is the order zero empirical entropy of T, i.e.

$$nH_0(T) = n \sum_{c \in \Sigma} p_c \log \frac{1}{p_c} = \sum_{c \in \Sigma} n_c \log \frac{n_c}{n} \text{ bits}$$

Where p_c is the probability of occurrence of c in T, empirically, $p_c = n_c/n$.

There exists several index achieving close to this optimal space for binary alphabets. For example, Raman et al. [Raman et al., 2002], its practical implementation by Claude and Navarro [Claude and Navarro, 2008], Okanohara and Sadakane [Okanohara and Sadakane, 2007], and our bitmaps based on differences (Appendix A). For $\sigma > 2$ there exists several variants

of the Wavelet Tree (WT) by Grossi et al. [Grossi et al., 2003]. For example, the WT with Huffman shape or with internal bitmaps compressed to nH_0, like surveyed by Navarro and Mäkinen [Navarro and Mäkinen, 2007].

2.3.3 Final Notes

The objective of this work is to produce fast and small proximity searching indexes. However, understanding techniques behind indexing sequences is required because they are part of our indexes, and new representations. A review of the basic techniques on indexed bitmaps and sequences is given at Appendix A. Also, the appendix introduces new structures taking advantage of the access patterns of our techniques, and in general, they introduce a set of powerful indexes for large alphabets.

Part II

Exact Proximity Searching

Chapter 3

The Reverse Nearest Neighbor List of Clusters

One of the most efficient index for similarity search is the so called *List of Clusters* detailed in Section 2.2.2, introduced by Chavez and Navarro [Chávez and Navarro, 2005]. This data structure has a counterintuitive construction that can be seen as an extremely unbalanced tree, contrasting balanced data structures for exact searching. In practice, there is no better alternative for exact indexing, when every search return all the incumbent results; as opposed to approximate similarity search. The major drawback of the list of clusters is its quadratic time construction.

In this chapter we revisit the List of Clusters (LC) aiming at speeding up the construction time without sacrificing its efficiency. With this improvement we obtain the same storage cost with similar searching times while gaining a significant amount of time in the construction phase.

3.1 Introduction

In general, there is a gradation of the different complexities of the data. From a practical point of view, we can classify the indexes as effective in a region of the complexity spectrum. A long standing index, with asymptotic optimal performance (counting computed distances), is AESA [Vidal Ruiz, 1986; Micó et al., 1994] which can be seen as a pivot-based index using all the database objects as pivots. This optimal performance can be of use in relatively small databases because of a quadratic dependance on the size of the database. As a rule of thumb, in the pivot based indexes with linear space usage such as LAESA [Micó et al., 1994], one can trade speed at query

method	preprocessing distances	searching distances	memory
List of clusters (LC)	$O(n^2)$	$O(n^\beta)$	$O(n \log n + m f_d)$ bits
AESA	$O(n^2)$	$O(1)$	$O(n^2 f_d)$ bits
Linear AESA (LAESA)	$O(n\ell)$	$O(n^\beta)$	$O(n\ell f_d)$ bits

Table 3.1: Complexities of the faster proximity searching algorithms for a fixed dimensionality dataset.

time for the size of the index. One way to obtain a good speed/space tradeoff is by using a compact index such as the Fixed Queries Array (FQA) [Chávez et al., 2001], or the Fixed Queries Trie [Chávez and Figueroa, 2004].

Another way to cope with the space usage is the technique of the List of Clusters [Chávez and Navarro, 2005], which is faster than LAESA or the FQA for any practical space bounds, specially when the data is high dimensional. Table 3.1 compares complexities among search-efficient indexes, m the number of centers in the LC (following $m < n$), ℓ the number of pivots used by LAESA, and β is a value between 0 and 1, and depends on the intrinsic dimension of the database. Summarizing, indexes that allow fast searches are highly expensive at the preprocessing step and/or in memory requirements.

In this chapter we propose a new index for metric searching allowing fast searches, using $O(n^\beta)$ distances per query, and $O(n \log n + m f_d)$ bits of space, and a preprocessing time of $O(nm^\beta)$, with $\beta \leq 1$. Our index is inspired on the LC data structure, and use the same searching procedure hence it can be plug into applications already using the LC without modification.

The key difference on the approach is in the construction phase. The LC have a quadratic construction time which limits its usage on large databases, probably in the same way AESA cannot be used in practice. With our proposal it is possible to index large databases. Furthermore our index can be built in parallel, making efficient use of modern hardware. Bottom line our approach is faster to build than the original LC with a very small penalty in the searching complexity, which makes the index usable for complex search pattern analysis and data mining in very large datasets.

Here we introduce a simple and effective metric index, the *Reverse Nearest Neighbor* list of clusters (**Rev-LC**). The preprocessing time of Rev-LC is way smaller than the LC in most cases. The central idea is to select the centers beforehand (say m of them) instead of obtaining them in the recursive

construction of the list.

Our construction time is in worst case $O(nm)$, but it is likely to achieve $O(nm^{\beta})$ for some $\beta \leq 1$.

Contrasting with the LC, Section 2.2.2, all the centers are selected beforehand, once certain order is established, the population of balls around each center can be done in parallel. This particular feature make the index suitable for taking advantage of modern hardware. We conducted a thorough experimentation to demonstrate the efficiency of the new index.

3.2 The Reverse Nearest Neighbor List of Clusters

Algorithm 2 works with *any* partition of the database. It should be clear that if we have an arbitrary partition in the mathematical sense, $S = \cup I_i$ and $I_i \cap I_j = \emptyset$, then querying each I_i is equivalent to searching the entire database S. The key is to avoid buckets not being relevant to the query. As pointed out by Chavez et al. [Chávez et al., 2001], this schema fits most metric indexes, since the majority of them differ on the application of the rule that discard partition elements.

If we want to improve the $O(n^2)$ construction complexity of the LC then we need to examine the origin of the problem. Please notice that in the LC, illustrated in Figure 2.1, the next center is chosen from the set of unassigned objects and that the tail recursion is responsible for the quadratic behavior.

A solution is to choose m centers beforehand and define a Dirichlet domain, just as if it where a GNAT (Brin [Brin, 1995]) of one level with very large arity. For convenience, we unzip LC's triplets into its three components, i.e. the centers $C = c_1, c_2, \cdots, c_m$, the buckets $I = I_1, I_2, \cdots, I_m$, and the covering radii $\mathsf{COV} = \mathsf{cov}(c_1), \mathsf{cov}(c_2), \cdots, \mathsf{cov}(c_m)$. In order to simplify algorithms, both I and COV are indexed with its entry number i, and with the corresponding c_i center.

The construction of Rev-LC is depicted by Algorithm 3. If we assume no ties for the nearest neighbor, an alternative succinct definition is $I_c = \{u \in \{S \setminus C\} \mid \mathsf{nn}_{S,C,d}(u) = c\}$, and $\mathsf{cov}(c) = \max\{d(c,u) \mid u \in I_c\}$, i.e., buckets are populated with the reverse neighbors of each $c \in C$.

The searching procedure is very similar to the original LC since the discarding rule is similar. We can apply the searching algorithm (Algorithm 2), only avoiding the last condition (line 10). This is because the Rev-LC divides S into m regions, contrary to the recursive binary division of the LC.

With the right setup, C can be chosen to be large enough to be considered a representative sample of the distribution of S, such that the average value

algorithm 3: Construction of the Rev-LC

Input: The number of centers, m.
Output: The Rev-LC index, i.e. C, I's, and COV.

```
1: C is initialized selecting m random centers from S
2: for all i = 1 to m do
3:     I_i ← ∅
4:     COV_i ← 0
5: end for
6: for all u ∈ S \ C do
7:     Let c_i to be the nearest neighbor of u in C
8:     I_i ← I_i ∪ {u}
9:     COV_i ← max {COV_i, d(c_i, u)}
10: end for
```

of $\mathsf{cov}(c)$ corresponds to a small percentile of the cumulative distribution function of the distances. Under these circumstances, we would have the same performance conditions of the LC, and hence Theorem 3.2.1 also holds.

Theorem 3.2.1 (Chavez and Navarro [Chávez and Navarro, 2005]) *The number of distance computations performed by the (Rev-)LC to solve some query is $O(n^\beta)$ for some $\beta \leq 1$.*

Notice this theorem follows if $m = O(n^\beta)$, for some $\beta < 1$, and in fact, close to 1, such that $n/m = O(n^{1-\beta})$ is quite small, and enough to produce compact buckets (in the radii sense). Nevertheless, the resulting β is larger than the exposed by the LC. Based on this theorem, we can produce the following conclusion about the construction of the Rev-LC.

Theorem 3.2.2 *The preprocessing cost of the Rev-LC is $O(nm^\beta + m^3/n)$ for some $\beta \leq 1$.*

Proof. From Algorithm 3, the preprocessing step for high intrinsic dimensional datasets requires $O(nm - m^2)$ distance computations, which is a very pessimistic assumption. For the construction we need $n - m$ nearest neighbor searches over C. We can use the Rev-LC index for this smaller set. Based on Theorem 3.2.1, we need n nearest neighbor searches in the smaller set C with cost $O(m^\beta)$. The additional $O(m^3/n)$ term comes from the construction of the Rev-LC index for C, but this time using a sequential search to retrieve the nn and using the same proportion of centers. □

Please notice that the term m^3/n should be small enough, i.e. $m^3/n < nm/2$, this implies $n/m > \sqrt{2}$. If $\gamma_{\text{bsize}} = n/m$ this can be expressed as

$n^3/(n\gamma^3_{\text{bsize}}) \ll n^2/(2\gamma_{\text{bsize}})$, yielding to the simplified formulae $n^2/\gamma^3_{\text{bsize}} \ll n^2/(2\gamma_{\text{bsize}})$.

Even if $\gamma_{\text{bsize}} = O(1)$ the cost would be small enough. For example, if $\gamma_{\text{bsize}} = 12$ (i.e. a suggested value of the bucket size for high intrinsic dimensional data sets according to [Chávez and Navarro, 2005]), we obtain that $n^2/1728 \ll n^2/24$. Thus, the $O(m^3/n)$ overhead is negligible, we can take only the significant term $O(nm^\beta)$ as the processing time for building the Rev-LC index for S.

The parallelization of the construction is straightforward, unlike the LC algorithm (to be addressed on the Chapter 4). A simple modification of the Algorithm 3 is required at line 6, here we must search the nearest neighbor in C in parallel, line 7. Finally, the rest of the lines inside the loop must be serialized. We call this version of the algorithm as Parallel Rev-LC (PRev-LC).

3.3 Experimental Results

As we commented in Section 1.3.1, the entire databases and indexes are maintained in main memory and without exploiting any parallel capabilities of the workstation, excepting for the PRev-LC. On the parallel version, the setup was left to the default configuration of the *parallel tasks* of the mono's framework.

We present an experimental comparison of our Rev-LC index against the LC, in both preprocessing and querying time. As it is customary we count the number of distances computed in each one of them to compare. We also measured the total time elapsed in the construction.

3.3.1 Construction Time

We selected a low-cost distance for testing the construction. Our choice is a four dimensional dataset of one million vectors under the L_2 distance (RVEC-4-1000000, Section 1.3.3). The results are reported in Table 3.2. The time to build the LC is more than twice larger than the Rev-LC for n/m of 1024 and 128, and 17.5 times for $n/m = 16$. The parallel algorithm (PRev-LC) have the faster preprocessing times, it runs close to 10 times faster than the LC for $n/m = 1024$, and more than 46 times faster for $n/m = 16$.

method	n/m	m	preprocessing time	
			seconds	human readable
LC	1024	976	331.13	5 min 31.13 sec.
LC	128	7812	2056.5	34 min 16.52 sec.
LC	16	62500	16163.16	4 hours 29 min.
Rev-LC	1024	976	168.65	2 min 48.65 sec.
Rev-LC	128	7812	895.41	14 min 55.41 sec.
Rev-LC	16	62500	920.61	15 min 20.61 sec.
PRev-LC	1024	976	35.50	35.50 sec.
PRev-LC	128	7812	327.57	5 min 27.57 sec.
PRev-LC	16	62500	348.11	5 min 48.11 sec.

Table 3.2: Construction time for random vectors of dimension 4 and $n = 10^6$.

3.3.2 Searching Performance

In Figures 3.1 and 3.1, the number of distance computations to retrieve the nearest neighbor in different intrinsic dimensional datasets is shown. The LC have fixed bucket size while the Rev-LC has a fixed number of centers, m. Each plot title is composed of the bucket size (or the expected bucket size in the case of Rev-LC), and its number of centers (in this order), e.g. "16-62500".

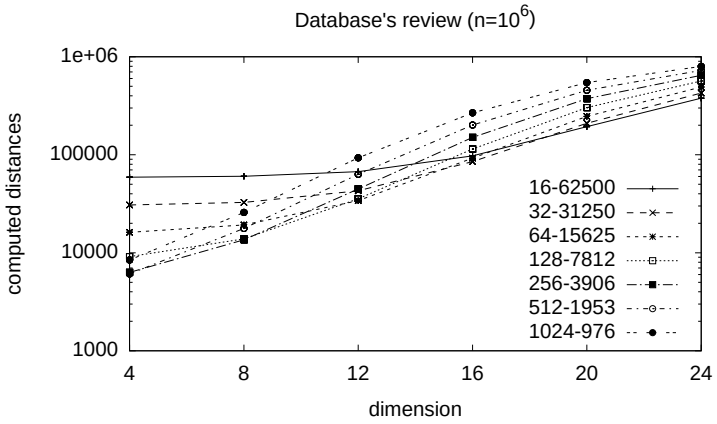
From Figures 3.1 and 3.1, we learn that there is a small penalty in the Rev-LC when compared to the LC. The number of distances is shown in Figures 3.1b and 3.1d, for LC and Rev-LC respectively. We must notice that the behavior of the Rev-LC is more faithful to the LC for larger m (smaller n/m values), this is an effect of Theorem 3.2.1. Furthermore, the real time behavior is more tight than the cost driven on counting distance computations. Please remember that Rev-LC never reaches the performance of the LC, however, this can be improved by increasing the number of centers, and hence we can make the difference as small as desired. In addition, the faster preprocessing step makes the Rev-LC and PRev-LC a competitive real option for large datasets with high intrinsic dimensions.

3.4 Summary

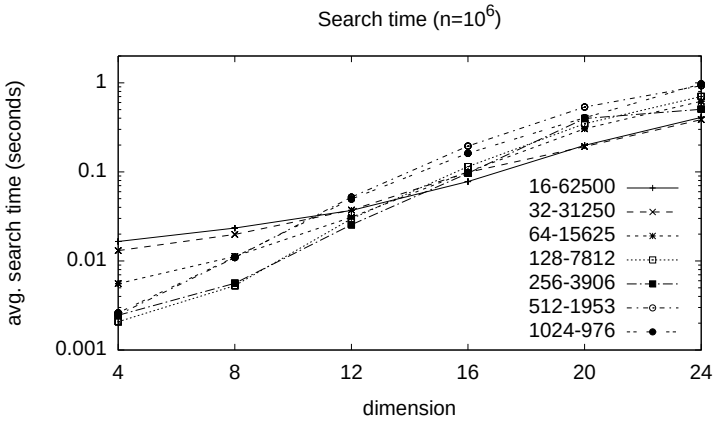
In this chapter, we introduced a new index for general metric spaces. This index uses a searching algorithm quite similar to the well known List of Clusters. The key difference is a better construction time, and the trivial parallelization of the preprocessing step. This last feature makes the index a good option to run on modern hardware.

A dynamic index supporting insertions and deletions, seems to be simpler to maintain for the Rev-LC than for the LC. More detailed, Rev-LC only needs to locate items into its nearest center, while LC needs to maintain a set of n/m nearest neighbors of each center. The dynamic Rev-LC promises to be useful even on highly transactional environments. However, dynamic operations over metric indexes are beyond the scope of this work. There is no doubt that more work it is necessary to take full advantage of the Rev-LC and is properties.

Even with this enhancements over the LC, m should be really large to achieve a similar performance to LC. However, the universe of Rev-LC indexes over a fixed database with n objects and selecting m centers is quite smaller than the universe of LC indexes, with the same n and m parameters. This is a core advantage of the LC that will be exploited in our main contribution to the exact metric indexes 5. Nevertheless, Rev-LC remains as a good option by itself, specially on large and high-intrinsic dimension datasets. Also, the Rev-LC show us how to break the rule that serializes the searching step of the List of Cluster, i.e., line 9, Algorithm 2. Using this basis, the next chapter is dedicated to parallelize both searching and preprocessing algorithms of the List of Clusters.

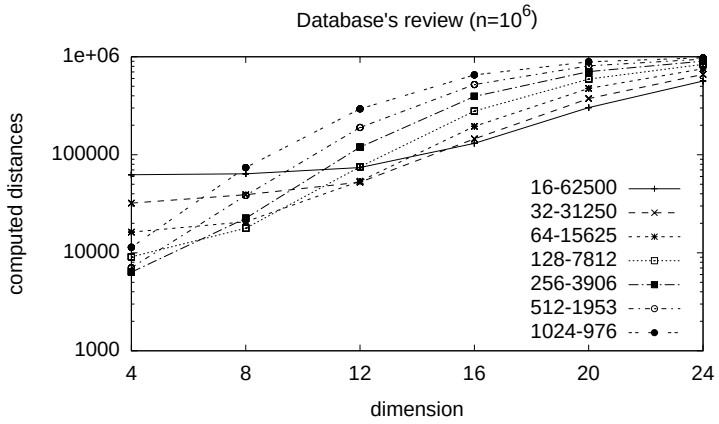


(a) LC's review of the database

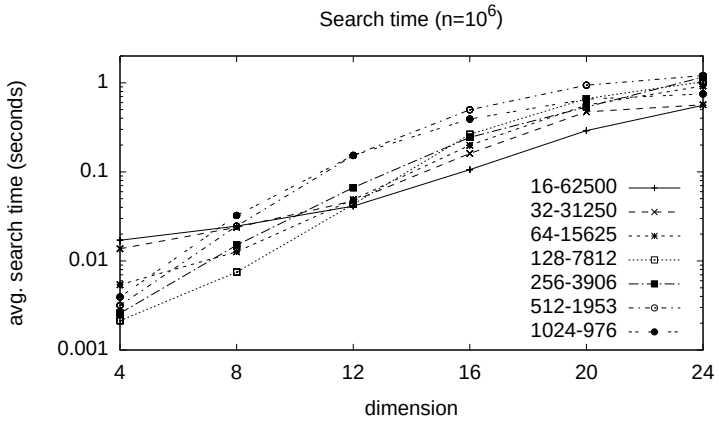


(b) LC's search time

Figure 3.1: Performance of the LC for the nearest neighbor search for increasing intrinsic dimension.



(c) Rev-LC's review of the database



(d) Rev-LC's search time

Figure 3.1: Performance of Rev-LC for the nearest neighbor search for increasing intrinsic dimension.

Chapter 4

Parallelizing the List of Clusters

With excellent searching times, and $O(n)$ identifiers stored by its data structure, the main obstacle limiting the practical usage of the List of Clusters (LC) is its expensive preprocessing step, i.e. $O(n^2)$ distance computations for high intrinsic dimensions.

Here, we apply the learned lessons of the previous chapter, parallelizing both construction and searching algorithms of the LC. We obtained a both faster constructions and searching procedures.

We already introduce the Rev-LC, a new metric index with fast preprocessing. The Rev-LC is very fast to construct, also it is trivially parallelized. Even with these advantages, the Rev-LC never reaches the searching performance of the LC, since the searching time is similar only when the number of centers is very large. Remarkably, when the number of centers is quite large, $n/m = O(1)$, matches the requirements of the LC on high intrinsic dimensional datasets. Also, Rev-LC can handle large databases, a domain not reached by LC. The reason of this scaling properties are that the set of centers of Rev-LC are selected and indexed at the beginning of the preprocessing step. Nevertheless, its behavior in not so high intrinsic dimensions is quite bad, without mention that never surpasses the LC's performance. Also, the number of possible instances of Rev-LC for a given database of size n, and m centers, is smaller than for LC.

In this chapter we speed up both preprocessing and searching algorithms of the LC, taking advantage of the new multi-core hardware.

4.1 Introduction

As evinced in the former chapter, the LC algorithms can be deserialized with a small effort. We apply the techniques of Rev-LC and PRev-LC to parallelize the preprocessing algorithm of the LC. The preprocessing step is conveniently simple, and not modifies the conceptual process of the LC metric index. On the other hand, the searching algorithm destroys the idea of an unbalanced binary tree (modifying the searching algorithm). So, conceptually, the structure is being replaced by a Voronoi partition [Chávez et al., 2001], as for Rev-LC. These changes are carried out just replacing parts of the sequential preprocessing (and searching) algorithms by code blocks with parallel execution. Our suppositions are abstractions of real world systems, i.e., we suppose that parallel blocks are executed in multiple threads with shared memory, with exclusive write and multiple readers as memory access.

About our algorithm's pseudo-coding style. The syntax of our pseudo-code uses *locks* (that can be acquired and released) defining regions that must be processed sequentially (only one thread can access at the same time). Please note that locks work on a shared resource, not in regions of code. For example, two threads working on the same code region may not be serialized if the threads are working on different resources.

4.2 Parallel Preprocessing algorithm

Algorithm 4 shows a simple parallelization of the range searching algorithm. An exhaustive k-nn algorithm is quite similar but storing only k nearest items in Q_h collections of Algorithm 4, and the last union stores only the k nearest items to q (i.e. $k = n/m$).

4.3 Parallel Searching algorithm

An elaborate range searching algorithm is presented in alg. 5. It uses the LC structure to avoid unnecessary distance computations. A simple modification to the original algorithm is applied, we remove the last condition (line 10 of Algorithm 2), i.e. the second condition of Figure 2.2 is not considered. This condition serializes the searching process. This small change barely increases the number of distance computations (since it occurs with very low probability).

algorithm 4: Exhaustive parallel range searching

Input: A set of objects S, a query $(q, r)_d$, and t as the number of threads to process the database.
Output: The set of objects intersecting the ball of radius r centered on q.

```
1: Let Q[1, t] be a collection of empty sets
2: begin parallel execution
3: for all u ∈ S do
4:    Let h be a random number between 1 and t.
5:    if d(c, u) ≤ r then
6:       adquire lock on Q_h
7:       Q_h ← Q_h ∪ {u}
8:       release lock on Q_h
9:    end if
10: end for
11: end parallel execution (wait all threads to commit)
12: return ⋃_{1≤h≤t} Q_h
```

4.3.1 Parallel k-nn Searching algorithm

The k-nn searching algorithm is shown in alg. 6. It is more complex than the parallelized range searching algorithm. The algorithm is composed of two parallelized regions, the first one is dedicated to select related centers, and bound r^*. The second one uses the information collected at the first step to determine buckets that will be reviewed. As in the parallelized range searching, we remove the bucket contention case of the original searching algorithm.

4.4 Experimental Results

Our parallelized algorithms have a high dependency on the distance cost, and low-cost distances are not showing important speed up factors, thus we omit a deeper study on this configurations, simply remarking the low performance exposed on this datasets. Most of real datasets contains many times more attributes than the strictly necessaries to represent the same information. So, distances of real world datasets are costly, compared to synthetic datasets generated uniformly. In order to give a real description of the behavior of our techniques, we select two real world databases (Colors and CoPhIR-1M, described in section 1.3.3), as detailed below. It is worth noticing that even if our datasets are vector spaces, we are not using the coordinates to discard elements. We use the distance as a black box. This

algorithm 5: Parallel range searching on the LC

Input: A set of objects S, the list of clusters L with m centers, a query $(q, r)_d$, and t as the number of threads to process the database.
Output: The set of objects that satisfies $(q, r)_d$.

```
1: Let Q[1, t] be a collection of empty sets
2: begin parallel execution
3: for all (c, cov(c), I) ∈ L (serialized access) do
4:    Let h be a random number between 1 and t
5:    if d(q, c) ≤ r then
6:       adquire lock on Q_h
7:       Q_h ← Q_h ∪ {c}
8:       release lock on Q_h
9:    end if
10:   if d(q, c) ≤ r + cov(c) then
11:      for all u ∈ I do
12:         if d(q, u) ≤ r then
13:            adquire lock on Q_h
14:            Q_h ← Q_h ∪ {u}
15:            release lock on Q_h
16:         end if
17:      end for
18:   end if
19: end for
20: end parallel execution
21: return ⋃_{1≤h≤t} Q_h
```

allows to work with the data disregarding its representation, all we need is a distance function to index the data.

All experiments were performed in a workstation with Intel(R) Xeon(R) CPU E5462 @ 2.80GHz, with eight cores (two quad-core processors), and 2GiB of main memory. The workstation runs the 9.8.0 Darwin Kernel. All indexes and databases were stored in main memory. Our implementations was written in the C# programming language and on the Mono (`www.mono-project.com`) framework.

Figure 4.1 shows the behavior of our parallel LC preprocessing step, for several fixed bucket sizes (n/m). The left column presents results for the Colors database, and the CoPhIR-1M result set is located on the right one. The first row, the total construction time, is clearly improved using our parallelized construction. The speed up varies directly with n, and the cost of the distance function; and inversely with m. As n/m decreases (m grows on a fixed n) the speed up for our indexes increase (Figures 4.1c and 4.1d, for

Colors and CoPhIR-1M). This example illustrates the actual limits of the LC on real world applications. The sequential implementation is quite costly, since large high cost distances and large databases like occurs for CoPhIR-1M. On these cases, our parallel techniques increase the chances that the LC can be applied to large and high dimensional datasets. As we can observe on the last row of Figure 4.1, our partitioning and joining policies have a higher performance on large m and n values. The efficiency per core (the ratio between the ideal performance over the real obtained) ranges from 15% to 42% for Colors, and 43% to 71% for CoPhIR-1M. This difference is mainly due to the cost of the distance function (208 coordinates for CoPhIR-1M vs 112 coordinates for Colors).

Figure 4.2 compares the performance of the parallel algorithm with respect to the plain implementation. On the left side of Figure 4.2, the performance of the LC parallel range search on Colors database. The speed up reaches up to 4 times the serial version, holding a core efficiency close to 50%. On the right side of Figure 4.2 the performance of the parallel LC range searching algorithm on CoPhIR-1M benchmark is studied. Here the speed up is much better, since it achieves close to 85% of efficiency (an speed up of close to 7, using 8 cores). The query time is drastically reduced as shown in Figure 4.2b.

The performance of our parallel k-nn searching procedure is shown in Figure 4.3. As expected, the speed up is higher on expensive distances, and large m. Nevertheless, the efficiency of the parallelized algorithm is lower than both preprocessing and range searching algorithms. We state that this reduction in performance is a consequence of the need of global shared information of R (since the final r^* is unknown beforehand), Algorithm 6.

4.5 Summary

This chapter introduces a parallelization of the well known List of Clusters. Up to the best of our knowledge, this is the first attempt to reduce the preprocessing time of the LC using parallel techniques. We presented parallel algorithms for the preprocessing step, range searching, and k-nn searching. Our research objective is to bring support for high intrinsic dimensional datasets and large databases.

In the construction step, notice that complex configurations achieve higher speed ups. For example, CoPhIR-1M performances are better than those obtained with Colors, right side and left side respectively of Figure 4.3. This behavior is consequence of the higher distance cost (208 vs 112 coor-

dinates), large m, and large n values. Also, we presented a parallel range search algorithm for the LC. Achieving peak efficiencies (peer core) going from 50% to 85%, for our testing databases. We remark that this difference comes from the variations on the size of the database, and the cost of the distance function.

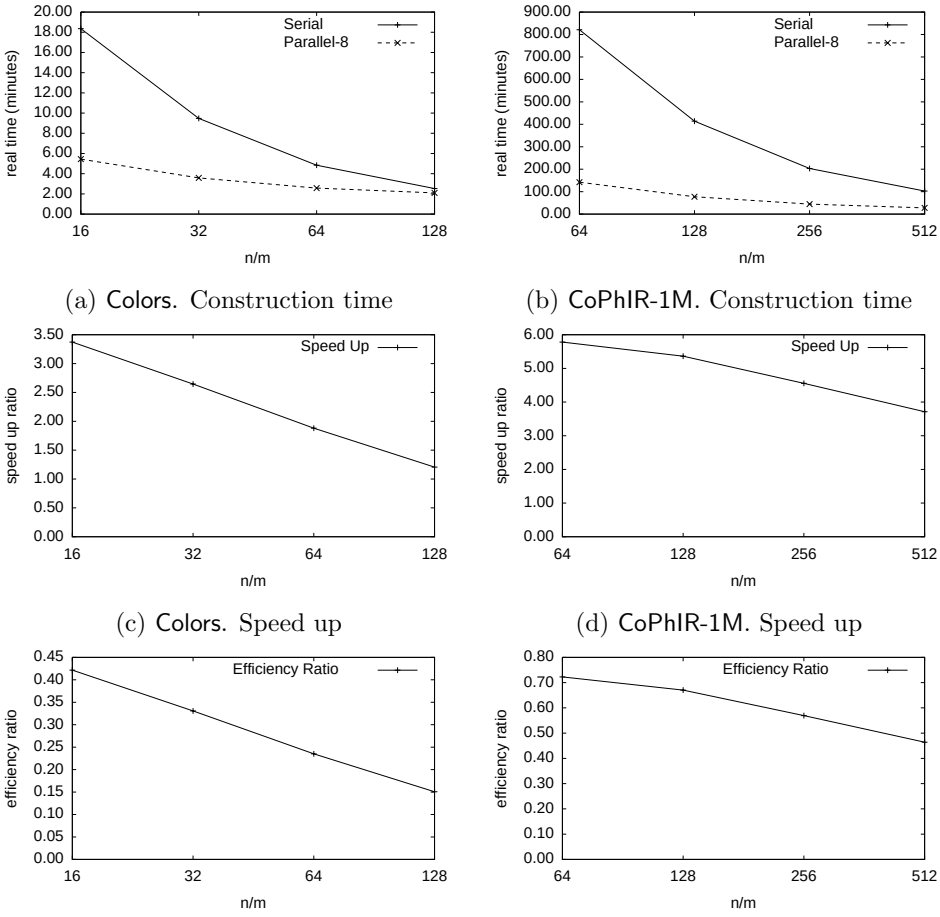
On the downside, our k-nn searching algorithm has a lower performance than the serial version, the probable reason is the global lock needed to maintain the bounds of the searching radius. There is not doubt that more work is required to improve partitioning and joining (i.e. locking) policies for our algorithms. The large performing differences in our datasets show that our locking policies are expensive (for example the bad behavior of our algorithms on low-cost distances). For our immediate purposes, the speed ups of this chapter are enough. Yet, these costs should be reduced, however, a deeper study of parallel techniques is beyond the scope of this work.

In the following chapter, we introduce a new metric index dubbed as Polyphasic Metric Index (PMI). PMI uses a set of Lists of Clusters metric indexes as building blocks. This is the reason of the emphasis on improving LC's performance, allowing to the PMI index to be of use in practice.

algorithm 6: Parallel k-nn searching on the LC

Input: A set of objects S, the list of clusters L with m centers, a query $\text{k-nn}_d(q, S)$, and t as the number of threads to process the database.
Output: R as the k-nn set.

1: Let $M[1, m]$ be a collection of tuples $(d(q, c), c, \text{cov}(c), I)$
2: Let $Q[1, t]$ be a collection of empty sets
3: **begin** parallel execution
4: **for all** $i = 1$ to m (serialized access) **do**
5: $M[i] = (d(q, c), c, \text{cov}(c), I_i)$
6: **end for**
7: **end** parallel execution (wait for all threads to commit)
8: Sort M in ascending order using the first entry in the tuple as key
9: Initialize R as the first k centers in M (second entry in tuples).
10: Define $r^* = \max_{u \in R} d(q, u)$, it dynamically follows R. {It is implemented accessing the first entry in $M[k]$}
11: **begin** parallel execution
12: **for all** $(d(q, c), c, \text{cov}(c), I) \in M$ (serialized access) **do**
13: **if** $d(q, c) \leq r^* + \text{cov}(c)$ **then**
14: **for all** $u \in I$ **do**
15: **if** $d(q, u) \leq r^*$ **then**
16: **adquire** lock on R
17: $R \leftarrow R \cup \{u\}$
18: remove the farthest item on R
19: update r^*
20: **release** lock on R
21: **end if**
22: **end for**
23: **end if**
24: **end for**
25: **end** parallel execution (wait for all threads to commit)
26: **return** R



(a) Colors. Construction time

(b) CoPhIR-1M. Construction time

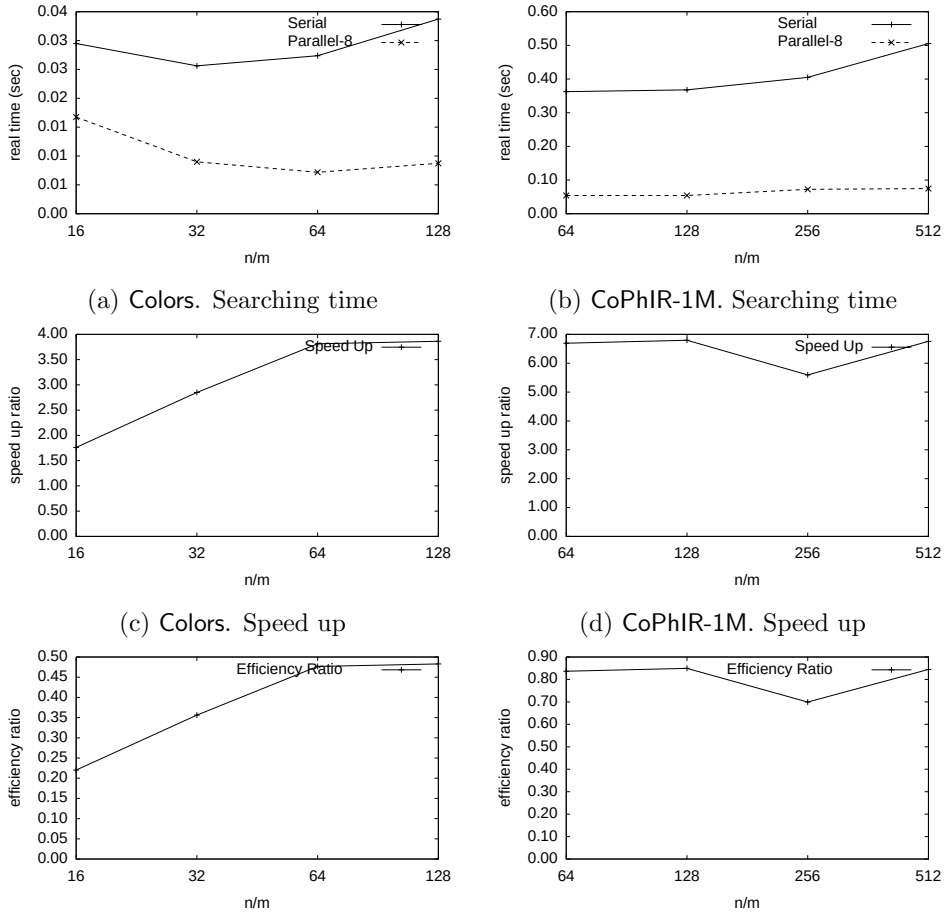
(c) Colors. Speed up

(d) CoPhIR-1M. Speed up

(e) Colors. Core efficiency ratio

(f) CoPhIR-1M. Core efficiency ratio

Figure 4.1: Performance of the parallel preprocessing of the List of Clusters for our real world benchmarks.



(a) Colors. Searching time

(b) CoPhIR-1M. Searching time

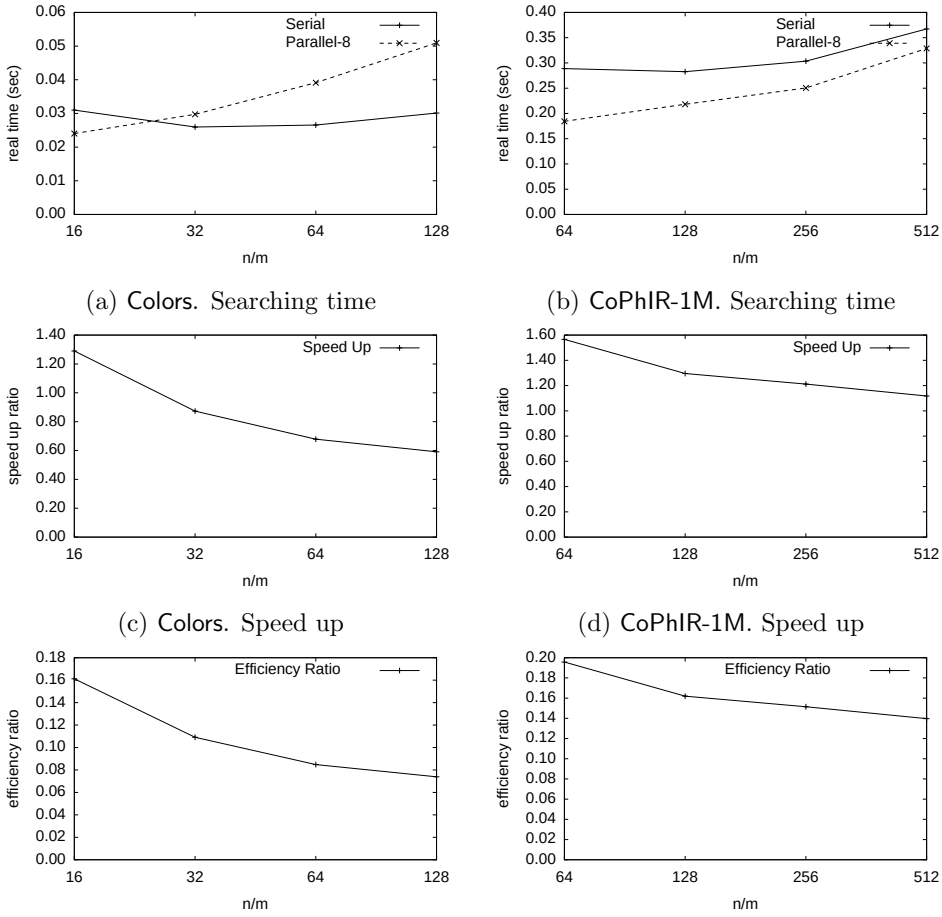
(c) Colors. Speed up

(d) CoPhIR-1M. Speed up

(e) Colors. Core efficiency ratio

(f) CoPhIR-1M. Core efficiency ratio

Figure 4.2: Performance of the parallel range searching algorithm. Colors searches a radius recovering 0.02% of the database, and CoPhIR-1M recovers 0.01% of the database.



(a) Colors. Searching time

(b) CoPhIR-1M. Searching time

(c) Colors. Speed up

(d) CoPhIR-1M. Speed up

(e) Colors. Core efficiency ratio

(f) CoPhIR-1M. Core efficiency ratio

Figure 4.3: Performance of the parallel k-nn searching algorithm

Chapter 5

Polyphasic Metric Index

5.1 Introduction

Proximity searching is a challenging problem since exact indexes (those returning exactly the objects satisfying a query) have a linear worst case on the size of the database, even when the query output set has $O(1)$ size. This behavior was described in Chapter 1.

In this chapter we introduce a new metric index very robust to the intrinsic dimension growth, with very good tradeoffs among memory, real search time, and number of computed distances.

In general, a metric index can be regarded as a partition of the space. The index then guides the search by filtering some partition elements for each particular query. The unfiltered elements are then exhaustively checked. Our algorithmic idea is to use several indexes, several partitions, applying the corresponding filters and then search in the intersection of all the non-filtered partition elements. One key aspect of the above idea is to efficiently implement union/intersection operations to quickly obtain the answer.

We propose novel algorithms for proximity searching, based on fast union-intersection operations. Specifically, we introduce algorithms to solve range and k-nn searches (Section 2.1). Our algorithm is optimal in the same sense given by Samet et al. [Samet, 2006], where the cost of searching for the k-nn is the same as the cost of a range search with the proper searching radius.

Since our index is composed of several underlaying indexes, a basic requirement should be to build on the better brick. We have to select the better index, appropriate for intrinsically high dimensional data. Unfortunately the options are scarce. The most robust indexes are expensive either

in memory usage or preprocessing time (or both), as detailed below:

- AESA [Vidal Ruiz, 1986] need to store $O(n^2)$ distances. The cost of construction is of the same order. Moreover, it requires a quadratic number of arithmetic and logical operations at query time. If we count the number of computed distances, the searching cost is $O(1)$ but with an exponential dependency on the intrinsic dimensionality.

- The List of Clusters (LC) [Chávez and Navarro, 2005], use $O(n)$ integers for the index, and $O(m)$ distance values, with m the number of centers. In general, it requires roughly $mn/2$ evaluations of the distance, $O(mn)$. Nevertheless, as explained by Chavez and Navarro [Chávez and Navarro, 2005], high intrinsic dimensional datasets require $n/m = O(1)$ to be useful. Hence the LC needs $O(n^2)$ distances on the preprocessing step. Its searching time is $O(n^\alpha)$, for some $\alpha \leq 1$ dependent of the database.

The above costs are even worst for our case, since we need to create several metric indexes. In practice, this restrict us to low-cost indexes, in both space and preprocessing time. Neither AESA nor the LC are suitable for the task for the prohibitive construction time, also, the storage costs definitely removes AESA as a possible backend.

Thus, we select LC as our building block, due to its speed and linear number of identifiers, yet not using its optimal configuration $n/m = O(1)$, we focus on larger values, i.e., like $m = o(n)$. We call this non optimal construction of the LC as LC$^-$ to clarify the context. Furthermore, we randomize the construction algorithm of the LC to provide *different points of view* of the database. All these points will be cleared below. The storage cost is already linear. We also provide a probabilistic model on the searching performance of our index, and this is experimentally verified.

Summarizing our contribution, we obtain a powerful metric index with $O(n^{1+\beta})$ preprocessing time, $O(n^\alpha)$ searching time, for $\alpha \leq 1$ and $\beta < 1$. Also, our index requires $O(n)$ identifiers for a fixed dataset, or $O(\lambda n)$ identifiers ($\lambda \geq 1$) if the intrinsic dimension of the dataset is taken into account.

One final remark is that our algorithmic proposal is general enough to support any mixture of indexes, beyond our proposed modifications of the LC. Moreover, the index may not be based of the triangle inequality, as will be clear later.

5.2 The Polyphasic Metric Index (PMI)

Let Λ be a collection of metric indexes of size $\lambda = |\Lambda|$. Each $T \in \Lambda$ will produce a partition Π_T of the database, as it is standard for metric indexes. $L_i \in \Pi_T$ denotes the i-th part of Π_T.

We must encourage that most metric indexes fall on this categorization, since all of them are based on equivalence classes as described by Chavez et al. [Chávez et al., 2001]. Nevertheless, not all of them have a well described structure that can be easily exploited. But due to its organization, the LC naturally fits the methodology, since each bucket I_{c_i} is in fact a part. The set of centers, C, can be seen as another element of the partition. The details and definition of LC were described in Section 2.2.2.

We can take advantage of the representation of a metric index based on the resulting partition to produce smaller metric indexes. The compact representation will be tackeld in the next chapter.

5.3 Range Search

Solving a radius query $(q, r)_{U,S,d}$ requires the computation of the set of candidates, $\mathcal{C}$. Then, exhaustively review $\mathcal{C}$, dropping objects not intersecting the ball radius r centered at q. $\mathcal{C}$ is the intersection of all $\mathcal{C}_T$, i.e. the candidate set of the underlying indexes $T \in \Lambda$. $\mathcal{C}_T$ is computed retrieving all partition elements not discarded by the triangle inequality, and then joining them. Algorithmically speaking, the range search is a set union-intersection algorithm. Formally, we must compute the following set operations.

$$\mathcal{C}_T = \bigcup_{L \in \mathcal{L}_{T,(q,r)}} L$$

Where $\mathcal{L}_{T,(q,r)}$ is the set of all parts in Π_T such that the triangle inequality cannot discard them. Finally, the complete candidate list is computed as

$$\mathcal{C} = \bigcap_{T \in \Lambda} \mathcal{C}_T$$

Please notice that a center can be shared by some backend indexes, specially when m is large, such that the we can duplicate distance evaluations against centers. We can select centers to be disjoint at built time. In our implementation, we decide to use a simpler and flexibler solution: we add a cache of distances per query such that a performed distance is only evaluated one time. As expected, only centers can be optimized with this cache.

Since we always need to compute the union, we cannot guarantee efficiency in the worst case, then we prioritize real time performance using a specialized union and intersection algorithm for our problem using the fact that our lists are composed of integer identifiers. Additionally, the structure of this algorithm is used to solve the k-nn searching with dynamic programming, Section 5.4.

Algorithm 7 implements a fast $\Theta(\sum_{T\in\Lambda} |\mathcal{C}_T|)$ union and intersection algorithm. On the code, the array A is explicitly stored because n is not so large in practice, i.e. a few millions at most. If the plain storage of A is not feasible, it can be easily replaced by a hash table, the complexity holds on average. The idea behind the algorithm is to perform the union of all elements of the partitions, per index, and then intersect these unions to obtain a final candidate set $\mathcal{C}$.

algorithm 7: Union-intersection algorithm

Input: $\mathcal{L}_{T,(q,r)}$ for all $T \in \Lambda$, i.e. the set of buckets that cannot be discarded using the triangle inequality.
Output: The candidate set $\mathcal{C}$.

```
1: Let A[1, n] be an array of integers initialized to zero, each item has
   ⌈log (λ − 1)⌉ bits.
2: for T ∈ Λ do
3:    for L ∈ L_{T,(q,r)} do
4:       for u ∈ L do
5:          if A[u] + 1 = λ then
6:             C ← C ∪ {u}
7:          else
8:             A[u] ← A[u] + 1
9:          end if
10:      end for
11:   end for
12: end for
```

5.4 Nearest Neighbor Search

Our algorithm is based on the *best first* strategy described in [Samet, 2006] and our union-intersection algorithm (alg. 7).

The nearest neighbor algorithm is a sequence of range searches, such that each query stretches the covering radius $r^*_\top$. Eventually the nearest neighbor will be contained in the result set. Thus, the naive solution performs many union-intersection operations, as required by the algorithm of

range search. Fortunately, a cleaver algorithm arises noticing that subsequent range searches composed in terms of the previous ones, hence we can use our union-intersection Algorithm 7 without losing the information of the previous computations.

algorithm 8: Best first nearest neighbor search

Input: A query object q.
Output: $r^*_\top = d(q, \mathsf{nn}(q))$ and $q^* = \mathsf{nn}(q)$.

1: Let $A[1, n]$ be an array of integers initialized to zero, each item has $\lceil \log(\lambda - 1) \rceil$ bits.
2: Let q^* be the best candidate at any moment of the nearest neighbor, $q^* \leftarrow$ undefined.
3: Let $r^*_\bot = 0$
4: Let $r^*_\top$ be the best guest at any moment of $d(q, \mathsf{nn}(q))$, $r^*_\top \leftarrow \infty$.
5: **while** $r^*_\bot \leq r^*_\top$ **do**
6: advance_bottom $\leftarrow$ true
7: **for** $T \in \Lambda$ **do**
8: {Inside next_best both $r^*_\top$ and q^* should be adjusted if it is necessary.}
9: $L \leftarrow$ next_best(T)
10: **for** $u \in L$ **do**
11: advance_bottom $\leftarrow$ false
12: **if** $A[u] + 1 = \lambda$ **then**
13: **if** $d(q, u) \leq r^*_\top$ **then**
14: $r^*_\top \leftarrow d(q, u)$
15: $q^* \leftarrow u$
16: **end if**
17: **else**
18: $A[u] \leftarrow A[u] + 1$
19: **end if**
20: **end for**
21: **if** advance_bottom **then**
22: Increase $r^*_\bot$ to the minimum radius such that at least another candidate (in any $T \in \Lambda$) list will be available.
23: **end if**
24: **end for**
25: **end while**

Nearest neighbor queries are solved using Algorithm 8, here there are three special variables, $r^*_\top$, $r^*_\bot$ and q^*; $r^*_\top$ is the best upper bound of the covering radius for our query at any moment, $r^*_\bot$ the best known lower bound, and q^* is the best known candidate to be $\mathsf{nn}(q)$. At the beginning, $r^*_\top = \infty$, $r^*_\bot = 0$, and $q^* =$ undefined; at the end of the procedure, $r^*_\top =$

$r^*_\perp = d(q, \mathsf{nn}(q))$,[1] $d(q, q^*) = d(q, \mathsf{nn}(q))$, and q^* is $\mathsf{nn}(q)$.

The objective of Algorithm 8 is to convert the nn search in a sequence of range searches. In each internal range search the covering radius $r^*_\top$ can be reduced, while $r^*_\perp$ is increased. The algorithm follows the constraint $r^*_\perp \leq r^*_\top$. Under the schema, the complication comes because we need to perform several times the union-intersection operations over the same parts, as required by the algorithm of range search. More detailed, let $r^*_\perp$ be decomposed in its steps in the algorithm, thus let $r^{*h}_\perp$ be h-th value of $r^*_\perp$ at the h step. Since $r^{*1}_\perp \leq r^{*2}_\perp \leq \cdots \leq r^{*s}_\perp$, after s steps, then follows that $(q, r^{*1}_\perp)_d \subseteq (q, r^{*2}_\perp)_d \subseteq \cdots \subseteq (q, r^{*s}_\perp)_d$. Fortunately, range searches can be decomposed in terms of the previous ones, hence we can use our union-intersection (Algorithm 7) since it stores in A the cardinality of the intersection of previous steps. So, while the algorithm advance on the partition elements, it can arise new information testifying that the intersection exists.

Let us define $\mathsf{next\_best}(T)$ as the procedure that returns at each call a list *not yet visited*, such that this list intersects the current query ball, i.e. $(q, r^*_\perp)$. It is necessary to remark that $\mathsf{next\_best}(T)$ adjusts $r^*_\top$ and q^* as needed. $\mathsf{next\_best}(T)$ accesses $L \in \Pi_T$ in the same order than consecutive range searches $(q, r^*_\perp)$. So, at each step $r^*_\perp$ is increased (Algorithm 8, line 22) to the minimum necessary to obtain another L.

Please notice that the efficiency of $\mathsf{next\_best}(T)$ is linked to the implementation. For example, when T is a tree, $\mathsf{next\_best}(T)$ procedure should be implemented using a stack to emulate recursive calls. Another point to be careful is that objects of the partition Π_T should be in $S \setminus \bigcup_{T \in \Lambda} T$, following that $\bigcap_{T \in \Lambda} T = \emptyset$, i.e. ensuring that the machinery used to partition is not indexed again, so, at most we review n items. The cache scheme presented in section 5.3 is a flexibler alternative solution.

Example 5.1 ($\mathsf{next\_best}(T)$ over a single pivot) *Consider a pivot $P \in S$, inducing a partition Π_P, using a discretizing function $g(d(P,u))$ for each $u \in S$.*

The first step is to localize the list which can contain q with radius zero, i.e. $|g(d(q,P)) - g(d(u,P))| \leq g(r^*_\perp) = 0$. Then, $r^*_\perp$ grows as necessary to advance and retrieve the next promising list, e.g. $g(r^*_\perp) = 1$. In the process, both $r^*_\top$ and $r^*_\perp$ are adjusted. The process is repeated until both bounds converges. Figure 5.1 depicts the advance of $\mathsf{next\_best}(T)$, with $|\Pi_P| = 8$. Please notice that this is the gross procedure to be performed by any pivot based index.

[1] Ideally, $r^*_\perp$ will stop in $r^*_\top$ it could overrun $r^*_\top$ since it advance in ranges.

algorithm 9: The next_best(T) procedure. General steps to solve next_best(T)

Initialize: Let $\mathcal{L}_{T,(q,r^*_\perp)}$ be the set of partition elements intersecting the ball $(q, r^*_\perp)$.
Input: Let $r^*_\perp \leftarrow 0$.
Output: A set containing objects intersecting the current query ball.
Procedure: At each call it proceeds as follows:

```
1: if r*⊥ > r*⊤ then
2:    return ∅
3: else
4:    if r*⊥ was incremented then
5:       Retrieve the necessary lists to complete L_{T,(q,r*⊥)}
6:    end if
7:    if L_{T,(q,r*⊥)} has not visited lists then
8:       Let L be a not visited lists from L_{T,(q,r*⊥)}
9:       Mark L as visited
10:      return L
11:   else
12:      return ∅
13:   end if
14: end if
```

Note 1: At any moment, if it is possible (e.g. at line 5), $r^*_\top$ is better bounded and the corresponding q^* must be set.

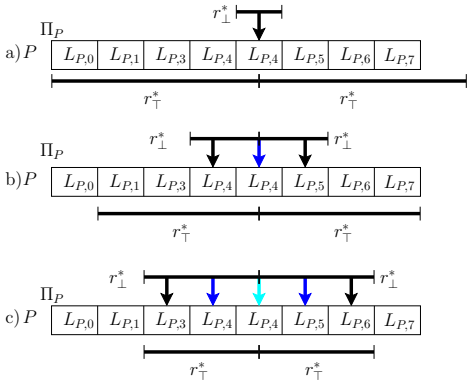


Figure 5.1: next_best on a single pivot P. $r^*_\perp$ and r^*_{top} starts being opposite, and finalizes when both converge.

Example 5.2 (next_best(T) for the LC) *We first find out the order to visit all lists. In the process $r^*_\top$ is better bounded, and give an initial value to q^*. This step consist in evaluate $d(q, c_i)$ for all $c_i \in C$, sorting c_i's in ascending order of $|d(q, c_i) - \mathsf{cov}(c_i)|$, because this is the order to be accessed while $r^*_\perp$ increases. Section 2.2.2 shows the details of searching in the LC metric index.*

Searching for k-nn is implemented converting q^* to a *max* priority queue of fixed cardinality k. In this variation, both $r^*_\perp$ and $r^*_\top$ are bounding the covering radius of the k nearest neighbor.

5.4.1 Expected Performance

The cost of solving a query, in computed distances, is linked to the expected performance in the internal indexes. A rule of thumb is that every index must provide a diverse (low correlated) set of candidates, such that $\bigcap_{T\in\Lambda} \mathcal{C}_T$ is quite close to the query answer, i.e. items not being in the result set should be discarded by at least an index in Λ.

Let $P_{T,u}$ be the probability that a random object u needs to be reviewed for some query $(q, r)_d$ in the metric index T. In the same sense, define P_u for our PMI. Considere that all $P_{T,u}$ for $T \in \Lambda$ are independent probabilities, then $P_u = \prod_{T\in\Lambda} P_{T,u}$. If each T has been built ensuring that $P_{A,u} \simeq P_{B,u}$ for all $A, B \in \Lambda$ follows that $P^\lambda_{A,u}$. This suggest that we can improve our search simply adding independent Λ indexes, arbitrarily decreasing P_u. This simplification can be seen as a probabilistic lower bound in the computing

of the probability P_u.

A more precise model will considere that probabilities are not independent. Let us concentrate on candidate sets, such that $P_u = |\bigcap_{T\in\Lambda} \mathcal{C}_T|/n$. So, the lower bound of P_u is 0, i.e. empty intersection. On the other hand, the upper bound is $P_u = \min_{T\in\Lambda} |\mathcal{C}_T|/n$. The probabilistic lower bound is found for independent probabilities under the uniform distribution. But, a more precise model may considere the dependency between objects, that can appear in several ways. For example, let A and B be LC indexes, then it is quite probable that one of the following cases arise:

- u is a center on A and $v \in I_u$, then if v is a center on B it is probable that $u \in I_v$.
- $u, v \in I_c$ for a center c on A, then if c' is a center on B, it is probable that $u, v \in I_{c'}$.
- the most common case is that a query ball intersects with centers and buckets such that the previous cases are extended to set of centers and buckets.

This model is highly dependent of both dataset and queries, such that a detailed model is very complicated. At this point, our remaining work can be summarized as an attempt to find indexes being small, with fast searching and fast preprocessing, compromising any index into an increasing of the independence of the presented probabilities. That is the main reason of revisit the list of clusters.

5.4.2 Revisiting LC

Also, we have observed that the LC searching algorithm straightforwardly transformed into k-nn searching algorithm does not perform as expected due to the last condition. The trick is not to greedily review internal items if the covering radius (which is being stretched) indicates, but advance to external ones. The buckets are reviewed at the end of the procedure, only those not discarded after the evaluation of the m centers.

As explained in Section 5.4.1, our method requires a high diversity in the partitions of the underlying indexes. Thus, we must promote this behavior. The LC's original algorithm does not specify the order to select centers, since it does not make sense for it, but since our index requires a high diversity on the order, see Section 5.4.1, the original construction does not helps for our index, thus we introduce a new randomized LC preprocessing algorithm.

We replaces the *deterministic* selection of the center by a *random* selection of $c \in S$. This modification is implemented applying Knuth's Fisher-Yates shuffle to the set of identifiers. The complexity remains untouched.

As a rule of thumb, in high dimension datasets, an optimal LC will follow that $n/m = O(1)$, that is prohibitive for most real world applications. An alternative strategy is to produce non optimal LCs (called LC$^-$), such that its preprocessing step would be cheaper, as we will experimentally show, this non optimal construction does not affect our index since at the combination of several LC$^-$ we obtain a much better index than an optimal LC.

Let $m = O(\log^b n)$ for some $b \geq 1$. Under this approach, we require close to $nm/2 = O(n \log^b n)$ computations of the distance function. If $b = 1$ then we obtain $O(n \log n)$ time, similarly to the Vantage Point Tree or Burkhard-Keller Tree [Chávez et al., 2001].

Another possible approach is to define $m = O(n^\beta)$, resulting in a preprocessing step of $O(n^{1+\beta})$, and $n/m = O(n^{1-\beta})$.

Example 5.3 (Gaining three orders of magnitude) *Let $n = 10^6$, suppose that $m = O(\log^b n)$, specifically $b = 2$, and an involved constant of 2.52, thus $m \sim 1000$. Finally, $nm/2$ becomes 5×10^8, which is much smaller than 5×10^{11}.*

The same preprocessing cost for the example's configuration is found fixing $m = n^{1/2}$, such that $n/m \simeq O(\sqrt{n})$.

Using these configurations, solving a query with a single LC$^-$ performs more distance computations than an optimal LC. But, as experimentally will be shown, our PMI reduces the number of distance computations over the optimal LC. In the following sections, we experimentally verify our claims, obtaining excellent tradeoffs among space, search time, and preprocessing time.

5.5 Experimental Results

In order to describe the performance of our index we use several datasets; some synthetic, and some generated from a real life process. Synthetic databases are used to describe the characteristics of the LC (and LC$^-$) under controlled properties of the intrinsic dimension, and the size of the databases. This datasets are randomly generated vectors in the unitary cube (Section 1.3.3) On the other side, it is well known that real world datasets differs from the randomly generated ones, since they exhibit clustering properties. All queries consist of searching for the nearest neighbor of a not indexed query object.

n/m	real time
1024	43 min
128	6 hours 5 min
16	53 hours 21 min

Table 5.1: Real time required for the construction step of the LC on CoPhIR-1M

The running hardware and operating system are the same than those described at Section 1.3. Recall that the build time is critical for the LC, since it requires $O(nm)$ distance computations, and high dimensional datasets require that $n/m = O(1)$, we got a $O(n^2)$ distance computations. Under this perspective, LC is limited to lightweight distances or small databases. Our approach is based on exploit configurations $m = o(n)$, such as $m = O(n^\beta)$ with some $0 < \beta < 1$, and take advantage of several indexes, and the diversity found in their partitions.

5.5.1 Build Time

Since we select LC$^-$ as our backend, particularly, we are interested on large n/m setups (e.g. 1024, or 128). See Table 3.2. This implies that creating several λ indexes is cheaper than create a single *optimal* LC. Furthermore, as indexes are independently created, each index can be built in a separate computer, and each computer can process a single index in parallel as presented in Chapter 4. Thus, our PMI requires λ LC$^-$ indexes, each one with a cost of roughly $mn/2$ distance evaluations, and with a high performance parallelization if the distance function is time expensive, as described in Section 4.4. Table 5.1 shows the real time spent by the construction step. We found a clear advantage on large n/m values, even multiplied by λ. So, based on the results of Section 4.4, and Figure 4.1, the parallel algorithm of Chapter 4 do not achieve its best performance because n/m is relatively large. An better solution is to construct an independent single LC$^-$ per core, using shared memory to store the database. This scheme achieves a high efficiency per core (Table 5.1), while the number of threads does not surpasses the number of available cores.

5.5.2 Searching Performance

The complexity measured as the number of computed distances is useful to estimate the performance of other distance functions (with a similar prop-

erties of the database, i.e., intrinsic dimensions and size of the database), independently of a particular distance function or the running hardware. On the other hand, the real time quantifies performance in practical applications. Thus, we are interested in both parameters.

Synthetic Datasets

Figure 5.2 shows the average number of computed distances for the nn query in *Random vectors* (RVEC) datasets. The real time cost is depicted in Figure 5.3. Notice that curves with $\lambda = 1$ are equivalent to the LC's expected performance in the specified configuration. In this serie of figures, there exists four variables:

- The bucket size (n/m). Fixed for figures in a row, we present three different values. $n/m = 16$, this is a typical value for high intrinsic dimensional datasets, a value close to the suggested value of LC ($n/m = 12$, Chavez and Navarro [Chávez and Navarro, 2005]). The other two are 128 and 1024, those values are outside of a typical value for LC, yet they are useful for the PMI for both high and low intrinsic dimensions.

- The size of the dataset (n). Three different lengths where used, (250000, 500000, and 1000000), the parameter is fixed for figures in a column.

- The number of partitions (λ). A curve per value on each figure, going from 1 to 12. Some values where ignored to preserve clarity in the figures, the behavior of missing values can be easily inferred from surrounding curves.

- Dimension of the dataset. The horizontal axis of each figure is marked with the number of coordinates on the RVEC dataset.

One of the biggest debacle on proximity search is its tight dependency on the intrinsic dimensionality. All exact metric indexes are condemned to degrade its performance as intrinsic dimensionality grows. One of the most robust indexes is the List of Clusters, being of particular interest due to its simplicity and linear number of identifiers required to be stored. Even on this scenario, our PMI surpasses the LC's searching performance, and using larger n/m is useful for smaller intrinsic dimensionality. For example, all points for dimension 24 in Figure 5.2, observe that for any dimensionality the PMI is unbeatable (considering), particularly for $n/m = 128$, where we

need to review 20% of the database (RVEC-24-1000000), compared against 38% of the best configuration achieved with the LC ($n/m = 16$).

For lower dimensions, it is natural to select large n/m, for example, the PMI requires $\lambda = 2$ and $n/m = 1024$ to review 0.3% of the database for dimension 4 and $n = 10^6$. (Figure 5.2i), while the plain LC ($\lambda = 1$, $n/m = 16$), requires to review approximately 10% of the database.

For smaller databases, RVEC-*-250000 and RVEC-*-500000, we have a similar behavior than the one exposed by indexes on RVEC-*-1000000. Nevertheless, while n decreases, the ratio of computed distances increases. This behavior is specially evident as dimension grows. The reason is the *expressivity* of vectors, that is, the density and number of regions that defined with a given intrinsic dimensionality, and the size of a database. We must notice that randomly generated datasets (with an uniform distribution) are commonly harder to index than databases generated by a real world processes. The key on this performance differences is the clustering properties exhibited by real world datasets, since the intrinsic dimensionality inside clusters is smaller.

Summarizing, a PMI with fixed (large) n/m with varying λ can be optimized for an unknown intrinsic dimensionality (as is common in real world datasets), or a query with unknown complexity. A simple way to do this, is to increase λ until the number of expected candidates does not significantly decreases between two consecutive λ values. We consider that the adaptive searching algorithm of PMI is beyond the scope of this work.

On the other side, Figure 5.3 shows the real time taken to solve nn queries. Here, the figures are not showing the same dramatic performance enhancement than those found on the reduction of the number of distance evaluations. The reason is that time performances also reflect the cost of the union-intersection algorithms (computed distances plus the complexity of the index), and some effects of the cache. Even with this time overhead the speed up induced by PMI is captured by figures with $n/m = 128$ (fig. 5.3d, 5.3e, and 5.3f), and figures with $n/m = 1024$ (fig. 5.3d, 5.3e, and 5.3f). As in review-ratio's figures, the differences are more noticeable as n grows.

Real World Datasets

Figure 5.4 shows performances for our real world databases, Colors and CoPhIR-1M. The experiment shows the dependency of the performance with n/m, i.e. the main parameter of the practical List of Clusters, and both average total (internal + external) number of distances and real time (Figures 5.4a and 5.4b) required to solve a nearest neighbor query. In this set

of figures, the left column is dedicated to show the number of distance computations, while the right one shows the real time expending to solve a nn query. This experiment varies λ and n/m, over two fixed size datasets.

The LC's index over the Colors database minimizes the number of distance computations $n/m = 32$, for larger n/m values, there exists a speed up (distance computations and real time) for all λ values, remarkable those larger than 128 and $\lambda \geq 4$. On those setups, the cost is the close the half of the best LC. Also, our PMI has a cheaper preprocessing time. On CoPhIR-1M, the performance behavior is quite similar, but we must remark that preprocessing time set an enormous difference since $n = 10^6$ and each vector contains 208 coordinates that implies a very costly preprocessing time. The LC ($\lambda = 1$) is optimized at $n/m = 128$, on larger values all setups are better than the single LC, in both number of distance computations and real time, see Figures 5.4c and 5.4d. A similar performance is found at $n/m = 64$, but we omit the point (and smaller values) since preprocessing time is much larger and it does not improves neither LC nor PMI.

5.6 Summary and Perspectives

We presented a new metric index for general metric spaces called the Polyphasic Metric Index (PMI). Our PMI is more robust to the dimension growth than the well known List of Clusters (LC), one of the most robust indexes with small memory requirements. The central idea of our index is the usage of several backend indexes, where each one respond with a set of candidates containing the exact result set of the proximity query. So, the final set of candidates is obtained by intersecting all individual sets. We choose the List of Cluster index as backend index. This selection is driven by the fact that the LC is a fast metric index, with an small memory footprint. Those properties are inherited by our index, and even when our index is composed of several LC$^-$ backend indexes, their configuration allows a very fast preprocessing time, far away from the $O(n^2)$ time required by the original version of the algorithm. So, for example, when $n = 10^6$ we obtain faster searches than the LC when $m = \sqrt{n}$ computing close to $n^{1.5}$ distances to construct its internal indexes (in both real world and synthetic databases).

Due to the composite functionality of our index, it is possible to adjust the number of indexes at searching time. Such that *hard* queries are solved with a complex machinery (several indexes), and *easy* ones with light weight ones.

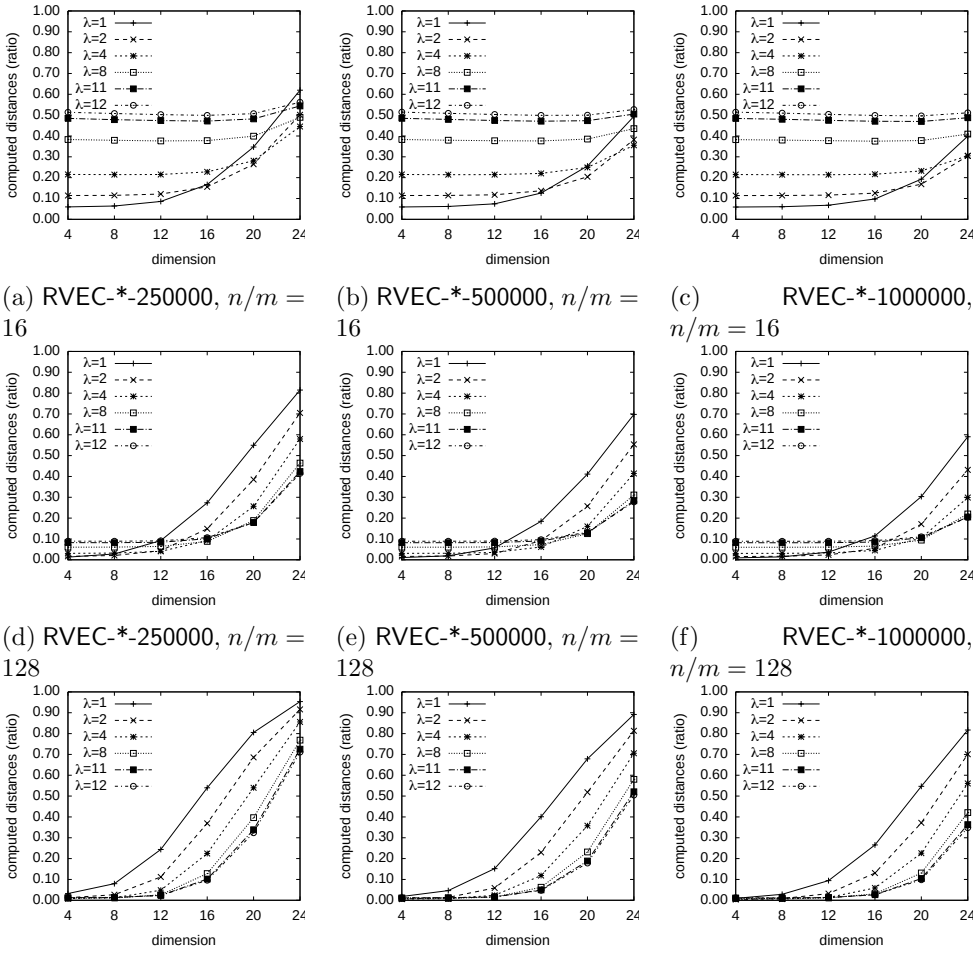
The above scheme is easily adapted to discover the better PMI configu-

ration for the (unknown) intrinsic dimension of a dataset and a query. Based in our experimental evidence, configurations with large n/m are quite good for small dimensions, and several backed indexes with this setup are useful for high intrinsic dimensions.

Based on our experimental results, we conjeture that λ is a function of the intrinsic dimension, and for a fixed intrinsic dimension, there exists a maximum λ that makes sense. Yet, our algorithms can handle a unbounded and dynamic number of indexes; such that Λ (and consequently λ) can be adjusted at query level. Nevertheless, the adaptive selection of indexes for a particular query is beyond the scope of this work, and should be explored in a dedicated work.

We presented algorithms solving range and nearest neighbor queries, which are new in its type. Both algorithms are based on set union and set intersection operations, then we present a fast union-intersection algorithm. Nevertheless, there is a place for optimization of our algorithms using better set union-intersection algorithms reducing the overhead introduced by these operations in the PMI. However they should be aware that those algoritms must support *partial intersections*, since they are the core of our dynamic programing nearest neighbor algorithm.

The biggest complication of the PMI's approach is that the required space is multiplied by λ, even when the LC$^-$ is a light weight index and λ seems to be $O(1)$ for a fixed dataset, we still concerned about the representation requirements. Trying to reduce this problem, the next chapter is dedicated to describe and test a new representation of a metric index (we work on particular with LC, but the approach is general enough to approach many other indexes). The new representation allow us to describe a lower bound on the memory requirements of the LC metric index and properly bound the PMI storage requirements. Finally, we will give an implementation reaching this lower bound without practical reduction of the searching speed.

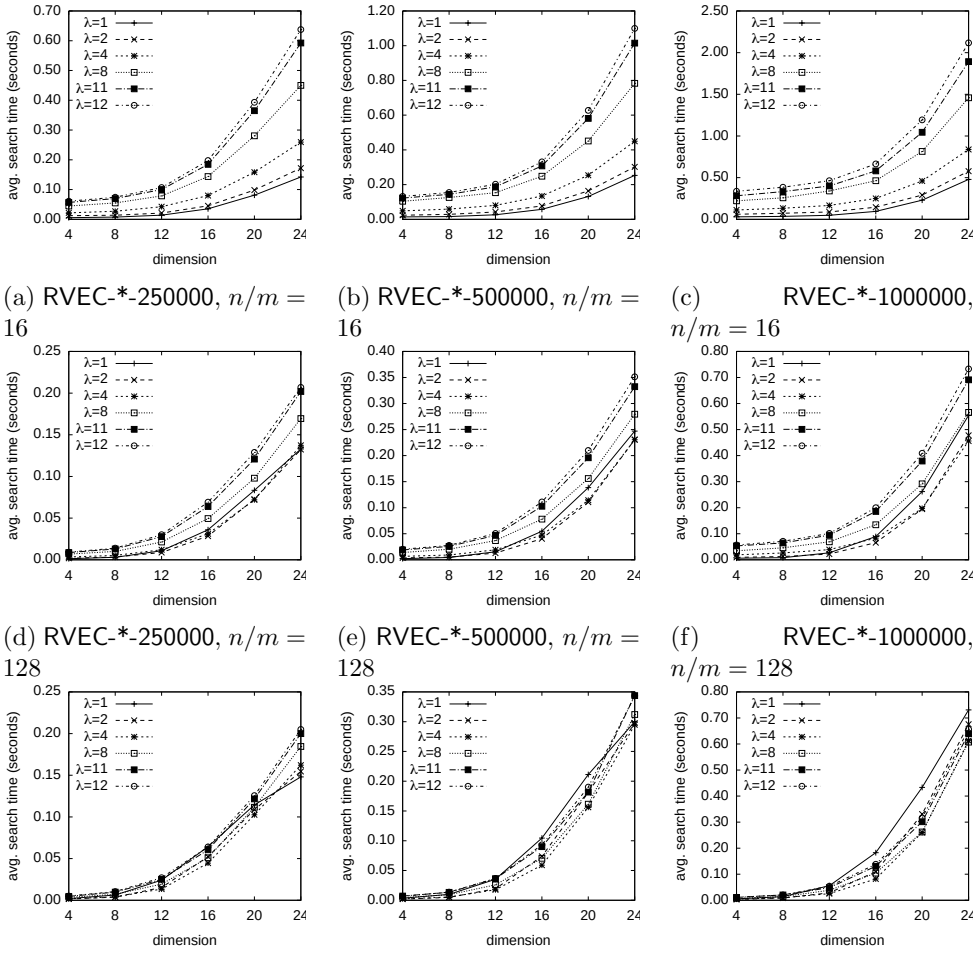


(a) RVEC-*-250000, $n/m = 16$ (b) RVEC-*-500000, $n/m = 16$ (c) RVEC-*-1000000, $n/m = 16$

(d) RVEC-*-250000, $n/m = 128$ (e) RVEC-*-500000, $n/m = 128$ (f) RVEC-*-1000000, $n/m = 128$

(g) RVEC-*-250000, $n/m = 1024$ (h) RVEC-*-500000, $n/m = 1024$ (i) RVEC-*-1000000, $n/m = 1024$

Figure 5.2: PMI's review of the database, searching for the nearest neighbor with an increasing intrinsic dimension and several n.

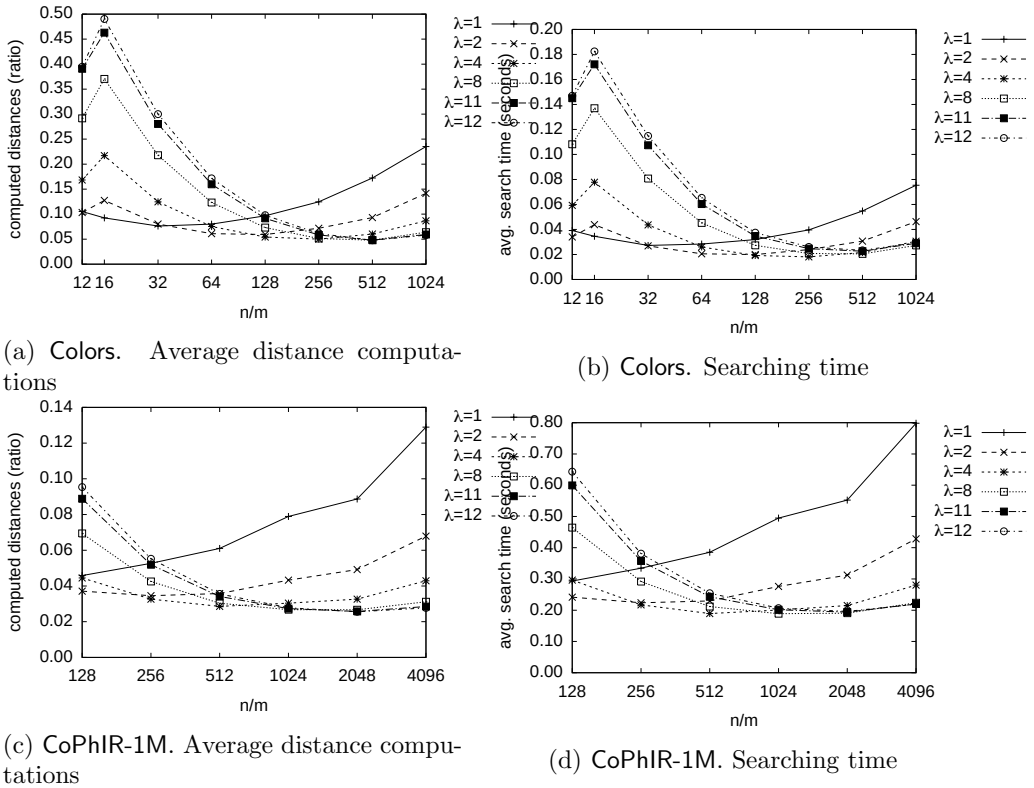


(a) RVEC-*-250000, $n/m = 16$ (b) RVEC-*-500000, $n/m = 16$ (c) RVEC-*-1000000, $n/m = 16$

(d) RVEC-*-250000, $n/m = 128$ (e) RVEC-*-500000, $n/m = 128$ (f) RVEC-*-1000000, $n/m = 128$

(g) RVEC-*-250000, $n/m = 1024$ (h) RVEC-*-500000, $n/m = 1024$ (i) RVEC-*-1000000, $n/m = 1024$

Figure 5.3: PMI's real time performance on increasing intrinsic dimension and increasing n on *Random vectors* (RVEC) databases. nn searching.



(a) Colors. Average distance computations

(b) Colors. Searching time

(c) CoPhIR-1M. Average distance computations

(d) CoPhIR-1M. Searching time

Figure 5.4: Performance of the PMI on two real world datasets.

Chapter 6

Compressing Metric Indexes

On the previous chapter, we introduced a powerful metric index, outperforming the List of Clusters (LC), one of the faster metric indexes available on the literature. We called it the Polyphasic Metric Index (PMI). It holds a fast construction, and fast searching times. Also, the PMI has a small memory footprint compared to pivot indexes, yet, it requires to store λ LC indexes where λ is a small value dependent on the intrinsic dimensionality of the dataset.

In this chapter, we find the minimum number of bits required to represent an LC of a database with n objects, and a bucket size of n/m. However, the techniques that we use to describe the structure of the index are general enough to implement other metric indexes, yet we limit ourselves to LC since the metric information is highly dependent of the particular index.

6.1 Introduction

As already commented, pivot based indexes suffer from excessive space complexity, particularly those exhibiting smaller time complexities on high intrinsic dimensions. The extra memory is used on the allocation of a linear number of distance values from all database's objects to some selected object called pivot. As proved by Chavez et al. [Chávez et al., 2001] the optimal number of pivots is too large on high intrinsic dimensional datasets. This limits its applicability only to small datasets due to memory issues.

A less memory expensive family is based on creating a set of compact partitions of the database. Although a smaller performance is expected, it can be really competitive in some exponents, such as LC [Chávez and Navarro, 2005] and the M-Tree [Ciaccia et al., 1997]. Typically, an LC requires $O(n)$

identifiers and pointers to represent the tree structure, i.e., $O(n \log n)$ bits in a RAM machine. On practice, each identifier (and pointer) uses a word machine (an integer of either 32 or 64 bits on a modern computer) plus the memory to represent all results of $\mathsf{cov}(\cdot)$ (Section 2.2.2). In this chapter, we introduce a lower bound of the LC's memory requirement on a database of size n using m centers, $(n-m)\log\frac{nm}{n-m} + m\log\frac{n}{m}$ bits. Also, we present an implementation closely achieving this bound. Finally, we study special cases of interest consuming less memory than the general lower bound.

Our indexes require half the memory space of the plain representation for the studied datasets, without a neither preprocessing nor searching time penalty. This achievement can be straightforwardly applied to the Polyphasic Metric Index.

6.2 Compressed Metric Indexes

Compression of exact metric indexes is a novel technique, at the best of our knowledge, this is the very first attempt to compress an exact metric index. However, there are some successful efforts trying to reduce the necessary memory of metric indexes, mostly, the idea is to increase the number of pivots to be kept on memory. Under this approach, we found the Fixed Queries Array (FQA) [Chávez et al., 2001], and the Fixed Queries Trie [Chávez and Figueroa, 2004]. Both indexes try to reduce the memory footprint of the Fixed Hight Fixed Queries Tree (FHFQT) [Baeza-Yates and Navarro, 1998].

6.3 Revisiting the Data Structure

The next statements apply for most metric indexes, yet we focus our efforts on the LC, since it is a good example of a successful index for high dimensions. Also it is used as backend in our Polyphasic Metric Index (Chapter 5).

Let us recall the basic properties and operations of the data structure behind the LC index. The preprocessing step must deal with the following issues, Section 2.2.2.

(a) It recognizes the set of centers C, i.e., m objects selected from n possible candidates. The original implementation uses $O(m)$ machine words, mostly for pointers.

(b) Each one of the remaining $n-m$ objects is assigned to a bucket of the m possible centers. The buckets are plain arrays (or linked lists)

containing all $n - m$ object identifiers. Each identifier requires $\log n$ bits, in practice a complete word machine is used.

(c) The function $\mathsf{cov}(c_i)$ is defined for each center, i.e. $\mathsf{cov}(c_i)$ for all $c_i \in C$. In the original structure, it is stored as satellite data in LC's nodes. Each distance requires f_d bits, the necessary bits to represent any value of d. So, all responses to $\mathsf{cov}(\cdot)$ are stored with $f_d m$ bits.

Our main focus is to reduce the space requirements of (a) and (b). Finally, (c) is stored in some convenient order.

6.3.1 Operations of the LC

In order to solve queries, any LC structure must be able to retrieve centers (C) in the same order as seen by the tree, retrieve items being internal to center c, i.e. I_c. Finally, solve $\mathsf{cov}(c_i)$.

An alternative view of the data structure is as follows. Under an enumerated database from 1 to n, each object u is labeled according to issues (a) and (b), as follows:

$$\mathsf{tag}(u) = \begin{cases} 0 & \textbf{if } u \in C \\ i & \textbf{if } u \in I_{c_i} \end{cases} \tag{6.1}$$

Let $T = \mathsf{tag}(u_1)\,\mathsf{tag}(u_2)\cdots\mathsf{tag}(u_n)$ be the string composed of the concatenation of all labels in the order of the database. Each label will be manipulated as a *symbol*. An array $\mathsf{COV}[1, m]$ is stored to solve $\mathsf{cov}(\cdot)$, such that $\mathsf{COV}[i] = \mathsf{cov}(c_i)$. The order of the centers is the one found at the database.

6.3.2 Storage Requirements

As a consequence of our mapping, the main part of our index is a text T over an alphabet $\Sigma = 0, 1, \cdots, m$ of size $m + 1$. Consider that n_i is the number of items labeled with symbol i, then the lower bound based on Information Theory to store a string of length n using a unique code per symbol is as follows.

$$\log \binom{n}{n_0, n_1, \cdots, n_m} = \log \frac{n!}{\prod_{i\in[0,m]} n_i!} \tag{6.2}$$

$$= nH_0(T) \text{ bits.} \tag{6.3}$$

Thus, we require at least this quantity of bits to be able to represent any possible instance of an LC with n objects and m centers. $H_0(T)$ is the order zero entropy of T. A more convenient formula is the following:

$$nH_0(T) = \sum_{i \in [0,m]} n_i \log \frac{n}{n_i} \text{ bits.} \tag{6.4}$$

In the particular case of the LC with buckets of fixed size, all $n_i = (n-m)/m$ are equal excepting for $n_0 = m$,[1] hence the following formulation arises (all terms are representing bits).

$$\begin{aligned} nH_0(T) &= \sum_{i \in [0,m]} n_i \log \frac{n}{n_i} && (6.5)\\ &= \sum_{i \in [1,m]} \frac{n-m}{m} \log \frac{nm}{n-m} + m \log \frac{n}{m} && (6.6)\\ &= (n-m) \log \frac{nm}{n-m} + m \log \frac{n}{m} && (6.7)\\ &\leq n \log m \text{ bits.} && (6.8) \end{aligned}$$

Equation 6.7 is important on datasets with large intrinsic dimensionality since n/m should be set to $O(1)$, as pointed out by Chavez and Navarro [Chávez and Navarro, 2005]. So, we can expect smaller representations of T, however, taking advantage of this feature is only possible for very high intrinsic dimensions, where LC (and any metric index) are useless. Inequality 6.8 establishes the worst case, which is smaller or equal than a carefully implemented Plain LC, i.e., $n \log m \leq n \log n$ bits. Contrasting with our representation, the traditional implementation uses a complete pointer/integer (32 or 64 bits) in the machine instead of $\log n$ bits.

Now, T requires at least $nH_0(T)$ to represent any possible instance, under the stablished terms. We will show an efficient way to index T while kept close to optimal storage.

6.3.3 Implementing LC Operations

Using an Index of Sequences (IoS), Appendix A, it is possible to represent the desired structure in the promised space, $nH_{0(T)}$ bits. Also, the primitives of an IoS reproduce all operations required by the LC. Our functionality is

[1] Another exception could arise, since the last bucket can be smaller than others. However, without lose generality, we ignore this exception.

strongly based on Select operation, and lesser on Rank. Even when Access is not necessary to reproduce the functionality of the LC, it is of great help to certain applications. For example, those applications recognizing the neighborhood of an object in a timeline induced by the database enumeration, i.e., it retrieves what center is owning any given object.

LC's operations (Section 6.3.1) are implemented as follows:

(i) Iterate over C. It is solved with $\mathsf{Select}(T, 0, r)$ for $1 \leq r \leq m$.

(ii) Solving $\mathsf{cov}(c_i)$ simply returns $\mathsf{COV}[i]$. An unknown i can be computed as follows. Let us suppose $p = c_i$, then $i = \mathsf{Rank}(T, 0, p)$.

(iii) Iterate over I_{c_i}. The bucket related to center c_i is retrieved as $\mathsf{Select}(T, i, r)$ for $1 \leq r \leq \mathsf{Rank}(T, i, n)$.

Additionally, we can retrieve the context of the i-th object, i.e., the center identifiers around it, using $\mathsf{Access}(T, i \pm j)$ for some $j = 0, 1, \cdots$. For example, in a serie of events in time, we can retrieve the kind of event (denoted by the center covering objects) that precede and follow an incident. This behavior is specially welcome in forecast and forensic applications.

6.4 Experimental Results

The running hardware, operating system, and conditions are the same than those described at Section 1.3. Our datasets are Colors, CoPhIR-1M, and the synthetic datasets of RVEC-*-1000000.

Please note that the compressing time is worthless compared against the preprocessing time necessary by the LC index, then it is not reported. Even when m is the parameter that determines the space requirements in the worst case, the parameter n/m is the one managed by the LC in the literature, thus, we select n/m in order to allow an easy comparison (and extrapolation) against to previous work.

All memory costs figures shows two special curves, nH_0 and $n \log m$. The curve nH_0 depicts the theoretic lower bounds, as determined in Equation 6.7. In the same way, $n \log m$ shows the necessary storage of the worst case for the given n and m (Equation 6.8). Also, the number of distances performed by the LC is common to compressed and uncompressed versions, so, this performance is not important for our current study. Also, on figures showing searching times, the differences among *Plain* and other methods show the payment for the compression techniques.

On high intrinsic dimensions, LC needs setups where m follows n, such that the alphabet of our sequences is large. In contrast, our PMI requires setups with smaller m, but in any case, all LC primitives are (consecutive) Select based operations in our compressed representation. So, we require an indexed sequence (IoS) with good support for large alphabets, and fast Select implementation. Accordingly to Appendix A, the best IoS for our problem characteristics are the following ones.

- Golynski et al. [Golynski et al., 2006], an uncompressed but fast index.
- Our family of indexes called Extra Large Bitmaps, specifically XLB-SArray and XLB-DiffSet ($B = 31, 63, 127, 255$).[2] We fix $t = 16$ for all indexes and configurations (see Appendix A).

Memory Usage and Searching Times on Colors

Figure 6.1 shows the performance on Colors. Both curves (nH_0 and $n \log m$) are pretty closer, showing the incompresible case exposed by LC.

The LC's memory requirements for the Colors dataset is depicted in Figure 6.1a. Here we observe that GMR06 and XLB-SArray have a similar performance for $n/m \leq 256$, after this point XLB-SArray produces smaller indexes. Yet, this behavior is expected since GMR06 is specially designed for large alphabets, while XLB-SArray performs well for many other setups. Even when GMR06 and XLB-SArray achieves $n \log m$ bits, and in the case of LC we obtain that $nH_0 \simeq n \log m$. However, a great difference arises on searching times, since GMR06 is not benefited of the pattern accesses of the LC's searching algorithms, contrary to most XLB indexes. On the other side, XLB-DiffSet perform pretty well for large B (sampling gap, Appendix A). However, XLB-DiffSet indexes does not improves over worst case indexes (GMR06 and XLB-SArray), meaning that the Colors dataset does not has a strong correlation between object's position and closeness. This is the worst case of XLB-DiffSet (uniform distribution), Appendix A, since for this index the memory space is dependent of the relation between order and closeness. For instance, let us permute the database and re-index the dataset again (using our randomized construction, Section 5.4.2), such that the relation between locality and closeness is increased, without obtaining the LC. The resulting behavior is shown by Figure 6.2. In this figure, both GMR06 and XLB-SArray still have the same memory cost. This effect is an evidence that both structures do not improve on this kind of instances. On the other

[2] DiffSet is called DSet inside figures in order to reduce the size of labels.

hand, XLB-DiffSet is adaptive to the instance, and its memory requirements become smaller (on this case we obtain a 20% reduction). Please note that searching times remain untouched, even on our smallest index ($B = 255$). Also, a small speed up induced by a better locality patterns is present on the permuted dataset, Figure 6.3.

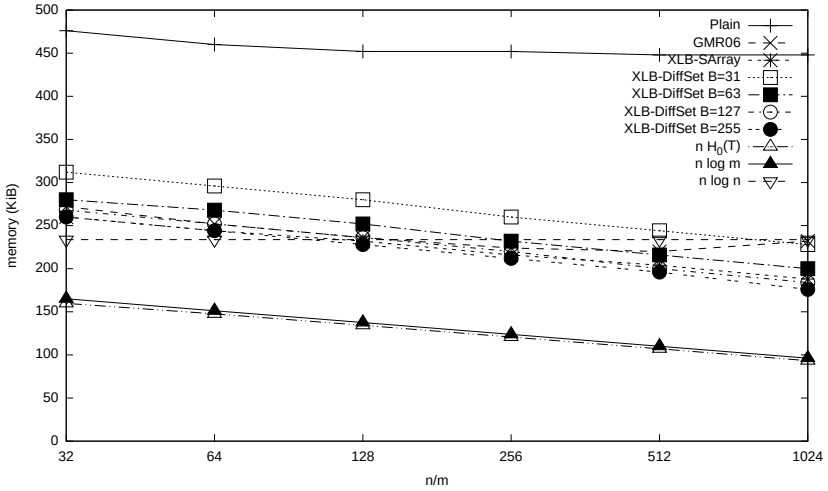
Memory Usage and Searching Times on CoPhIR

On this dataset, the XLB-DiffSet indexes already show competitive performances, as compared to XLB-SArray and GMR06, indicating a correlation between locality and closeness. As expected, both GMR06 and XLB-SArray have a similar performance on the memory usage. However, they hold a great difference on the searching time. This difference is a consequence of the access pattern of the LC (consecutive arguments on Select_c calls). As shown in Appendix A (and by the performance of Colors), this pattern access is not well handled by GMR06, and exceptionally good by most XLB indexes.

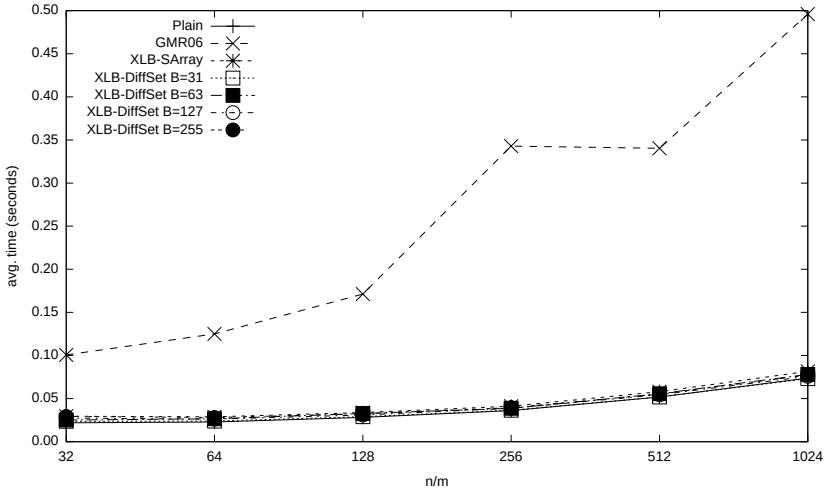
6.4.1 Performance on Synthetic Datasets

Figure 6.6 shows the performance of the compressed LC on synthetic datasets with varying dimensionality. Searching times are grouped by bucket size, i.e. $n/m = 16, 128, 1024$. For instance, Figure 6.6a shows the performance for $n/m = 16$. Here, GMR06 has the slower times, as on real-world datasets, and XLB-DiffSet shows its dependency on B (i.e. large B values produce smaller indexes). This dependence on B is less noticeable on real-world databases due to the correlation between distance and item location on the database. As n/m grows the impact of B decreases, since the number of occurrences of each symbol is large (n/m to be precise). This is a consequence of the $O(1)$ amortized time of the consecutive Select operations, as shown in Figures 6.6b and 6.6c.

Finally, the memory requirements of the compressed LC are shown in Figure 6.7. Here, we fix n/m since the memory requirements seems to be independent of the dimensionality. However, the value of n/m is already dependent of the dimension. The figure shows curves for several sequences. Each curve has three points for $n/m = 16, 128$ and 1024. The XLB-SArray and GMR06 have the better performance, while XLB-DiffSet indexes have higher memory requirements. These results suggest that XLB-SArray is a good option on random datasets, since it achieves compression ratios of 60% to 50% against the uncompressed LC.



(a) Memory requirements



(b) Searching time (nn)

Figure 6.1: Performance of the compressed LC index for the Colors dataset.

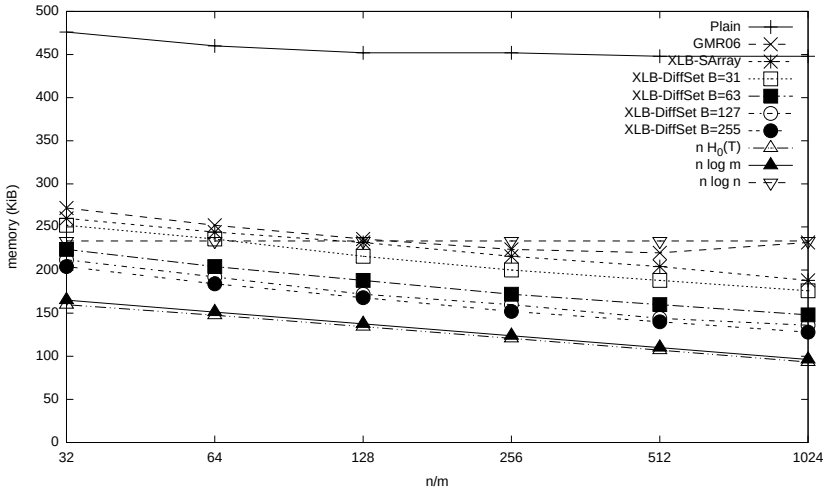


Figure 6.2: Memory usage of the permuted Colors

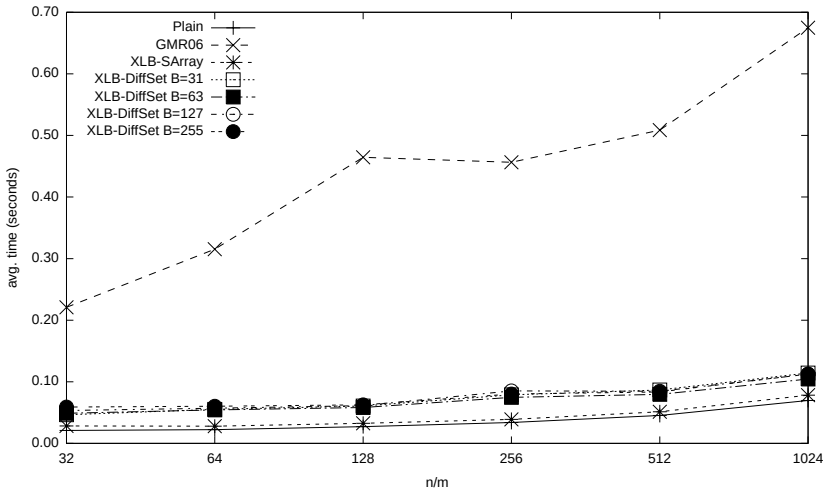
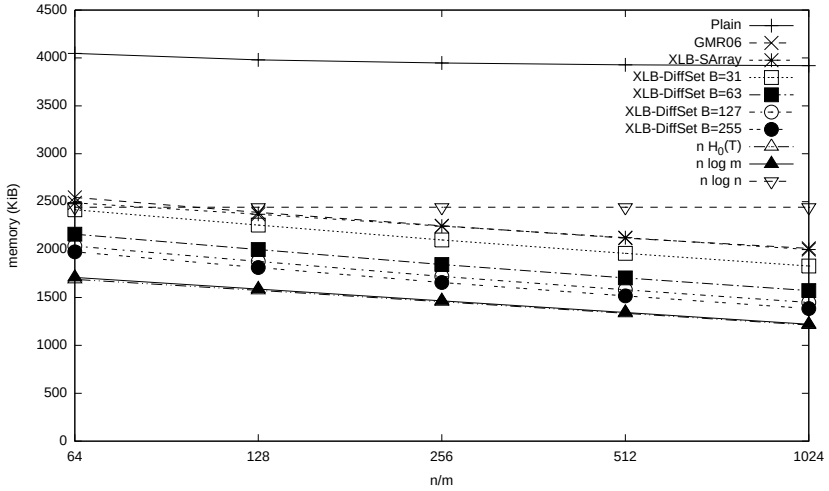
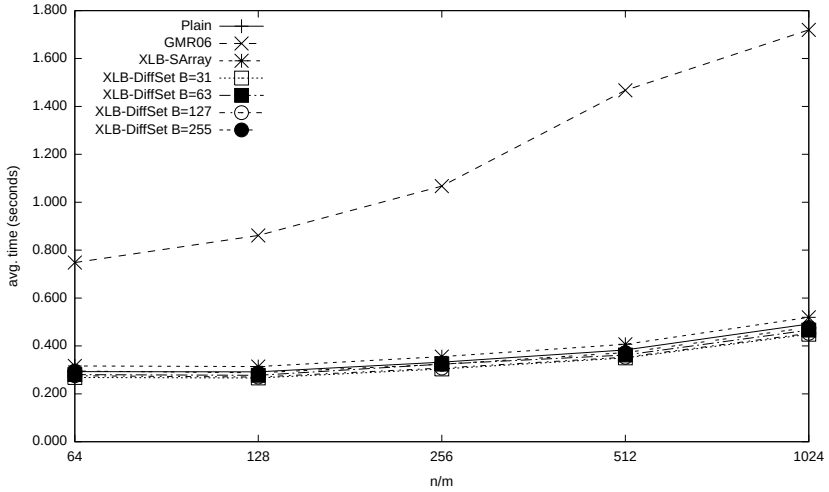


Figure 6.3: Searching time of the permuted Colors

Figure 6.4: Performance of the compressed LC on the permuted Colors database. The permutation was induced with an LC with $n/m = 128$.

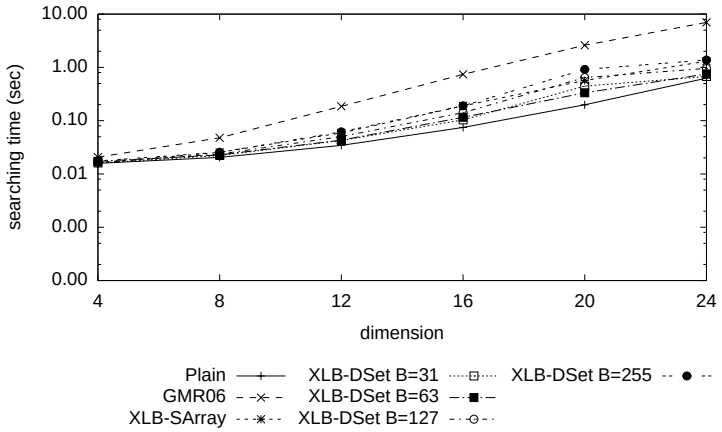


(a) Memory usage

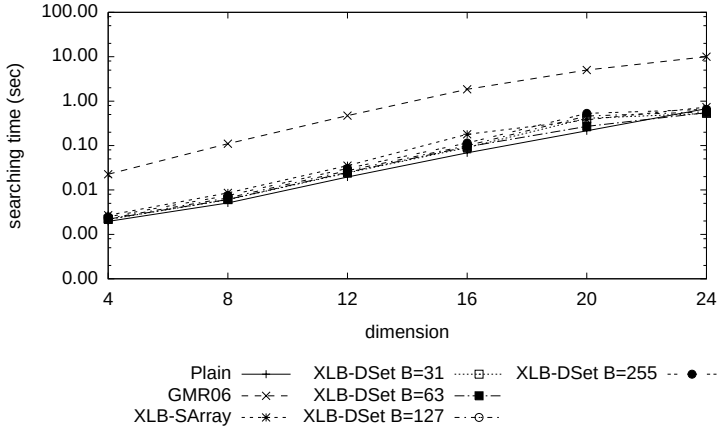


(b) Searching time

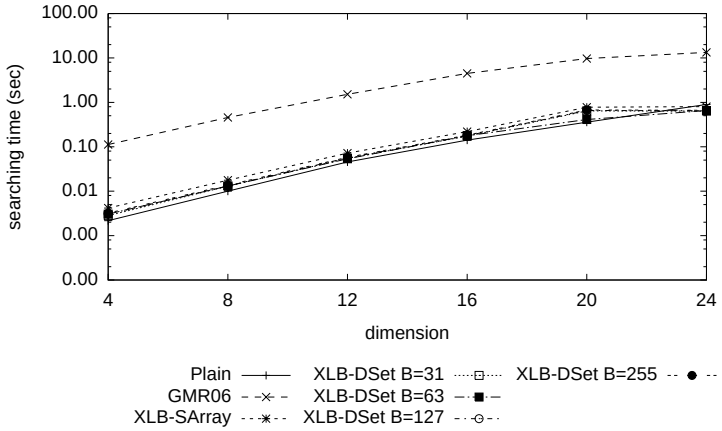
Figure 6.5: Performance of the compressed LC index on CoPhIR-1M



(a) $n/m = 16$



(b) $n/m = 128$



(c) $n/m = 1024$

Figure 6.6: Searching times of the compressed LC index on RVEC-*-1000000

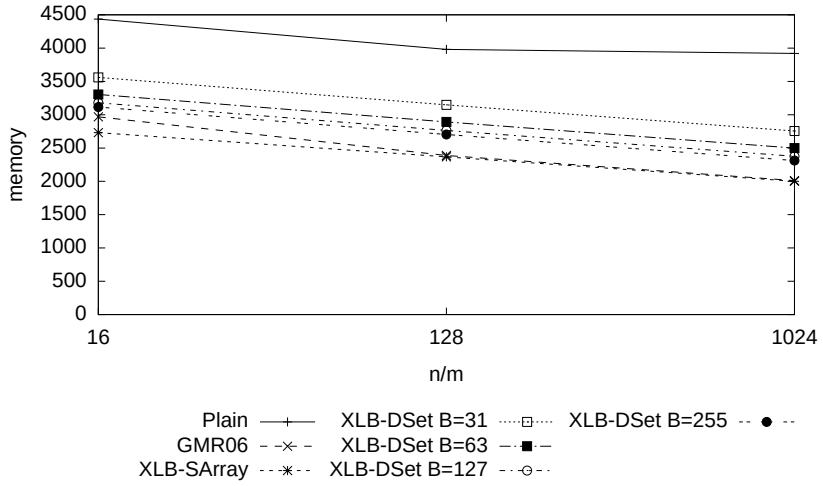


Figure 6.7: Memory requirement of the compressed LC index on RVEC-*-1000000

6.5 Special Cases

The LC creates a partition of the database, that can be represented as a string of symbols T, represented with $nH_0(T) \leq n \log m$ bits using the tagging function of the Equation 6.1. Any associated metric information is independent of this representation, but they are easily linked and accessed using sequence's primitives.

We observed that the compression ratio of XLB-DiffSet is directly dependent of the preservation of the order of the database. Thus, in the limit, the LC does not permute the database since the order induced by LC is the same than the order of the dataset, using $\log \binom{n}{m} = m \log \frac{n}{m} + O(m)$ bits, i.e., the necessary bits to mark centers and store the size of the buckets. Notice that this is the case of buckets of variable size on the LC. In this case we save up $f_d m$ bits, since we do not need to store $\mathsf{cov}(\cdot)$, while T requires $m \log \frac{n}{m} + O(m)$ bits (that is, a single bitmap). On the other side, only $f_d m$ bits are necessary to store COV if the bucket size is fixed.

It is unlikely that a database hold the same order than the LC, but it is really easy to construct this instance. The procedure is as simple as permute the database's objects in the order of the LC, and finally rewrite T's LC with the new order.

Unfortunately, this construction is limited to databases that allow to change the order, that is not the general case, but a large number of applications actually can do it. Furthermore, this permutation largely improves the performance of a secondary memory index, since it increases the locality inside buckets, and reduces the primary memory requirements, i.e., $f_d m$ or $\log \binom{n}{m}$ bits.

6.5.1 Polyphasic Metric Index with Compressed Indexes

Chapter 5 introduces a fast metric index, called Polyphasic Metric Index (PMI). In practice, it is composed of λ LC indexes. Even when each individual index is created with n/m much larger (e.g. $n/m = \sqrt{n}$) than the necessary value for a single optimal LC ($n/m = O(1)$), its memory requirements increases considerably (i.e. λ times). Using the compressed representation of the present chapter, we are able to reduce significantly the memory overhead of the PMI, while kept a fast preprocessing and searching steps.

Table 6.1 and 6.2 show the compression ratio of several compression methods (sequences) on five n/m setups on our real-world datasets. CoPhIR-1M compression ratios are shown in Table 6.1, here we can observe ratios

	n/m				
	64	128	256	512	1024
GMR06	0.636	0.597	0.562	0.53	0.503
XLB-SArray	0.622	0.592	0.561	0.531	0.500
XLB-DiffSet $B = 31$	0.604	0.564	0.525	0.490	0.457
XLB-DiffSet $B = 63$	0.540	0.500	0.461	0.426	0.393
XLB-DiffSet $B = 127$	0.509	0.469	0.430	0.395	0.362
XLB-DiffSet $B = 255$	0.494	0.453	0.414	0.379	0.346

Table 6.1: Ratio between the plain representation of the LC and the compressed ones on the CoPhIR-1M dataset

	n/m				
	32	64	128	256	512
GMR06	0.60177	0.55752	0.52212	0.49557	0.48673
XLB-SArray	0.57522	0.53982	0.51327	0.47788	0.45133
XLB-DiffSet $B = 31$	0.55752	0.52212	0.47788	0.44249	0.41593
XLB-DiffSet $B = 63$	0.49558	0.45133	0.41593	0.38053	0.35398
XLB-DiffSet $B = 127$	0.46903	0.42478	0.38053	0.35398	0.31858
XLB-DiffSet $B = 255$	0.45133	0.40708	0.37168	0.33628	0.30974

Table 6.2: Ratio between the plain representation of the LC and the compressed ones on the permuted Colors dataset

from 0.64 to 0.35, for GMR06 with $n/m = 64$ and XLB-DiffSet $B = 255$ with $n/m = 1024$, respectively. The effectiveness of XLB-DiffSet suggest a tight relation between distance and item location. This relation allow us to create very small indexes. For instance, we can create close to three indexes using XLB-DiffSet $B = 255$ with $n/m = 1024$ (the smallest compression ratio) on the same space required by a Plain LC. So, the compression drastically reduces the PMI's memory requirements, for example and index requiring $\lambda = 12$ only multiplies the memory usage in a factor of 4 not 12 as on the previous approach. Remarkably, datasets with a small intrinsic dimension will require large n/m and small λ values, Figure 5.3, this can yield to indexes even smallers than the Plain LC.

In general XLB-SArray is a good worst case solution, since it achieves a good tradeoff between space and searching time. However, we can achieve a better compromise with XLB-DiffSet if there exists a correlation between the item's location and the distance. This correlation does not necessarily exists, yet we always can artificially increment this property as we did it on the Colors dataset. Also, on the PMI, the following procedure can be of help:

1. Permute the database by proximity using the LC itself, as demonstrated for Colors. Notice that we obtain a much smaller index as commented on the previous section.

2. For the resting $\lambda - 1$ indexes, we index them as usual yet using XLB-DiffSet.

The first step actually increases the correlation between locality and distance, and the second step takes advantage of this property to obtain smaller (and faster) indexes, as shown in Figure 6.4 and Table 6.2. Finally, when databases cannot be permuted it is impossible to avoid the storage of the first LC (first step). The second stage should use the permutation of the first step in order to obtain the desired properties.

6.6 Summary

In this chapter we tackle the compression of metric indexes, particularly, we focus on the well known List of Clusters, i.e., the main building block of our Polyphasic Metric Index (PMI), Chapter 5. At the best of our knowledge, this is the first attempt to compress exact metric indexes. We state the storage lower bounds to represent the index, our technique is to transform the metric index into a sequence of symbols. Our method, is general and

can be applied to metric indexes organizing the structure in disjoint classes of equivalence (as most metric indexes, Chavez et al. [Chávez et al., 2001]). Also, we reproduce all operations of the LC, and furthermore our representation supports more primitive operations, like those being implemented with Access.

Our representation has not noticeable speed payment, since compressed representations take advantage of the memory hierarchy present in modern hardware.

The compression problem behind the LC is quite complicated if modifying the order of the database is prohibitive, since the underlying sequence is compressed to H_0. The cause is that buckets have the same length (or similar size on fixed radius construction). Also, a higher entropy model H_h cannot be used since we cannot ensure an appropriated context to exploit this scheme. Even with this limits, our representation is smaller than half the plain representation.

If we allow to change the order of the dataset, we obtain a very small index of either mf_d or $\log \binom{n}{m}$ bits.

Finally, our PMI has a direct improvement with the compression of the LC. The PMI requires λ indexes, but using compressed LC indexes reduces memory requirements to $\lambda(n \log (m+1) + mf_d)$ bits, in the worst case. Thus, we obtain a smaller structure where the compression ratio against the uncompressed PMI is of $\frac{\log (m+1)}{\log n}$, without a noticeable payment in the searching time.

Part III

Approximate Proximity Searching

Chapter 7

Locality Sensitive Classification

In this chapter we present a new representation for the Locality Sensitive Hashing (LSH) index. This is our first contribution to the approximate proximity searching indexes.

LSH is the industry standard for proximity searching tasks. It is a fast approximate proximity searching technique giving probabilistic guarantees on the quality of the result set. In practice, an LSH index applies a set of hashing functions to the representation of an object to *clustering* proximal objects, i.e., objects inside a cluster will be one another proximal with high probability. Even that LSH is defined for general metric spaces, currently the hashing functions preserving the required properties for any metric space are unknown. Thus, hashing functions are based on the representation of the objects. LSH is a powerful tool if the representation of an object is available.

7.1 Introduction

In the literature, LSH indexes are typically organized as hash tables, such that objects in the same bucket are closer under a distance function d with high probability [Indyk, 2004; Andoni and Indyk, 2008; Gionis et al., 1999].

As commented, LSH offers quality guarantees if the hashing functions hold some basic properties. Nevertheless, these guarantees are distance based, not recall based. This is enough for several real-world applications. Also, there exist a limit on the probabilistic guarantees that can be ensured with a sigle instance of LSH. If higher quality is required, the simplest solu-

tion is to use multiple LSH instances. In this context, high quality results imply larger memory requirements. Our motivation is to reduce the space requirements of high quality LSH indexes, while maintain a moderate time overhead.

In this chapter we introduce a new representation of the Locality Sensitive Hashing (LSH) indexes, for static datasets, with close to optimal storage. Also, we extend primitives of LSH with the extraction of context of a proximity search matches, without increase memory space. The idea is to view hashes as labels on a classification process, such that items marked with the same label will be proximal with high probability (in the same sense that LSH does for items in the same bucket). The technique is similar to that applied to compress the List of Clusters, Chapter 6.

Summarizing, the traditional hash table is replaced by a sequence T (indexed with an index of sequences, Appendix A). Under this scheme, we found several memory-time tradeoffs, and new interesting primitive operations.

7.1.1 Locality Sensitive Hashing

Locality Sensitive Hashing (LSH) is the default option for proximity searching in most structured databases, as those described in vector spaces. LSH is a fast index with tunable accuracy. It is simple to implement and it can be used in very large databases and moderately high dimensions. But a major disadvantage exists, since LSH needs a set of a hashing functions g_i with particular properties for each data model, and distance function. The process of finding g_i can be a hard problem by itself. Hence, LSH requires an in-depth knowledge of the structure of the underlying metric space, and indeed, of the database's properties.

Definition 7.1.1 (Locality Sensitive Hashing, Gionis et al. [Gionis et al., 1999]) *A family of hashing functions $\mathcal{H} = \{g_1, g_2, \cdots, g_h\}$, $g_i : U \to \{0, 1\}$ is called (p_1, p_2, r_1, r_2)-sensitive, if for any p, q:*

— *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, i.e. $\mathsf{hash}(u) = g_1(u)g_2(u)\cdots g_h(u)$.

Let d_{max} be the maximum possible distance between objects in the metric space; the probability that some g_i computes the same hash for $u, v \in U$

is determined as

$$Pr[g_i(u) = g_i(v)] = 1 - d(u,v)/d_{max}$$

If hashing functions are defined independent, with replacement, and equally probably to fail, we obtain

$$Pr[\mathsf{hash}(u) = \mathsf{hash}(q)] = 1 - (d(u,v)/d_{max})^h$$

In order to well defined LSH scheme, this formula should follow the properties of definition 7.1.1.

Example 7.1 *LSH is naturally defined on* Hamming *spaces, here a hashing function is simply defined as a randomly picked bit from the bit-vector, as detailed by Gionis et al. [Gionis et al., 1999], Andoni and Indyk [Andoni and Indyk, 2008].*

Considere a binary Hamming space of height dimensions, and $u, v, w \in U$ *defined as follows:*

		12345678
u	=	10010101
v	=	11010000
w	=	01001101

Then, $D(u,v) = 3$, $D(u,w) = 4$, $D(v,w) = 5$. *Let* hash_{15} *be the final hash produced by family* $\mathcal{H}_{15}$*, that retrieve and concatenate the first and fifth bits of an object. In the same way, we define* hash_{74} *(and* $\mathcal{H}_{74}$*). After the application to our three testing objects we obtain*

— $\mathsf{hash}_{15}(u) = 10$, $\mathsf{hash}_{15}(v) = 10$, $\mathsf{hash}_{15}(w) = 01$

— $\mathsf{hash}_{74}(u) = 01$, $\mathsf{hash}_{74}(v) = 01$, $\mathsf{hash}_{74}(w) = 00$

Under both families ($\mathcal{H}_{15}$ *and* $\mathcal{H}_{74}$*) we classify* u, v *as equally items. Notice that they are the real near neighbors. So, if we want to classify near neighbors in the same equivalence class (defined by the equality of hashes) with probability of* 0.9*, for* u *and* v *(i.e.* $d(u,v) = 3$*, and* $d_{max} = 8$*)* h *must be* $\log 0.1 / \log 0.375 = 2.35$.

In practice, without loose of generality, we can represent vector spaces in L_p with integer coordinates, i.e., since we can multiply coordinates by a sufficiently large integer. Then, we concatenate every coordinate represented as a fixed-size unary number finally L_1 can be computed using the

hamming distance. An explicit computation of the unary representation is not required. Furthermore, the L_2 norm can be embedded into L_1 using projections. In general L_p can be projected in any other L_s, for $s < p$ using random projections, as detailed by Gionis et al. [Gionis et al., 1999], and Andoni and Indyk [Andoni and Indyk, 2008].

7.2 Sequence Representation

Consider the database $S \subseteq U$, $S = \{u_1, u_2, \cdots, u_n\}$, and a family of hashing functions $H = \{g_1, g_2, \cdots, g_h\}$, where $h = |H|$ and $g_i : U \to \{0, 1\}$. A tag of an object is defined as $\mathsf{tag}(u) = g_1(u)g_2(u)\cdots g_h(u)$. The set of all possible values of $\mathsf{tag}(\cdot)$ is called the alphabet, $\Sigma = \{0, 1, 2, \cdots, \sigma - 1\}$ where $\sigma = |\Sigma| \leq 2^h$. Even when $\mathsf{tag}(u) = \mathsf{hash}(u)$, conceptually tag is an atomic item (indivisible and recognized as a unit), and defines a sequence's symbol. Let us define $T = \mathsf{tag}(u_1)\,\mathsf{tag}(u_2)\cdots\mathsf{tag}(u_n)$. We can store T using $\log \binom{n}{n_1, n_2, \cdots, n_\sigma}$ bits, where n_i is the number of occurrences of the tag i in T.

Based on Information Theory, using an unique codeword per symbol, an alternative lower bound on the storage of T is as follows:

$$nH_0(T) = \sum_{i \in \Sigma} n_i \log \frac{n}{n_i} \tag{7.1}$$

$$\leq n \log \sigma \tag{7.2}$$

The procedure is similar to the compressed representation of the List of Clusters of Section 6.3.2.

This sequence can be represented with a Index of Sequences (IoS), Appendix A. The primitives of an IoS can reproduce all operations required by LSH. Our functionality is strongly based on Select_c operation, and it has a particularly large alphabet, i.e. $O(n/\sigma) = 1$. These both properties have an important impact on the performance of our index, and in general of IoS, as detailed on Appendix A.

7.2.1 Solving Approximate Nearest Neighbors with T

In short, a data structure for LSH needs to access to the clusters. In order to solve a query the structure needs to count the number of items in a cluster, and retrieve all items on it. Figure 7.1 shows a hash table of a fictitious database of 16 objects. Each row is a cluster, represented by some hash value.

In a way, T solves similarity queries using the same proximity properties than LSH tables. Algorithm 10 solves the approximate $\mathsf{nn}_{d,S,U}(q)$ queries.

hash	tag		occurrences list
0000	0	→	11
0001	1	→	6, 12, 13, 14
0010	2	→	5
0011	3	→	7
0100	4	→	3, 10
0101	5	→	15
0110	6	→	4
0111	7	→	2
1000	8	→	1, 8, 9
1001	9	→	16

Figure 7.1: An example of the LSH hash table representation

The idea is of retrieving all items using Select marked with the computed tag.

algorithm 10: Searching for the approximate $\mathsf{nn}_{d,S,U}(q)$

Input: The query q, the distance function d, and T.
Output: The approximate nearest neighbor $\mathsf{nn}^*(q)$

1: Let $c = \mathsf{tag}(q)$
2: Let $\mathsf{nn}^*(q) \leftarrow$ undefined
3: **for** $i = 1$ to $\mathsf{Rank}_c(T, n)$ **do**
4: Define p as $\mathsf{Select}_c(T, i)$-th object in T
5: $\mathsf{nn}^*(q) \leftarrow p$ **if** $\mathsf{nn}^*(q)$ is undefined or p is closer to q than the previous $\mathsf{nn}^*(q)$
6: **end for**

The necessary operations on LSH are reproduced as follows.

— The number of items with the same hash is computed with $\mathsf{Rank}_c(T, n)$ (Figure 7.2b).

— All items with the tag are retrieved as $\mathsf{Select}_c(T, i)$ for $i = 1, 2, \cdots, \mathsf{Rank}_c(T, n)$.

So, our implementation reproduces all basic operations of an LSH table. Also, we introduce the SuccCtx and PredCtx operations, useful to recover the context of an object on a time series application, defined as follows.

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$\mathsf{tag}(u_i)$	8	7	4	6	2	1	3	8	8	4	0	1	1	1	5	9

(a) An LSC sequence T

	$\mathsf{Select}_c(T, i)$			
	1	2	3	4
$\mathsf{Rank}_0(T, n) = 1$	11			
$\mathsf{Rank}_1(T, n) = 4$	6	12	13	14
$\mathsf{Rank}_2(T, n) = 1$	5			
$\mathsf{Rank}_3(T, n) = 1$	7			
$\mathsf{Rank}_4(T, n) = 2$	3	10		
$\mathsf{Rank}_5(T, n) = 1$	15			
$\mathsf{Rank}_6(T, n) = 1$	4			
$\mathsf{Rank}_7(T, n) = 1$	2			
$\mathsf{Rank}_8(T, n) = 3$	1	8	9	
$\mathsf{Rank}_9(T, n) = 1$	16			

(b) Reconstructing the LSH table

$$\begin{aligned} \mathsf{SuccCtx}(T, 8, 4) &= 4011 \\ \mathsf{PredCtx}(T, 8, 4) &= 6213 \end{aligned}$$

(c) Retrieving context

Figure 7.2: An example of the LSH sequence representation LSH, and its operations

— $\mathsf{SuccCtx}(T, p, s)$ returns the s next tags after position p in T, that is, it returns the string of labels $\mathsf{SuccCtx}(T, p, s) = \mathsf{Access}(p+1)\,\mathsf{Access}(p+2)\cdots\mathsf{Access}(p+s)$.

— $\mathsf{PredCtx}(T, p, s)$ returns the s previous tags before position p in T, i.e., $\mathsf{PredCtx} = \mathsf{Access}(p-s)\,\mathsf{Access}(p-s+1)\cdots\mathsf{Access}(p-1)$.

Those new primitives are of use in, for example, forensic and forecasting techniques, that is given a set of ordered events (objects), those new primitives retrieve the classification (hashes) of past and future events around a given event.

7.3 Experimental Results

The running hardware, operating system, and conditions are the same than those described at Section 1.3. We focus our attention on Audio dataset. This dataset contains more than 55 million objects, and each object is a bit-vector dimension 720. The intrinsic dimension is around 140. Details are given in Section 1.3.3. Due to the large alphabet and our Select_c based algorithms, we select three IoS (Appendix A) performing well under these characteristics.

An important measure of the result quality is the recall, i.e. the number of objects returned by the approximate algorithm being part of the exact result set. If we want to preserve efficient searching times (one of our main goals), the quality on LSH is proportional to the number of indexes. Figure 7.3 shows this quantification evaluated on hashing family functions of different size (i.e. $h = 16, 18, 20, 22$), as a function of the necessary memory. Each curve is composed of five points, from left to right, each point correspond to indexes composed of 1 to 5 LSH instances. Each query consist on objects at distance 26 of a database object (0.03 on the normalized distance). Queries being farther yield to bad results. Nevertheless, our intention is not to stress the LSH technique, but study LSH on domains being of interest to most users.

Results obtained with a single LSH's instance contain 60% of the real results (when $h = 16$), 50% for $h = 20$, and a little more than 40% for $h = 22$. On the other side, LSH achieves close to perfect recall on five instances, with a clear inflection point on three instances. Here, GMR06 and XLB-SArray maintain the better compromise with memory requirements. All figures are validating this behavior, however the memory gain becomes smaller as h increases. When $h = 16$, we obtain the smaller memory footprints and

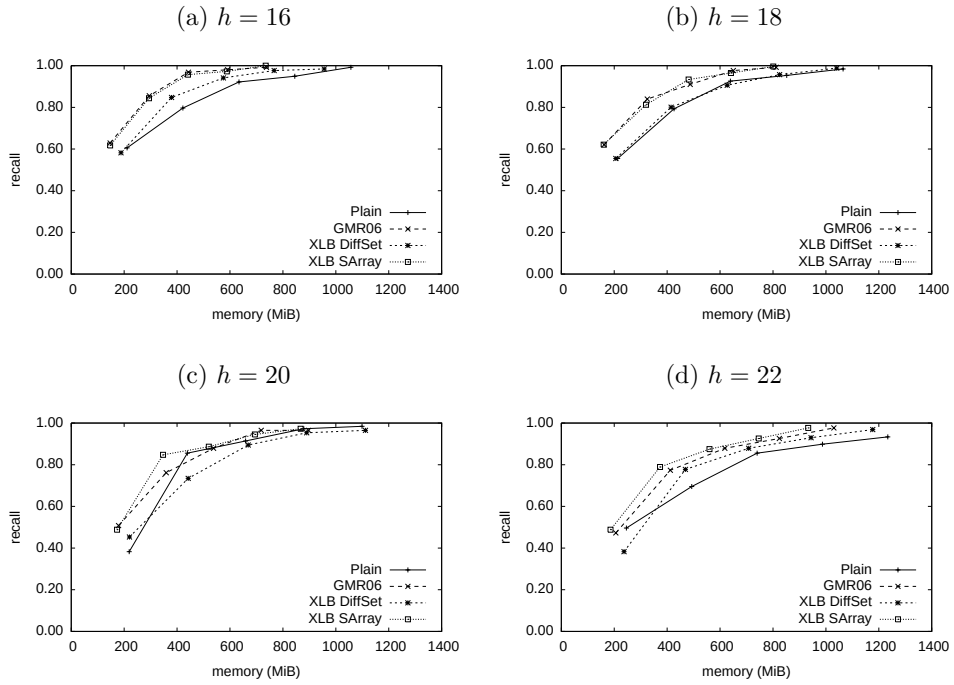
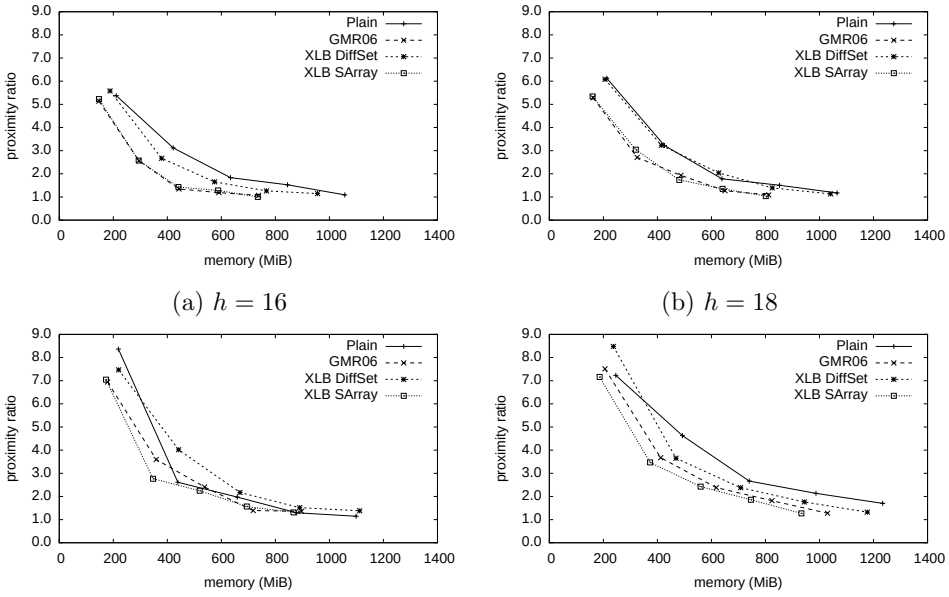


Figure 7.3: Comparison between recall and the required memory on different LSH families. The recall barely vary at each point because random $\mathcal{H}$ where selected.

better recall, yet it has a searching time payment, as will be described on advance. Also, when $h = 22$, the recall has a slower growth rate, and even achieving faster searching times. For all h values, the compression ratio on five LSH instances ranges between 70% and 80%. This improvement is of use on practical applications since the Plain storage ranges 1050 to 1250 MiB of memory, and this memory saving can be the difference on the possibility of maintain indexes on memory or to create an on-disk structure.

Figure 7.4 mesures the quality from another perspective, the proximity ration, i.e. the ratio between the obtained covering radius and the covering radius of the exact result. The ratio ideally should be close to 1. So, while memory increases, the proximity ratio decreases (with an improvement of the result's quality). This measure exposes the quality of results when the exact result is not obtained (or not necessary as in the majority of practical



(a) $h = 16$ (b) $h = 18$

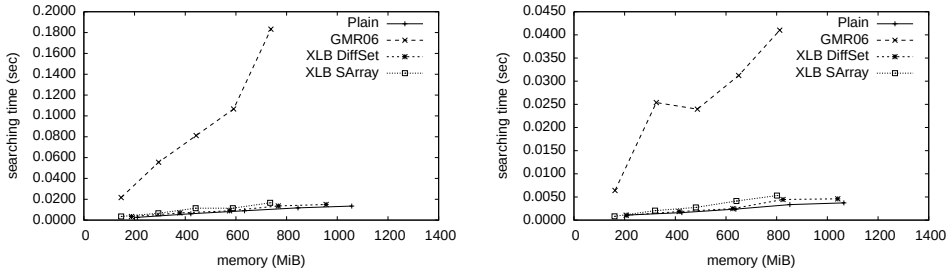
(c) $h = 20$ (d) $h = 22$

Figure 7.4: Comparison between the proximity ratio and the required memory on different LSH families

applications). This is important since, our expected normalized distance value is of 0.03. So, diverging 5 times, means a distance of 0.15, and this difference becomes smaller as the memory increases. Thus, the response is far from the mean, such that the retrieved object is on a small percentile of the dataset, see Figure 1.1e.

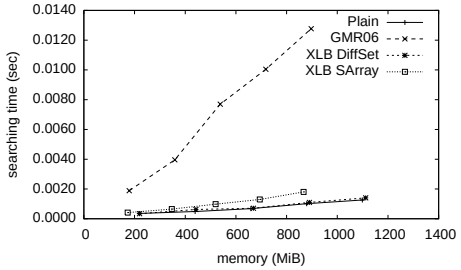
Finally, the searching time is compared against the memory usage (Figure 7.5). Here, we find a great difference between GMR06 and XLB-SArray, the smaller indexes. In short, XLB-SArray is several times faster (and a bit smaller) than GMR06 because the operation patterns of our LSC increments the number of cache's misses, while the contrary occurs for XLB indexes. This effect is well documented in Appendix A. Other alternatives, Plain and XLB-DiffSet, are quite faster but hold a large memory footprint such that their usage is limited to a pair of instances to be competitive against smaller approaches.

It is clear than XLB-SArray is the best option for the LSC. However, as in the Compressed List of Clusters (Section 6) the large memory cost of

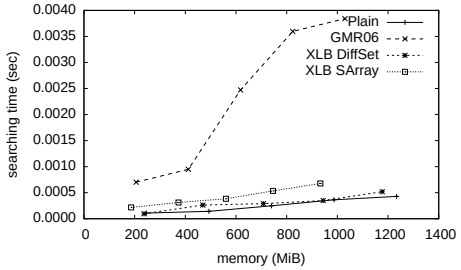


(a) $h = 16$

(b) $h = 18$



(c) $h = 20$



(d) $h = 22$

Figure 7.5: Comparison between the searching time and the required memory on different LSH families

XLB-DiffSet is because the location and the closeness is not related. XLB-DiffSet is a very good option if a (strong) relation between location and proximity can be ensured.

7.4 Summary

In this chapter we introduce a new representation for the Locality Sensitive Hashing, where its proximity information is converted to a sequence of hashes (tags). The proximity searching process is then transformed into a set of IoS primitives. Remarkably, the new representation achieves space close to the optimal one, with a surprisingly good searching times.

The new index was tested on environments with high quality requirements such that several instances were necessaries. Our approach demonstrates to be a fast and small alternative compared to the traditional implementation. Nevertheless, as LSH does, the indexing is limited to databases

with a well define structure, due to the requirements of the hashing functions. Also, only a small difference on the queries is supported.

In the next chapters, we explore other techniques that are more robust to the intrinsic dimensionality, and the remoteness of the nearest neighbors. Also, those new approaches use abstract objects with a common distance function acting as a black box, allowing proximity searching functionalities for complex objects (e.g. objects without a fixed structure, or with a really large number of coordinates), and complex distance functions as well.

Chapter 8

Neighborhood Approximation

In this chapter we introduce a new technique for approximate metric searching, dubbed as *Neighborhood Approximation* (NAPP). Fixing the size of the neighborhoods of NAPP we create the *K nearest references* (K-nr) framework. This framework help us to produce several approximate metric indexes, all of them holding excellent tradeoffs among preprocessing time, searching time, result's quality, and storage cost. Also, we describe several indexes in the literature under the K-nr framework.

We provide an extensive set of experimental results, describing in depth the behavior of the NAPP indexes on several real-world datasets and datasets with controlled parameters.

8.1 Introduction

The general approach to solve any metric proximity problem (either range or k-nn) is to map the objects in the original space into a simpler data space. In the literature there are many examples of such representations. The first generation of these representations used distances to selected points to map objects [Chávez et al., 2001]. Recently, as described in Section 2.2.3, this mapping shifted to an alternative representation using just the relative order to a set of references. Here we generalize the last idea obtaining a new family of indexes based on comparing references. We obtained several indexes with competitive performances, with excellent tradeoffs between preprocessing time, searching time, result's quality, and space cost. Also, several of the previous indexes based on comparing references are described with our

framework.

We found that most of these indexes can be represented as a simple sequence of references. We index these sequences with an index of sequences, Appendix A, obtaining simple and fast algorithms. Furthermore, most algorithms of these indexes can be applied to the same sequence of references. Such that a single index can be used with several searching techniques, allowing to dynamically change the searching method as desired. We describe those algorithms and present extensive experimental results.

8.2 Neighborhood Approximation

Most approaches described in the Section 2.2.3 are variants of the idea of using permutations as object proxies. We will introduce a new formulation of the technique which is more powerful and simple.

We call our approach *Neighborhood Approximation* (NAPP). In a way NAPP is a generalization of the permutation idea and we believe it captures the features responsible of the effectivity (high recall) of the permutations, while simultaneously allowing a compact representation and fast searching. We will reuse the notation of Section 2.2.3.

— A random sampling of the dataset S, R of cardinality $\sigma \ll n$, is dubbed as the set of references. R is ordered.

— γ, the maximum allowed number of distance computations in the final filtering step.

8.2.1 The Core Idea

We partition the space into a set of *regions*. Each region is represented by an object of the database. An object representing a region will be called a *reference*, and set R, is the set of all references, representing all regions.

Each object $u \in U$ is represented by a set of objects, called the *neighborhood* of u, defined as

$$P_{u,r_u} = (u, r_u)_{R,d}$$

That is, the set of objects in R intersecting the ball centered at u of radius r_u. r_u is a parameter to be discussed later. We assume P_{u,r_u} to be an ordered set. The default order will the proximity to u.

Now that the set of regions will act as database object proxies, just as permutations did in permutation based indexes. We follow the same path:

every object in the database is transformed into its neighborhood representation (a set of references) and to satisfy a query q we compute its representation P_{q,r_q}. As in any index, we will obtain a set of candidate objects in S which need to be checked against the query. The list of candidates in NAPP will be objects u such that $P_{u,r_u} \cap P_{q,r_q} \neq \emptyset$.

We believe the above framework captures the essence of the permutations approach, yet it is simpler and provides an excellent tradeoff between space usage, speed, and retrieval quality.

8.2.2 Retrieval Quality Considerations

Consider two objects $q \in U$, $u \in S$ and their respective neighborhoods $P_{u,r_u}, P_{q,r_q} \subseteq R$ (see Figure 8.1). The next two observations follow immediately.

Observation 1 *Let $M = P_{u,r_u} \cap P_{q,r_q}$. If $M \neq \emptyset$ then the distance $d(q,u)$ is lower and upper bounded as follows:*

$$\max_{w \in M} |d(q,w) - d(u,w)| \leq d(q,u) \leq \min_{w \in M} d(q,w) + d(u,w)$$

Observation 2 *If $R \subseteq S$ and q^* denotes the nearest neighbor of q in P_{q,r_q}, then $d(q, \mathsf{nn}_{S,d}(q)) \leq d(q,q^*) = d(q, \mathsf{nn}_{R,d}(q))$.*

The upper and lower bounds depend on the selection of R. If R is dense enough then Observation 2 becomes tighter. A final remark is that R should have the same distribution of S. A rule of thumb is to have as many references as one can handle without slowing down the process of comparing q with the set of references (since we need to compute the distance between q and all references), and select them uniformly at random from the database.

Figure 8.1 shows a bad case on the left, and a more realistic case on the right. This is a core heuristic in our approximate technique.

The parameter r_u defines a ball centered on u, and imposes several problems around the technique. A large value imply large neighborhoods, $\sigma/|P_{u,r_u}| = O(1)$. In contrast, small values can yield to retrieve empty sets of references. Also, fixing the value to a global value is not convenient, since large or empty P_u, r_u sets will arise. A simple alternative consist on fix r_u to the covering radius of the K nearest neighbor of u in R. Thus, we always populate P_{u,r_u}, and if $K = O(1)$ then a small r_u is expected. Using $K = O(1)$ simplifies the storage bounds, and in fact fix to a constant number of integers per object, yielding to deterministic storage requirements.

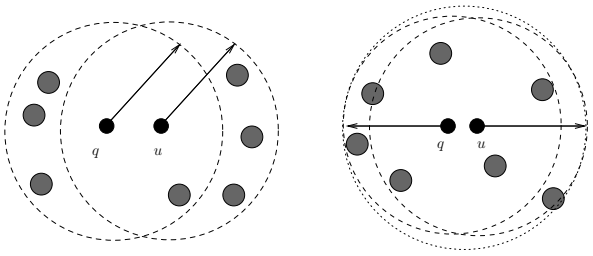


Figure 8.1: An example showing shared references as proximity predictor. Smaller balls are objects in S, bigger ones are references.

8.3 Proximity Searching with the K Nearest References

A simple way to tune NAPP is fixing the neighborhood radius to the covering radius of the K-th neighbor on R. Thus, given some region P_{u,r_u} we define $r_u = \max_{w \in \mathsf{K\text{-}nn}_{R,d}}\{d(u,w)\}$, such that $P_{u,r_u} = \mathsf{K\text{-}nn}_{R,d}$. In advance, we simplify our notation to simply our region notation to P_u, because K is fixed.

Several different indexing schemata introduced recently in the literature (see Section 2.2.3) can be understood and analyzed with the common framework introduced here. The informal idea is to map the metric space to an alternate space, that is easier to search than the original space.

We call this alternate space the *K Nearest References space* (K-nr space). In the preprocessing stage, each database object is mapped to the K-nr space using a set of references. The query object is also mapped to the K-nr space and relevant objects are detected and reported for a subsequent verification.

Formally, our framework is composed by the tuple $(\mathsf{encode}, \mathsf{sim}, K, R)$:

- R denotes a finite subset of objects randomly selected from the database S (i.e., $R \subset S$) the *reference objects*, such that $|R| = \sigma \ll n$.
- K is the parameter to compute $\mathsf{K\text{-}nn}_{R,d}(u)$, i.e. the K-Nearest-Neighbors of u on R.
- encode is a function mapping an ordered set into an abstract set $\hat{U}$. The domain U is converted to the K-nr space as: $\hat{U} = \{\hat{u} = \mathsf{encode}(\mathsf{K\text{-}nn}_{R,d}(u)) \mid u \in U\}$. We can extend encode to a whole set of objects in the usual way.

- Finally, a similarity function $\mathsf{sim} : \hat{U} \times \hat{U} \to \Re$.

The composition of encode and sim need to be a contractive mapping, in other words proximity in the original domain U using $d(\cdot, \cdot)$ should imply proximity in the mapped domain $\hat{U}$ under $\mathsf{sim}(\cdot, \cdot)$. This procedure is usual on other metric techniques like pivots. The detailed analysis is provided by Chavez et al. [Chávez et al., 2001]. More or less, the same properties are followed by other approximate metric indexes [Chavez et al., 2008; Tellez et al., 2009, 2011b].

Since R is a finite set it can be well ordered as $R = r_1, r_2, \ldots, r_\sigma$ and we can identify the objects in R with the respective indexes $(r_1, r_2, \cdots, r_\sigma) \leftrightarrow (1, 2, \cdots, \sigma)$.

Within this framework a search is done by following this steps

1. Map the query to the K-nr domain.
2. Search for the closest γ candidates using sim in the mapped space.
3. Verify γ candidates using d in the original space.

The number of candidates γ is a parameter governing the ratio speed/recall. A large number of candidates increase the recall, and slow down the query time. This parameter is determined for each dataset and the desired number of neighbors (k-nn) for a particular task.

8.4 Describing Existing Proximity Indexes using K-nr

Our framework captures the behavior of at least four different indexes found in the literature, namely, the Permutation Index (PI) [Chavez et al., 2008], the Brief Permutation Index (BIF) [Tellez et al., 2009], the Metric Inverted File (MIF) [Amato and Savino, 2008], and the PP-Index [Esuli, 2009]. In the next paragraphs, we briefly describe those indexes.

8.4.1 Permutation Index (PI)

PI is the first K-nr index reported in the literature (see [Chavez et al., 2008]). The main idea behind the PI is to capture a descriptive perspective of the metric space. Instead of using the distances to a fixed set of pivots, as it it customary in some type of indexes, they care about the relative order of a reference set. The reference set is called the *set of permutants*. Every

database element is transformed into a permutation and proximity is hinted by a distance between permutations.

Within our framework we need to describe $(\mathsf{encode}, \mathsf{sim}, K, R)$ for this index. Here R is the set of permutants, $K = \sigma$, the encoding encode is some permutation and sim is a Minkowsky distance over the inverse of the permutation.

Let $R = \{x_1, x_2, \cdots, x_\sigma\}$. Let R_u the set R ordered by distance to $u \in U$ (with ties broken arbitrarily, but in a consistent order). This define a permutation of the indexes of the set R. This permutation will enconde u, hence $\mathsf{encode}(u) = \Pi_u = (i_1, i_2, \cdots, i_\sigma)$ and $\mathsf{sim}(\mathsf{encode}(u), \mathsf{encode}(v)) = ||\Pi_u^{-1} - \Pi_v^{-1}||_1$ will be the L_1 distance between the inverse of the respective permutations taken as integer vectors.

In order to analyze the complexity of PI, note that Π_u requires the computation of σ distances and then sorting the distances, this implies $O(\sigma)$ operations if the distances are integer, and $O(\sigma \log \sigma)$ if the distances are real (because a comparison based sort is needed). To obtain the candidate list we need to compute the distance $\mathsf{sim}(\hat{q}, \hat{u})$ for all u in the database, this will cost $O(n\sigma)$ arithmetic operations. Finally, the space complexity requires σ items to be represented. The required memory is of at least $n\sigma \log \sigma$ bits.

8.4.2 Brief Permutation Index (BPI)

This index is inspired by the PI above. The key difference is the encode and sim functions. Let $\Pi_u^{-1} = (i_1, i_2, \cdots, i_\sigma)$ be the inverse permutation of the indexes of the set $R = \{x_1, x_2, \cdots, x_\sigma\}$. The encoding of u returns a binary vector $(a_1, a_2, \ldots, a_\sigma)$ where

$$a_i = \begin{cases} 1 & |\Pi_u^{-1}[i] - i| \geq m \\ 0 & \text{otherwise} \end{cases}. \tag{8.1}$$

with m a tuning parameter [Tellez et al., 2009] controlling the performance of the index.

Finally, given that encode returns a binary vector a natural definition of sim is the Hamming distance between the respective encodings.

The complexity of coding an element is of the same order of PI, i.e., $O(n\sigma)$, the gain in speed comes from computing sim as a Hamming distance. Let us assume that w is the size of the computer word then the computational cost of each distance sim is $O(\frac{\sigma}{w/2})$ using an additional table

of size $2^{w/2}$ to pre-compute Hamming distance.[1] That is, the complexity of BPI is $O(n\frac{\sigma}{w/2})$.

The space usage footprint is very small: a single bit is produced by each reference. So, it requires $n\sigma$ bits for the entire mapping.

8.4.3 Metric Inverted File (MIF)

This index is also based on the PI. The algorithmic contribution of MIF [Amato and Savino, 2008; Chavez et al., 2008] consist in an improved method to compute an approximation to the L_1 distance originally used in the PI. This improvement allows to compute this distance approximation to *all* the database object representations $e(\mathsf{encode}(q), \mathsf{encode}(u))$ at once. To achieve this, only a fixed number (K) of references are used to compute the distance between object representations. The missing references will add a fixed penalty each one.

More precisely, from R we select the K closest objects to the query q, using $K \ll \sigma$. Then each one of this objects are located in all the permutations to compute the distance. Since not all pairs will be found when using only a subset of the permutations, a fixed penalty is added.

Let $(r_1, \ldots, r_K)$ the closest references to q. Let $\mathsf{encode}(u) = \Pi_u$ and $\mathsf{encode}(q) = \Pi_q$. The distance between representations is computed as $e(\mathsf{encode}(u), \mathsf{encode}(q)) = \sum a_i$ with a_i defined as follows

$$a_i = \begin{cases} |\Pi_u^{-1}[i] - \Pi_q^{-1}[i]| & \text{if } r_{a_i} \text{ belongs to the } K \text{ objects retrieved} \\ \sigma/2 & \text{otherwise} \end{cases}, \quad (8.2)$$

where r_{a_i} correspond to the a_i-th object in R.

The nice thing of this schema is that it can be implemented using an inverted file [Baeza-Yates and Ribeiro-Neto, 1999; Witten et al., 1999]. The set R is used as a *thesaurus* and list of tuples (a_i, i) or $(reference, position)$ are used as *posting lists* [Amato and Savino, 2008]. In this data structure, we need to store only the K tuples (a_i, i) such that $a_i \in \mathsf{K\text{-}nn}_{R,d}(u)$.

The computational cost to represent each object is equivalent to PI; however, in this case the plain mapping (without inverted index representation) requires $Kn \log K\sigma$ bits i.e., each object requires K tuples of $(reference, position)$. These tuples are sorted by $reference$ (adding an additional sort over K items). The candidate list requires n evaluations of

[1] When $w = 32$ this scheme produces tables of 2^{16} entries of 5 bits each one, i.e. $\log(w/2 + 1)$ bits in general. For very large w one needs to divide w in smaller pieces.

the partial Spearman Footrule distance which requires in the worst case $O(K)$ and in the best scenario it takes $O(1)$ time. Please note that when the objects are not related the best scenario (or something close to it) is frequently found. This yields a worst case of $O(nK)$ and a smaller average worst case for obtaining the candidate list. Using an inverted index, requires $Kn \log(Kn)$ bits of space, and the total cost is driven by the cost to obtain the candidate list and $O(\gamma)$ distance computations.

8.4.4 Prefix Permutations Index (PP-Index)

The encode function used in PP-index [Esuli, 2009] is equivalent to the one used in PI; however, here one treats the result of encode as a string with a word size of K. Here sim measures the similarity using the length of the shared prefix, such that a long shared prefix means high proximity and a zero length prefix reflects low proximity.

This stringent notion of closeness yields to low recall (ranging from 0.3 to 0.5) [Esuli, 2009]. In contrast, it is really fast and can be represented efficiently using a compressed trie data structure [Baeza-Yates and Ribeiro-Neto, 1999; Witten et al., 1999].

The computational cost of this index is similar to MIF; however, the constants here are smaller and the best scenario (for sim) is even more frequent in the PP-Index than in the Metric Inverted File.

8.5 Using the K-nr Framework to Create Proximity Indexes

The use of K-nr framework to describe the existing proximity indexes allows to highlight the difference between these indexes. As mentioned previously, these difference can be easily explained in terms of R, encode, sim and K. Furthermore, it is evident that only a few combinations of the parameters have been explored so far, once detected the correct parametrization of the algorithms a very large number of alternatives can be discovered. In this section, we start filling this gap by proposing a number of novel proximity indexes.

Function encode maps each element of the database to another data model (e.g., the PI transform elements into vectors). Given this characteristic, we decided to group the new proximity indexes according to the data model of the mapped space. So far, we have seen that encode maps to vectors (e.g., PI) and to strings (e.g., PP-Index). In these data models,

we tested the behavior of the index when different sim and small variations over encode are used.

Both vectors and strings are ordered sets. One natural generalization is to use sets for the encode function. This data model will be tested as well in the experimental section. Below we give the details of the new proximity indexes.

8.5.1 Vectors

For vectors we tried two different distance functions: the first one sim equals to Spearman ρ (L_2 over the inverses) for *prefixes* of permutations, and the second sim is the cosine between the σ-dimensional vectors.

The cosine similarity is evaluated as:

$$\mathsf{sim}_C(u,v) = \frac{\sum_{i=1}^{\sigma} s(r_i,u)s(r_i,v)}{\sqrt{\sum_{i=1}^{\sigma}(s(r_i,u))_i^2} \times \sqrt{\sum_{i=1}^{\sigma}(s(r_i,v))_i^2}} \tag{8.3}$$

Where d_{max} is the maximum (possible or found) distance, r_i the i-th object of R, and $s(a,b) = 1 - d(a,b)/d_{max}$.

Hence in K-nr cosine, encode(u) stores the sparse vector $\hat{u}$, defined by the tuples: $\hat{u} = \{(i, 1-d(r_i,u)/d_{max}) \mid r_i \in \mathsf{K\text{-}nn}_{R,d}(u)\}$, which can be implemented easily with an inverted index implementing the TF-IDF model [Baeza-Yates and Ribeiro-Neto, 1999]. When the references are not in the sparse vectors they are represented as zero values and can be skipped from computation. This allows s to be implemented quite fast using a inverted index and set union algorithms.

The time complexity is on the same order of magnitude than Spearman Footrule; but the space complexity is $nK(\log\sigma + f_d)$ bits, where f_d is the number of bits required by floating point numbers. Using an inverted index increases the required space to $nK(\log n + w)$ bits.

8.5.2 Strings

We use two distance functions on the mapping build by PP-Index's encode function. These two distance functions are the *Levenshtein* (edit distance) and the Longest Common Subsequence (*LCS*).

The edit distance between two strings a and b is the minimum number of *edit* operations (insertion, deletion, or substitution of a single symbol at a time) needed to transform a into b (or viceversa, since its symmetric). The *LCS* is defined as the length of the longest common subsequence between a and b and can be found by allowing only insertions and deletions

as edit operations on Levenshtein's distance. Both distances are computed using dynamic programming, so the complexity of computing any of them is $O(K^2)$. We refer the reader to [Navarro, 2001] for a complete description of these distances. The space complexity is $nK \log \sigma$.

8.5.3 Sets

Here we push the K-nr idea even further by eliminating the requirement of sequences being in the same order. Proximity will be hinted simply by the number of shared references. This is at the same time more simple and more efficiently computed. Each object in the database will be represented by the set of its nearest references, among a relatively large set to choose from. We will use the unordered $\mathsf{K\text{-}nn}_{R,d}(u)$ as the set representing u.

In this space, we decided to test three different sim functions: Jaccard distance, Dice coefficient and the cardinality of the intersection. Let $\hat{u}$ and $\hat{v}$ two sets, Jaccard is defined as $d_J(\hat{u}, \hat{v}) = 1 - |\hat{u} \cap \hat{v}|/|\hat{u} \cup \hat{v}|$, this distance gives values between $[0, 1]$ where 0 means equality and 1 means disjointness. Dice coefficient is computed as $d_D(\hat{u}, \hat{v}) = 2|\hat{u} \cap \hat{v}|/(|\hat{u}| + |\hat{v}|)$ where a zero value means disjointness. It is important to note that both Jaccard distance and Dice coefficient are used in many information retrieval tasks [Grossman and Frieder, 2004; Baeza-Yates and Ribeiro-Neto, 1999].

The computational cost is equivalent to MIF; however, this representation requires simpler operations since the only operation needed is the union. The space complexity is smaller too, it is $O(nK \log(\sigma))$ bits.

8.6 Indexing K-nr Sequences

The indexes described above need a proper implementation to be useful. In this section we describe the details needed for achieving that. A data structure must sit behind computing $\mathsf{sim}(\mathsf{encode}(u), \mathsf{encode}(q))$ for all u. A sequential computation of every pair (u, q) does not scale on n, so, we require an index to skip unnecessary computations. In the same way, we care about the amount of space used to store the index.

If the set of references R is seen as an alphabet then $\hat{u}$ can be considered as a string. Let T be the concatenation of all K-nr sequences, i.e. $T = \hat{u}_1 \hat{u}_2 \cdots \hat{u}_n$. Each $\hat{u}_i \in R^K$ hence $T \in R^N$ with $N = Kn$. A plain representation of T needs $N \log \sigma$ bits. Using an unique codeword for each symbol, T can be represented optimally using $H_0(T) = \sum_{c \in R} \frac{N_c}{N} \log \frac{N}{N_c}$ bits, where N_c is the number of occurrences of c in T. This is the entropy of order zero of T, as typically defined for general sequences [Navarro and Mäkinen,

2007]. The result is that T requires at least $NH_0(T)$ bits to be stored. In the worst case, when a uniform distribution is found, i.e. $N_c = N/\sigma$, $nH_0(T) = N \log \sigma$ bits.

A sequence can be represented in close to optimal space and with high performance operations with a Index of Sequences (IoS). An extensive list of IoS are reviewed in Appendix A Appendix A. The primitives of an IoS are the following:

- $\mathsf{Rank}_c(T, pos)$ counts how many c's occurs in T until pos, $c \in \Sigma$.
- $\mathsf{Select}_c(T, r)$ returns the smaller position pos such that $\mathsf{Rank}_c(T, pos) = r$.
- $\mathsf{Access}(T, pos)$ retrieves the symbol stored at the position pos in T.

Using this primitives we are able to implement the majority of the K-nr searching algorithms.

8.6.1 Algorithms

Let $P = p_1 \cdots p_m$ be a sequence, $m \leq K$, the following algorithms are considered as basic operations in our K-nr procedures:

— $\mathsf{FindSubString}(T, P)$. Retrieves all object identifiers, as K-nr representation, containing P as substring. We change the substring problem for a conjunctive query as follows. Let $L_{i,c}$ be a sorted list, where each item is defined as $L_{i,c}[j] = \mathsf{Select}(T, c, j) - i + 1$. The problem is a conjunctive query, i.e., $L_{1,p_1} \cap L_{2,p_2} \cap \cdots \cap L_{K,p_K}$. Recently many fast solutions to the set intersection problem have been studied, the interested reader is referred to the recent surveys [Culpepper and Moffat, 2010; Barbay et al., 2009; Baeza-Yates and Salinger, 2010]. Notice that if P is located between two sequences, the occurrence must be ignored, hence false positives will not appear.

— $\mathsf{Count}(T, c)$ or N_c. It returns the number of occurrences of the symbol c in T. It is computed as $\mathsf{Rank}(T, c, N)$.

MIF, K-nr Spearman Footrule and ρ

Algorithm 11 computes the K-nr Spearman Footrule (with an straightforward adaption to Spearman ρ). Here the accumulator A incrementally computes all partial Spearman Footrule distances, using D^+ values. MIF is

algorithm 11: K-nr Spearman Footrule / MIF prefixes algorithm with an IoS

Input: The query $\mathsf{k\text{-}nn}_{S,d}(q)$, T the sequence representing $\hat{S}$, ω the penalty constant, and the number of candidates γ.
Output: The set of candidate objects $\mathcal{C}$.

1: Let $\hat{q} = \mathsf{K\text{-}nr}(q)$.
2: Let $\mathcal{C} \leftarrow \emptyset$.
3: Let A be a hash table storing pairs (object identifiers, accumulated distance).
4: **for all** $r \in \hat{q}$ **do**
5: **for all** $s = 1$ to $\mathsf{Count}(T, r)$ **do**
6: Define $pos = \mathsf{Select}_r(T, s)$
7: Define $\mathsf{objID} = pos/K$
8: **if** A contains key objID **then**
9: Define $D^- = K\sigma\omega$
10: **else**
11: Define $D^- = A[\mathsf{objID}]$
12: **end if**
13: Define $D^+ = |r - (pos \mod K) + 1|$ {+1 since sequences are starting on index 1}
14: $A[\mathsf{objID}] \leftarrow D^- + D^+ - \sigma\omega$
15: **end for**
16: **end for**
17: Linearly sort A by value, and store the γ top candidates on $\mathcal{C}$.

implemented with an explicit inverted file, so it requires at least $Kn \log nK$ bits, and it is implemented padding $\log n$ to the machine word (32 or 64 bits), and $\log K$ to a 16 bit integer. In contrast, our sequence based index requires space close to $Kn \log \sigma$ bits (plus some small order terms). The time complexity is $\sum_{r \in \hat{q}} \mathsf{Count}(T, r)$ plus the γ distance computations.

PP-Index and K-nr Prefixes

algorithm 12: K-nr prefixes / PP-Index algorithm over the IoS

Input: The query $\mathsf{k\text{-}nn}_{S,d}(q)$, T the sequence representing $\hat{S}$, and the number of candidates γ.
Output: The set of candidate objects $\mathcal{C}$.

1: Let $\hat{q} = \mathsf{K\text{-}nr}(q)$
2: Let Q be the list of all prefixes of $\hat{q}$ from shortest to the largest one i.e. $\hat{q}_1, \hat{q}_1\hat{q}_2, \ldots, \hat{q}_1 \cdots \hat{q}_K$
3: Let $\mathcal{C}$ and $\mathcal{C}'$ starts as two empty sets.
4: **for all** $p \in Q$ **do**
5: $\quad \mathcal{C}' \leftarrow \mathsf{FindSubString}(T, p)$
6: $\quad$ Delete all items $s \in \mathcal{C}'$ such that $s \mod K \neq 1$
7: $\quad$ **if** $|\mathcal{C}'| < \gamma$ **then**
8: $\quad\quad$ Break the loop and let $\mathcal{C} \leftarrow \mathcal{C}'$ if $\mathcal{C}$ is empty.
9: $\quad$ **end if**
10: $\quad \mathcal{C} \leftarrow \mathcal{C}'$
11: **end for**

Note: After the first iteration $\mathcal{C}$ contains the result of the previous intersections, thus $\mathcal{C}$ should be used by FindSubString instead of recompute the intersections.

After the ith call of FindSubString in Algorithm 12, $\mathcal{C}$ contains the result for the $i-1$th call, so we can speed up the computation of FindSubString computing the intersection of the new symbol in p using the previous $\mathcal{C}$. With this small change, the cost is the same that computing the intersection of $K-1$ pairs, i.e. $(\cdots((L_{1,\hat{q}_1} \cap L_{2,\hat{q}_2}) \cap L_{3,\hat{q}_3}) \cap \cdots) \cap \text{Ł}_{K,\hat{q}_K}$. In the worst case, all $\mathsf{Count}(T, q_i)$ are equal, $\mathsf{Count}(T, q_i) \simeq N/\sigma$, and the intersection does not reduce the cardinality of $\mathcal{C}$, so the number of comparisons to find $\mathcal{C}$ is $(K-1) \cdot O\left(\log \binom{2N/\sigma}{N/\sigma}\right) = O(K^2 n/\sigma)$. In our IoS implementation we actually should care about the number of calls to Select, nevertheless this number is closely related to the number of comparisons.

Set K-nr

Our indexes support set theoretic operations, i.e. union and intersection of $\hat{q}$ and $\hat{u}$, with the sequence representing a set collection. Defining G_c as $G_c[j] = \lceil \mathsf{Select}(T, c, j)/K \rceil$ we can translate de problem to set union. Since we only care for counting the cardinality of the intersection $|\hat{q} \cap \hat{u}|$, it is enough to compute the cardinality of the union in our G_c lists because $|\hat{q} \cup \hat{u}| = |\hat{q}| + |\hat{u}| - |\hat{q} \cap \hat{u}|$, and $|\hat{q}| = |\hat{u}| = K$ is fixed.

Counting union/intersection cardinality can be used as a filter, but actually computing them requires to visit all the items in the list. This coincide with the worst case complexity of the intersection. For this reason we use a simple dictionary based procedure. Algorithm 13 shows the basis for the all set K-nr methods, which is counting the size of $|\hat{q} \cap \hat{u}|$, which in turn consist in counting the union using G_c, for all $c \in R$, in the representation of T.

---

algorithm 13: Procedure to retrieve at least γ (if available) promising objects under the intersection cardinality

---

Input: The query $\mathsf{k\text{-}nn}_{S,d}(q)$, T the sequence representing $\hat{S}$, and γ (the number of candidates).
Output: The set of candidate objects $\mathcal{C}$.
Pseudocode:

1: Let $Q[1, K]$ be a list of empty sets
2: Initialize $\mathcal{C}$ as an empty set.
3: Let H be an empty dictionary mapping object identifiers to an integer accumulator. Undefined keys are mapped to 0.
4: Compute $\hat{q}$ as $\mathsf{K\text{-}nr}(q)$
5: **for all** $p \in \hat{q}$ **do**
6: **for all** $s \in G_p$ **do**
7: $H[s] \leftarrow H[s] + 1$
8: **end for**
9: **end for**
10: **for all** $(docid, cardinality) \in H$ **do**
11: $Q[cardinality] \leftarrow Q[cardinality] \cup \{docid\}$
12: **end for**
13: **for all** $cardinality = K$ down to 1 **do**
14: $\mathcal{C} \leftarrow \mathcal{C} \cup Q[cardinality]$
15: **break** the loop **if** $|\mathcal{C}| \geq \gamma$
16: **end for**

K-nr Levenshtein and K-nr LCS

Here the problem is to find all the (sequences representing) u which are at most κ edit operations from the representation of q. It is possible to filter the collection by using set operations, the number of shared references serve as a bound to the number of edit operations. The basic procedure is similar to Algorithm 13.

Let $ED(\hat{q}, \hat{u})$ be the edit distance between the representations of q and u respectively. We want to retrieve every u such that $ED(\hat{q}, \hat{u}) \leq \kappa$. This implies $|\hat{q} \cap \hat{u}| - |\hat{q} \cup \hat{u}| \leq 2\kappa$, and hence objects violating this condition can be excluded from comparison. In practice this procedure saves a very large number of edit distance operations. This is a costly operation, since each candidate sequence is retrieved using K calls to Access.

A fast searching algorithm for the list of candidates $\mathcal{C}$ requires $\kappa < K$, i.e. $(|\hat{q} \cap \hat{u}| - |\hat{q} \cup \hat{u}|)/2 \leq \kappa < K$. If we allow $\kappa = K$, then $\hat{u}$ will be candidate even if $|\hat{q} \cap \hat{u}| = 0$. Supporting equality of K makes sense if we want to approximate $ED(\cdot, \cdot)$, but this implies considering non intersecting sequences. In other words, we want to avoid giving unrelated sequences the same rank. Take for example accepting $ED(01234, 34567) = 5$, and rejecting $ED(01234, 56789) = 5$. Even when both pairs of sequences have the same edit distance, the former pair is more likely to be related. The same arguments are valid for LCS, providing a good filter for both methods.

8.6.2 Final Notes on Creating K-nr Indexes

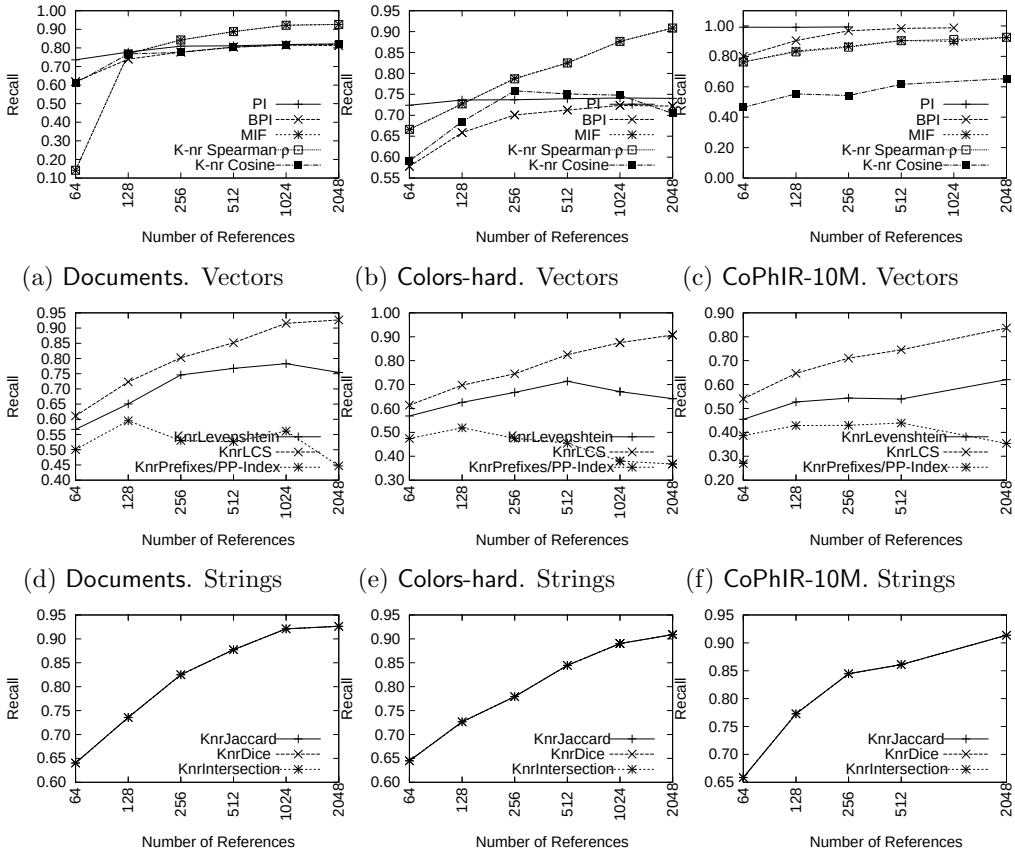
It is quite tempting to extend string and set K-nr methods to other combinations of substring matching and proximity indicators. This exercise will be excluded from this thesis because of the combinatorial explosion in considering all the options in the experimental part. We will show instead in the experimental section, the faster variants of the techniques.

8.7 Experimental Results

In order to study the behavior of the K-nr indexes we performed a serie of experiments using as benchmarks three real-world datasets (Documents, Colors-hard, and CoPhIR-10M), and six randomly created databases (RVEC-*-1000000). For a complete description of our datasets, please refer to Section 1.3.3. The running hardware and implementation guidelines follows the ones described in Section 1.3. We use 30 nearest neighbors because it is a common value as an output in a multimedia information retrieval sys-

tem. The entire databases and indexes are maintained in main memory and without exploiting any parallel capabilities of the workstation.

8.7.1 Quality of the Results



(a) Documents. Vectors (b) Colors-hard. Vectors (c) CoPhIR-10M. Vectors

(d) Documents. Strings (e) Colors-hard. Strings (f) CoPhIR-10M. Strings

(g) Documents. Sets (h) Colors-hard. Sets (i) CoPhIR-10M. Sets

Figure 8.2: Recall performance of K-nr mappings

Figure 8.2 shows the recall rate when the number of references is varied. The figure presents three K-nr mappings: vector, string and set mappings (rows) on our three datasets (columns). As we can see from Figure 8.2c the PI and the BPI have a perfect recall for a small σ. The other indexes performed below these two. On the other hand, for Documents dataset and Colors-hard, the lowest performance is presented by the PI and the BPI

(Figure 8.2a). This behavior is consequence of the high concentration around the mean for Documents and Colors-hard datasets, as seen in Figure 1.1. On the other side MIF and Spearman ρ are quite good on all datasets. Remarkably, the recall increases as σ grows.

String mappings (Figures 8.2d to 8.2f) expose a variety of performances. That is, K-nr *LCS* has the highest recall rate in both datasets, closely followed by K-nr *Levenshtein*. The worst performance is obtained by K-nr prefixes, note that for this method the recall get worst as the number of references increases. The same tendency is obtained in the three datasets.

In the set mappings, Figures 8.2g, 8.2h, and 8.2i, all the indexes have an almost identical recall rate. This is an indicator that the set mapping is less sensitive to the distance measured used, so, it can be used as default method on unknown setups (when details of datasets and distances are unknown).

It can also be observed that K-nr set methods need larger values of σ to achieve its optimal value. This is a good characteristic, because larger σ means faster indexes (remember the algorithmic problem, Kn items are divided on σ symbols), Section 8.6. Also, σ should be smaller than n ($\sigma \ll n$), because computing the K-nr sequence has a cost of σ distance computations, and R can be stored in RAM.

Furthermore, the recall rate for all the K-nr set methods (when $\sigma = 2048$) and all the datasets is above 0.90. Comparing these recall rates against the string mapping, one can note that the set mapping obtains better recall rates in the 3 datasets tested and equals MIF and Spearman ρ on all setups. The only dataset where the set mapping got a lower recall rate was in CoPhIR-10M and only against the PI and BPI which obtained a perfect recall rate.

8.7.2 Size of the Indexes

The memory requirement is diverse for all the indexes as shown by Figure 8.3. This figure is in log-log scale to appreciate differences between large indexes, such as PI and BPI, and smaller ones as K-nr strings and sets. So, indexes with small overheads by object are quite good to be used in large databases.

Figure 8.3, show *plain* (simple mappings without an index), *inv. index* (implementations using an inverted index) and K-nr Strings, Sets, MIF, and PP-Index are built over an index of sequences (IoS, see Section 8.6). From Figure 8.3 it can be observed that the better indexes in terms of memory usage are the set based mappings. We did not include the figures for the other datasets because they exhibit a similar behavior, the only difference is that range is scaled to the size of the problem.

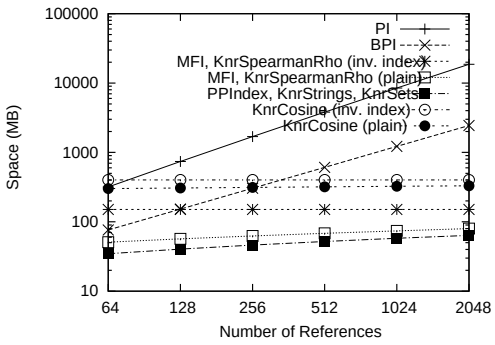


Figure 8.3: Memory requirements for the indexes for the CoPhIR-10M objects.

8.7.3 Enhancements to the K-nr Indexes

In this section, we present some enhancements to the indexes to increment the recall rate (see Subsection 8.7.3). However this will increase the query time and storage size. In order to provide a complete picture of this procedure, Subsection 8.7.3 presents an analysis of the parameters used to control the tradeoff between recall and time.

Increasing Recall Using Several K-nr Indexes

A general technique to increase the recall rate in K-nr methods is to use several indexes, as reported by [Esuli, 2009; Amato and Savino, 2008]. Each index will have independent results and hence the recall can only increase. The downside is the increase in query time (partial results should be joined) and the total amount of spent time. We analyzed a simplified version of this procedure as follows.

Firstly, we build the indexes using different $\sigma \in \{64, 128, \ldots, 2048\}$. Then for each different value of σ the resultant of the query is the union of the results of performing that particular query to the index having equal or less σ. For example, the index with $\sigma = 512$ returns the union of subset computed by the indexes working with σ equals to 64, 128, 256, and 512. This will be the *cummulative* recall.

Figure 8.4 presents the recall of the traditional indexes and the cummulative recall. We decided to include the recall of the bare bone indexes to simplify the comparison. Figures on the left side, 8.4a, 8.4c, and 8.4e are

presenting the *base* recall. Figures on the right side, 8.4b, 8.4d, and 8.4f, are showing the cumulative recalls.

In Figure 8.4 it is observed that at $\sigma = 64$ the standard index and the corresponding cumulative index have the same recall rate. The difference between the recalls is increased when σ is incremented, in other words when the number of indexes involved in the final result is incremented. A particular large improvement is found for K-nr Prefixes (PP-Index), which improves dramatically the recall, transforming the index into an appealing option for high quality requirements. Other indexes present a gain of $5 - 15\%$, which is still very important.

The same strategy is valid to speed up searches, using a partition of the database (a disjoint collection of subsets) across several searching servers. As usual, hybrid approaches can be used to achieve both recall and speed enhancements.

Improvements with Parameters K and γ

It is expected that the use of several indexes to resolve a particular query increases the time and memory complexities. Another option to tune the tradeoff between time and recall is the optimization of the parameters K and γ.

The parameter K appears in both preprocessing and searching steps. At building time, K modifies the size of the index, so increasing K increases N (see Section 8.6). On the other side, increasing K on the searching step increases the recall without growing the index size and using the same index. The cost of searching-K shows up in increasing the number of symbols to be processed in our IoS, which impacts the real time used for searching. Due to these characteristics, our experiments are focused on searching-K.[2]

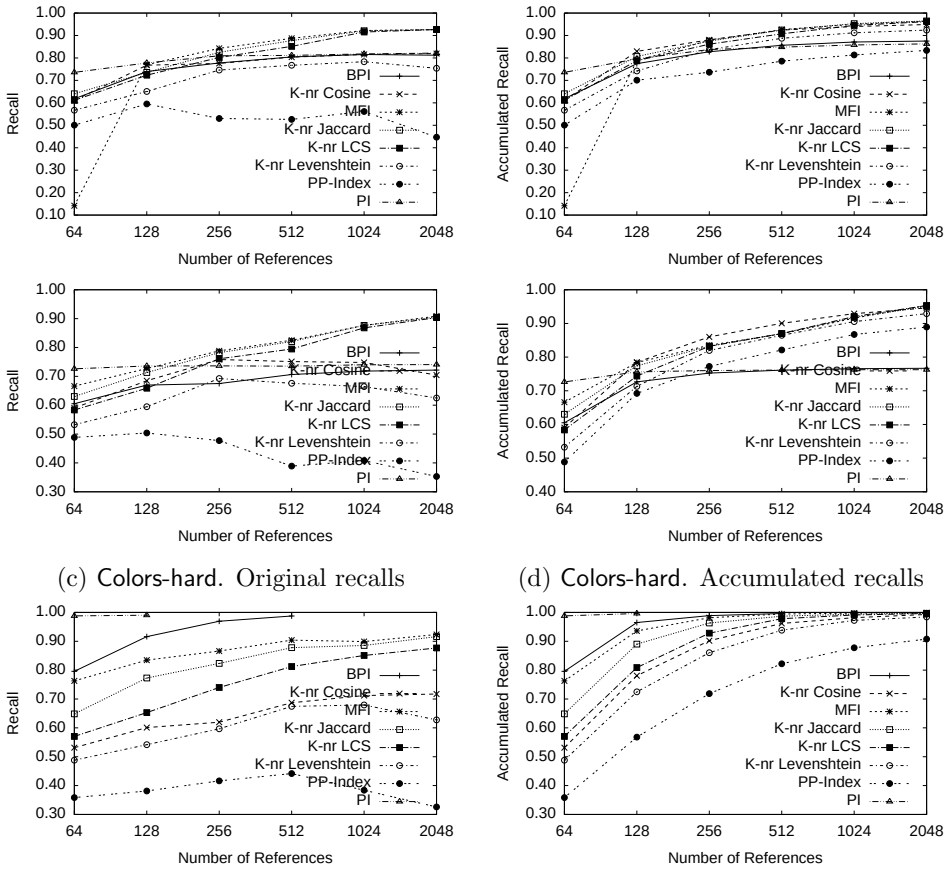
In this experiment we used an index of sequences computing the K-nr Jaccard method (see Section 8.5). The construction of the index uses seven nearest references ($K = 7$). The selection of this particular index of sequences is important since we present real time results along with the recall in the CoPhIR-10M database (which is a large database).

Figure 8.5 shows the behavior for K-nr Jaccard for our three datasets, using several γ and K values. Those figures present the time and the recall rate for each configuration of K and γ. It is observed that the recall rate is high for large values of K even with moderate γ values. On the other hand, for high values of γ we have a recall rate close to 0.9 when $K = 6$.

[2]Increasing searching-K is meaningless for K-nr prefixes, permutations and brief permutations.

(a) Documents. Original recalls

(b) Documents. Accumulated recalls



(c) Colors-hard. Original recalls

(d) Colors-hard. Accumulated recalls

(e) CoPhIR-10M. Original recalls

(f) CoPhIR-10M. Accumulated recalls

Figure 8.4: Joining K-nr Documents and Colors-hard, searching 30-nn. The *accumulated* curves uses the *union* of current and smaller σ results.

For intermediate values of γ the recall rate is rapidly stabilized and gives low increments for larger γ.

From Figure 8.5 one can infer a particular set of parameters working well in terms of recall rate and time. For all the datasets the best recall rate is obtained with $\sigma = 2048$ and an adhoc γ for each case. For the case of Documents an adequate value for γ is 400, for the case of Colors-hard is 1000, and for CoPhIR-10M is 15000.

Taking into consideration the recall rate and the time, the experiments above suggest that large K values imply higher cost than increasing the number of candidates (specially for large databases as CoPhIR), i.e. there are more inverted lists (and objects) to compute the union operation. For smaller databases, this problem is not noticeable. The tradeoff should be found taking into account the cost of the distance function, the speed of union algorithm, and the size of the database (i.e., n).

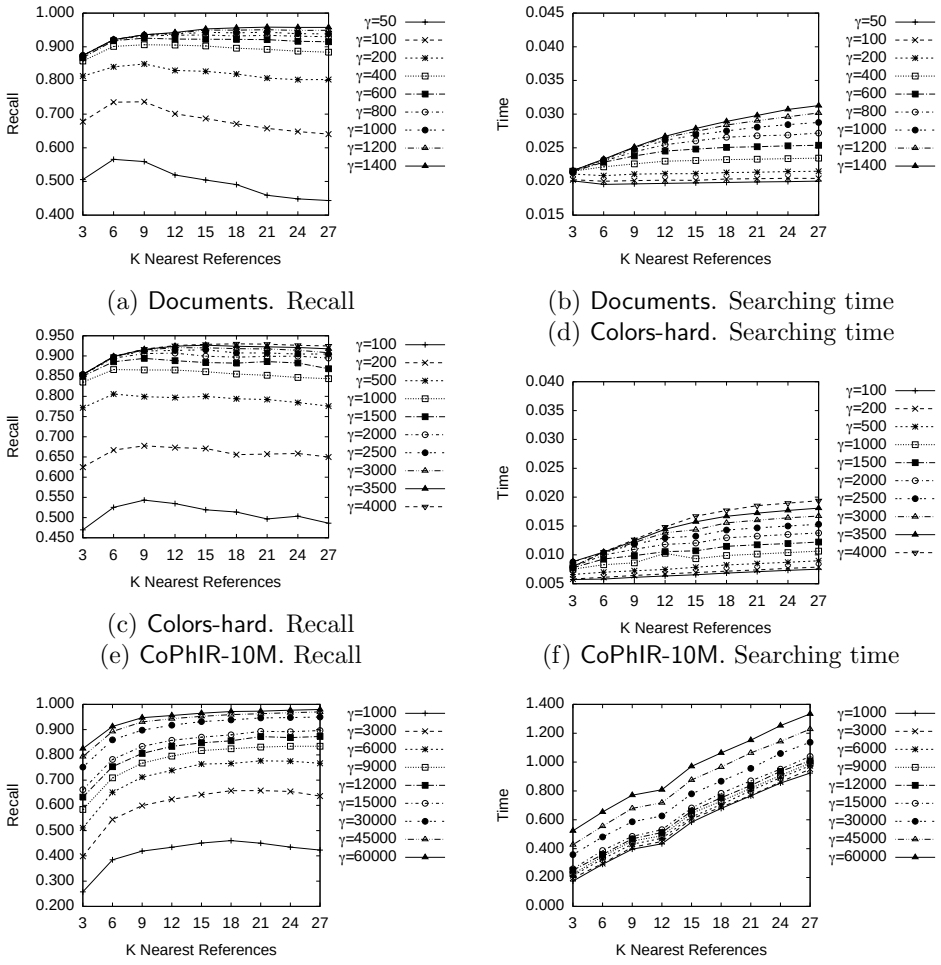
8.7.4 Searching with Sequence Indexes

In this section, we test the performance of our implementation based *only* with an indexed sequence. Here, we avoid the comparison of K-nr Levenshtein and LCS, since both are complicated to implement and do not surpass the performance of the K-nr set. The same case arises for BPI, PI, and K-nr cosine, because they need a more complex representation.

Figure 8.6 shows the searching time required to obtain a particular recall. The dataset Documents is tested on Figure 8.6a, here the majority of the indexes (excepting for PP-Index) are showing more than 90% of recall for points with more than 512 references. The main performance change among these points is on the searching time, however those differences are negligible. A similar behavior is found for Colors-hard and CoPhIR-10M, yet searching time differences are more noticeable, exposing the simplicity of K-nr Jaccard. In our figures, excepting for PP-Index, all techniques increase its recall and decreases the searching time as σ grows. Naturally, this behavior should be reversed on some point, not reached point yet.

Even on PP-Index, a lesser strict quality measure as the proximity ratio (see Section 1.3) has a good performance relative to the searching time. Almost points are quite good and close to 1. The reached percentiles are quite close as shown by the histograms of Figure 1.1. The proximity ratio becomes closer to 1 as σ grows. Remarkably, the searching time decreases, as well. In contrast, PP-Index barely changes on the proximity ratio.

The performance is directly proportional to σ and consequently in our sequence representation to the storage cost, as showin in Figure 8.8. Again the



(a) Documents. Recall

(b) Documents. Searching time

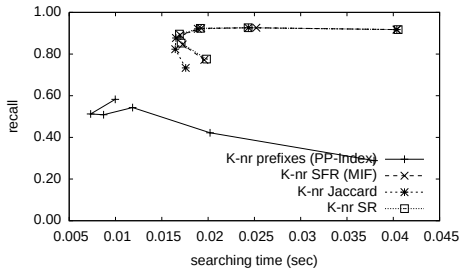
(c) Colors-hard. Recall

(d) Colors-hard. Searching time

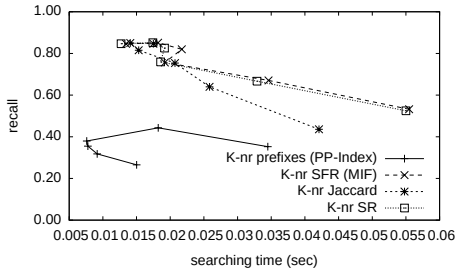
(e) CoPhIR-10M. Recall

(f) CoPhIR-10M. Searching time

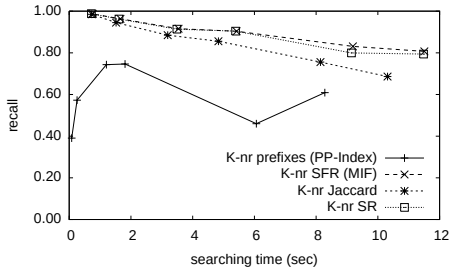
Figure 8.5: Recall and time performances for different γ and searching-K values. The building configuration is $\sigma = 2048$ and $K = 7$.



(a) Documents. $\gamma = 1000$.

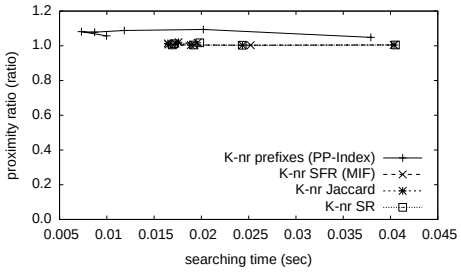


(b) Colors-hard. $\gamma = 3000$.

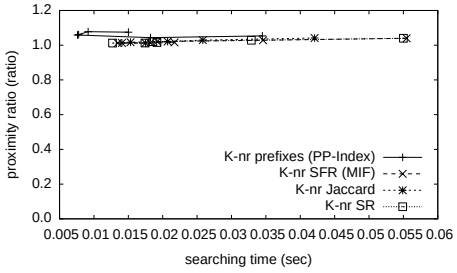


(c) CoPhIR-10M. $\gamma = 60000$.

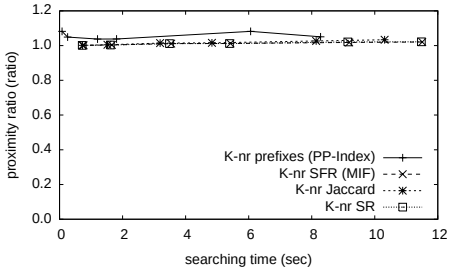
Figure 8.6: Comparison between recall and searching time on our real-world datasets. Note that σ grows in points from right to left, such that, as *sigma* grows most K-nr indexes increment their recall and reduce their searching time.



(a) Documents. $\gamma = 1000$.

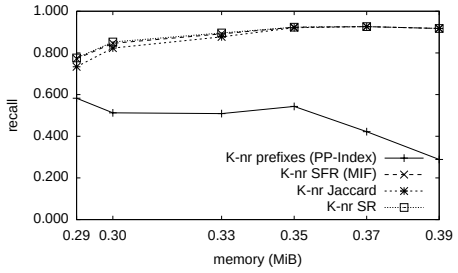


(b) Colors-hard. $\gamma = 3000$.

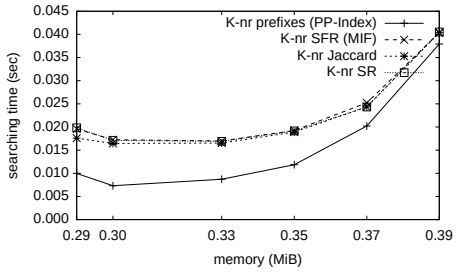


(c) CoPhIR-10M. $\gamma = 60000$.

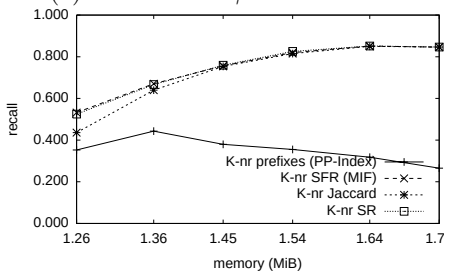
Figure 8.7: Comparison between proximity-ratio and searching time on our real-world datasets



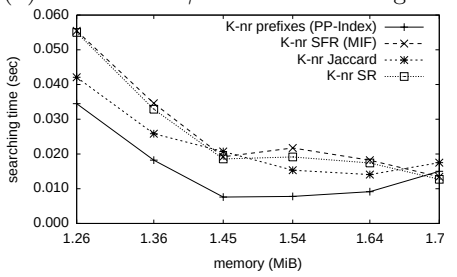
(a) Documents. $\gamma = 1000$. Recall



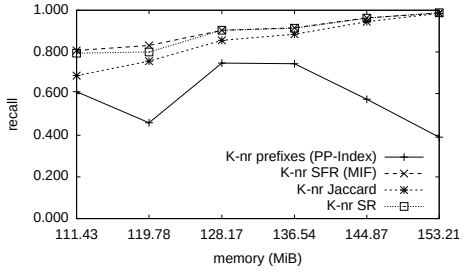
(b) Documents. $\gamma = 1000$. Searching time



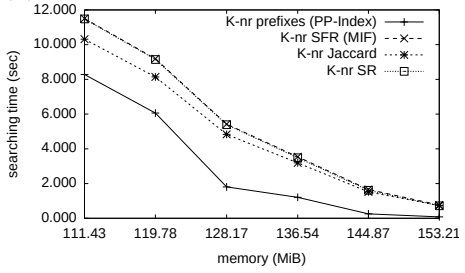
(c) Colors-hard. $\gamma = 3000$. Recall



(d) Colors-hard. $\gamma = 3000$. Searching time



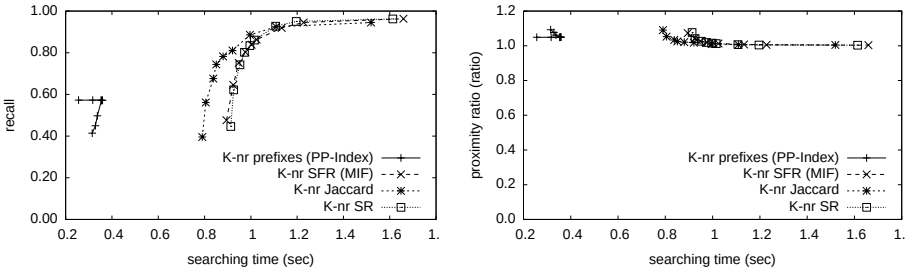
(e) CoPhIR-10M. $\gamma = 60000$. Recall



(f) CoPhIR-10M. $\gamma = 60000$. Searching time

Figure 8.8: Comparison among recall, searching time, and memory requirements on our real-world datasets. σ grows from right to left (one point per σ value on the curves).

exception is found for PP-Index since its optimal value resides on $\sigma = 128$. The proximity ratio is not shown since it barely varies (Figure 8.7). In all cases, our indexes require close to an half of an uncompressed index (compression ratio is close to 50%, and much smaller for the Spearman Footrule (MIF) and Spearman ρ) since they require to store the tuple (objID, *position*) on the inverted index.



(a) CoPhIR-10M. Recall vs time (b) CoPhIR-10M. Proximity ratio vs time

Figure 8.9: Performance on CoPhIR-10M with $\sigma = 2048$. Each point corresponds to γ in $1000, 3000, 6000, 9000, 12000, 15000, 30000, 45000, 60000$, appearing in increasing order from left to right.

The quality of the result set parametrized to the searching time is depicted in Figure 8.9 for CoPhIR-10M with $\sigma = 2048$. Each point corresponds to the maximum number of evaluations of the distance function in the last searching step. For recall, Figure 8.9a, all indexes hold a monotonic growth as γ grows, excepting for PP-Index. This is an indicator that PP-Index cannot found more candidates. The proximity ratio, Figure 8.9b, has a similar behavior, yet all indexes perform well under this quality measure. So, high quality results (in proximity ratio terms) can be achieved performing just a few distance computations. Remarkably, this behavior is enough for most real-world applications.

8.7.5 The Performance on Increasing Intrinsic Dimensions

In the end, Neighborhood Approximation is based on the order induced by distances, and as all techniques based on computing distances, it has a tight relation with the intrinsic dimensionality. This relation is exposed in Figure 8.10. We show results for RVEC-*-1000000 datasets, Section 1.3.3. We experimentally verify for $n = 250000$ and 500000, that the quality of results is barely affected (yet the searching time has an important relation to n, but we already learn this on previous experiments). The maximum number of

computed distances was fixed to $\gamma = 5000$, since it is a value contrasting our quality measures. We can observe that the recall rapidly degrades as dimension grows, going from 90% (dimension 4) to 60% (dimension 24). Please notice that, excepting PP-Index, the searching time decreases and the recall increases as σ grows. On the other side, we observe that proximity ratio barely changes its performance, this enforces our previous conclusions on fixed dimensionality (and Observation 2). Finally, a better performance on recall can be obtained applying the techniques of Sections 8.7.4 and 8.7.3.

8.8 Summary and Perspectives

In this chapter, we presented a novel framework for approximate proximity search algorithms called Neighborhood Approximation (NAPP), and its practical variant dubbed as K Nearest References (K-nr). This framework consists in mapping the original proximity problem in a general metric space to a simpler representation using K-nn queries in a set of references R.

Our framework allow us to analyze disparate proximity indexes such as: Permutation Index, Brief Permutation Index, and Metric Inverted File. We also used the framework to propose and test several novel indexes. Also, we proposed using plain, unordered collection of sets as universal representations of arbitrary metric spaces. This idea has not been used before in the literature, up to the best of our knowledge.

We also proposed and tested several mechanism to accelerate the proximity searching. We implemented the majority of the K-nr indexes using *indexes of sequences*, achieving close to optimal space and very competitive searching times. In addition, we investigated how varying γ and searching-K increase the recall rate without increasing the index size, yet impacting the query speed.

On the final part of this chapter we study the relation between the performance of the K-nr indexes and the intrinsic dimension of the dataset and query set. We conclude that even when the recall is affected on growing dimension, the quality measured with the proximity ratio is much more robust, being of interest to the majority of real world applications.

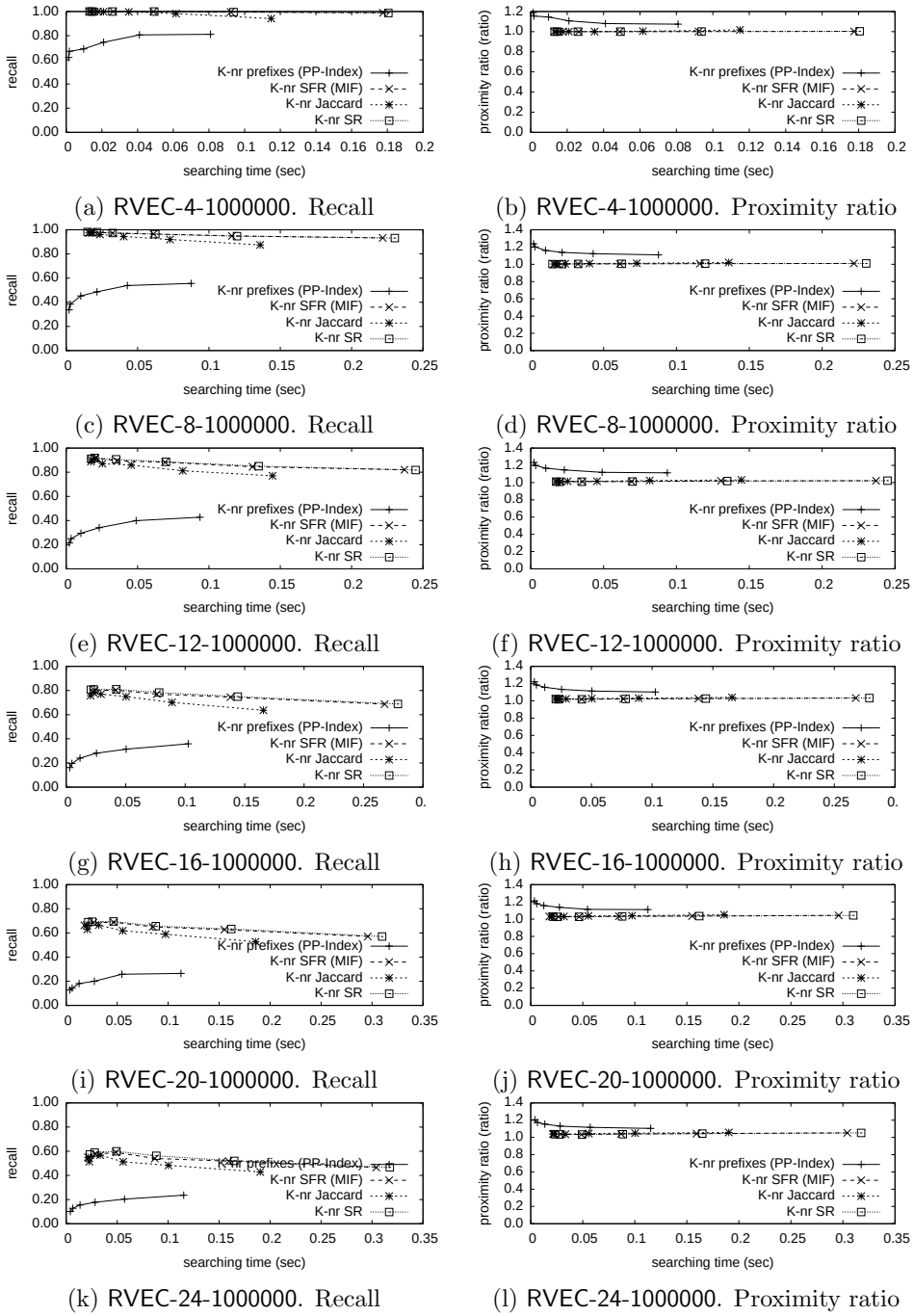
Despite the searching improvements achievements, the method still present several disadvantages, mostly in the construction of the K-nr indexes. In our K-nr indexes, the construction time is dominated by the σn distances computed, hence the preprocessing step is linear on σ for a fixed n and can be large. For example, for CoPhIR-10M it ranges from 49 minutes to 32 hours, for $\sigma = 64$ and 2048 respectively. For the documents database, it requires

14.45 seconds using 64 references, and up to 9 minutes for $\sigma = 2048$. A simple scheme to speed up the construction is to index the references and then solve K-nn searches over R, speeding both search and building times. In particular, we may use a NAPP index to index R. Notice that using a larger R set produces a faster index; the sketched boosting technique may allow a significant increase in the number of references. This is part of our future research.

Another solution is a simple parallelization of the preprocessing algorithms. We already implement it, achieving close to 100% efficiency per core, meaning that the preprocessing step is divided by the number of active threads (cores). However, in order to work on databases of internet scale, we require algorithms taking advantage of computing power of large distributed systems not just on single multi-core systems. Nevertheless, the distributed techniques are beyond the scope of this thesis.

The searching of data-structures for small, fast, and simple K-nr methods yield us to indexed sequences. However, there exists other options to be studied like Fulltext Self Indexes. Even when the PP-Index (K-nr prefixes) is directly implemented with these indexes, other proximity predictors should modify its algorithms to be able to work on the fulltext primitives.

On the next chapter we will present an alternative implementation of the K-nr set indexes, based on inverted indexes. The chapter will deeply study the set involved problems, producing faster variants of the searching methods, and much smaller indexes.



(a) RVEC-4-1000000. Recall (b) RVEC-4-1000000. Proximity ratio

(c) RVEC-8-1000000. Recall (d) RVEC-8-1000000. Proximity ratio

(e) RVEC-12-1000000. Recall (f) RVEC-12-1000000. Proximity ratio

(g) RVEC-16-1000000. Recall (h) RVEC-16-1000000. Proximity ratio

(i) RVEC-20-1000000. Recall (j) RVEC-20-1000000. Proximity ratio

(k) RVEC-24-1000000. Recall (l) RVEC-24-1000000. Proximity ratio

Figure 8.10: Performance of the K-nr indexes on varying dimensionality, RVEC-*-1000000. σ grows from right to left (one point per σ value on the curves)

Chapter 9

Succinct Nearest Neighbor Search

In the previous chapter we introduce NAPP, an approximate set of indexes with high result's quality, small memory footprint, and fast searching times. On this chapter one of the best NAPP indexes, the K-nr Jaccard Index, is studied and extended, obtaining faster and smaller indexes, while it preserves the quality of the answers. This new index is dubbed as the *compressed* NAPP inverted index.

In general, proximity in the NAPP framework is hinted by comparison of its shared references, e.g., the size of the intersection of two sets, hence the natural choice for an index is an inverted index. Each region in R will be represented by an integer identifier, and as consequence the representation of each object will be a list of integers.

9.1 The NAPP Inverted Index

As in the previous chapter, the size of R (the set of references) should be way smaller than the database S, yet it should reflect the distribution of objects in S. Hence we select $\sigma \ll n$ objects for R uniformly at random. Each element of S, and each element of R, will be denoted by an integer. Actual objects may reside somewhere else, for example on disk. We have $R = \{1, \cdots, \sigma\}$ and $S = \{1, \cdots, n\}$; it should be clear from context which collection an index i refers to.

For our computations we define a list for each reference r, $L[r] = \{s_1, s_2, \cdots, s_k\} \subseteq S$ such that $r \in P_{s_i}$. In other words, $L[r]$ is the list of all elements having reference r among their K nearest neighbors. Algorithm 14 gives the con-

struction.

algorithm 14: Construction of the NAPP inverted index

```
1: R is the set of references of size |R| = σ.
2: Let L[1, σ] be an array of sorted lists of integers.
3: Let S = {1, ··· , n}
4: for i from 1 to n do
5:    Compute P_i[1, K], the K nearest neighbors of i in R
6:    for j from 1 to K do
7:       L[P_i[j]] ← L[P_i[j]] ∪ {i}
8:    end for
9: end for
```

Experimentally, we have observed that most objects with a small intersection cardinality (1 or 2) appear very frequently in the candidate list even though they are not always close to the query. It is then natural to impose an additional condition about the minimum size of the intersection. This strategy is implemented using a t-threshold algorithm, a generalization of the set union/intersection problem of K sets, where the solution is a collection of objects appearing in at least t sets. Setting $t = 1$ is equivalent to the set union and $t = K$ is equivalent to the set intersection. We adapted the Barbay-Kenyon algorithm [Barbay and Kenyon, 2002] to obtain the candidate list. This is described in Algorithm 15.

To represent R we need $\sigma \log n$ bits, using pointers to S, or σ objects if they are represented explicitly. The storage requirements of the K-nr Jaccard mapping is $Kn \log \sigma$ bits, see Section 8.6. Using the inverted index the space cost increases to $Kn \log n$ bits, i.e., a total of Kn integers of $\log n$ bits, distributed among the σ sorted lists. This expands over the previous cost, yet it will simplifies involved searching algorithms and compression techniques.

There are at most $\frac{Kn}{B}$ samples overall and, as usual, the worst case in space will arise when each list contains $\frac{Kn}{B\sigma}$ of them.

9.2 The Compressed NAPP Inverted Index

The space of our index is Kn integers. For a typical value like $K = 7$, this is larger than the typical overhead introduced by some tree data structures (like the List of Clusters, Section 2.2.2). Yet it is much smaller than Pivot Indexes 2.2.1. Nevertheless, here there is room for improvement by compressing the index using indexed bitmaps, with a very small speed penalty

algorithm 15: Solve a k-nn query in the inverted index

Let t be the minimum allowed cardinality of the intersection, and Γ the number of desired candidates to be checked with the distance.
Let $L[1, \sigma]$ be an array of sorted lists of integers, i.e. the inverted index.
Compute $P_q[1, K]$
Let Q be the corresponding lists of the regions in P_q, computed as $Q[1, K] = L[P_q[1]], \ldots, L[P_q[K]]$
Let $\mathsf{POS}[1, K]$ be an array of pointers to the current position of the i-th list on Q, starting in 1.
Let CND be a priority queue to store the set of candidates
while $Q.Length \geq t$ **do**
    Ascending sort Q using $Q[i][\mathsf{POS}[i]]$ as key for $1 \leq i \leq Q.Length$; identifiers are permuted to follow the order of Q
    if $Q[1][\mathsf{POS}[1]] \neq Q[t][\mathsf{POS}[t]]$ **then**
        Advance all $\mathsf{POS}[i]$ for $1 \leq t-1$ such that $\mathsf{POS}[i]$ is the smallest item such that $Q[i][\mathsf{POS}[i]] \geq Q[t][\mathsf{POS}[t]]$
        Restart the loop
    end if
    Find the greatest $k \geq t$ such that $Q[k][\mathsf{POS}[k]] = Q[t][\mathsf{POS}[t]]$, then k is the cardinality of the intersection
    if $k = Q.Length$ **then**
        Increment all $\mathsf{POS}[i] \leftarrow \mathsf{POS}[i] + 1$ for $1 \leq i \leq Q.Length$
    else
        Advance all $\mathsf{POS}[i]$ for $1 \leq k$ such that $\mathsf{POS}[i]$ is the smallest item such that $Q[i][\mathsf{POS}[i]] \geq Q[k+1][\mathsf{POS}[k+1]]$
    end if
    Append $Q[t][\mathsf{POS}[t]]$ to CND
    Evaluate the distance between the query and Γ candidates (with the highest priority from CND)
    Return the k closest objects to the query
end while

Note 1: Increasing and advancing in POS and Q requires to checked for overflows, in such case the entry must be removed from both POS and Q. This is why we use $Q.Length$ instead K.

Note 2: *Advance* means searching for the desired key in the list, in particular we use doubling search [Knuth, 1998] since it makes the algorithm of Barbay-Kenyon instance-optimal in the comparison model [Barbay and Kenyon, 2002].

for set union and intersection computations, Appendix A.

Summarizing, we must handle σ lists. The s items of a list are distributed in the range $[1 \cdots n]$; then ideally we can represent that list using $\log \binom{n}{s}$ bits. Using the *SArray* indexed bitmap of Okanohara and Sadakane [Okanohara and Sadakane, 2007], we can represent such an inverted list using $s \log \frac{n}{s} + 2s + o(s)$ bits. As all the s items add up to Kn items overall, the worst case arises when $s = Kn/\sigma$ for each list, where the complete index takes $Kn \log \frac{\sigma}{K} + 2Kn + o(Kn)$ bits. The SArray supports constant access to every position of every list.

9.2.1 Inducing Runs in the Index

The plain representation and the *SArray* encoding are enough to host medium to large databases in main memory in a standard desktop computer. In particular, when using the *SArray* the index is compressed to its zero-order entropy and the extension to secondary memory is straightforward.

On the other hand, to handle very large databases or when using devices with scarce memory (such as a mobile phone), better compression is needed. The additional compression is obtained by renumbering objects (represented by integers) so as to obtain proximal integers in the inverted lists $L[r]$. This is done by observing that objects in any given inverted list $L[r]$ share at least the reference r, and hence cannot be much far apart from each other, as described in Observation 1. The procedure starts computing the mapped space, where each object $u \in S$ is represented by P_u, i.e., implemented as an array of integer identifiers of $\mathsf{K\text{-}nn}_{R,d}(u)$. Also, P_u is sorted according to the region identifiers, not by proximity to u. Secondly, the database is lexicographically sorted, using a linear sort for the first levels and a three-way quick sort for deeper levels, similarly to practical suffix array sorting algorithms [Puglisi et al., 2007]. Thirdly, the permutation of S induced by the sort is used to renumber the database S. Finally, the inverted index is created using the permuted mapping.

The first step creates ranges inside inverted lists of consecutive integers such that the i-th integer plus 1 is equal to the $(i+1)$-th integer. These regions are named *runs*, and are suitable to be encoded with a Run-Length scheme. For ranges not in a run we aim at having small differences between consecutive elements. Although *SArray* does not profit from runs and small differences, we can reduce space significantly by using the bitmaps DiffSet and DiffSetRL, described on Appendix A. Those bitmaps are basically sorted lists encoded as differences between contiguous items. These differences are encoded with Elias-γ or Elias-δ integer encoders. In order to

ensure the efficiency on its basic operations, we store absolute values each B differences. Details are provided in Appendix A.

Since a lexicographic sort was applied (section 9.2.1) and $R \subset S$, then we found at least σ runs in the whole inverted index.

In the worst case, each run has a length close to n/σ, induced by the first position of P_u. These runs are stored in $\sigma \log \frac{n}{\sigma}$ bits. Moreover runs induced by the first region of every P_u can be represented using $\sum_{p \in R} \log \#p$ bits, where $\#p$ is for the number of objects sharing the region p. In general, the minimum number of runs can be analized using a trie of the mapped space, as follows:

- Suppose that every P_u (sorted numerically) has a unique suffix, e.g appending the object id.
- Append every P_u to the trie.
- The number of leafs below a node defines the length of a run.

Yet, the parametrization to metric properties of a database is quite complicate and beyond the scope of this thesis.

Each sorted list L (of size n_1 with values in $[1, \ldots, n]$) are represented using DiffSet (Appendix A) using $n_1 \log \frac{n}{n_1} + o(n/\sigma)$. If the L contains *runs* (ranges of consecutive values) DiffSet-RL and DiffSet-RL2 can yield to a significant reduction of both memory requirements and operation time cost, Appendix A.

9.3 Experimental Results

As in Chapter 8, all experiments consist on querying the 30-nn on Documents, Colors-hard, CoPhIR-10M, and RVEC-*-1000000 synthetic datasets (see Section 1.3.3 for details on these databases and the associate queries). The entire databases and indexes are maintained in main memory and without exploiting any parallel capabilities of the workstation, Section 1.3.

All our experiments were performed fixing $K = 7$ (are based on the K-nr simplification of NAPP) and with several σ values. The selection of K affects the space required, the searching time, and the quality of the answer. We observed experimentally that $K = 7$ is a good tradeoff between space, time and recall. The support of this choice is based on the assumption that every object in the database holds the same probability distribution. Thus, fixing K to a constant will fix the covering radii of the references to a constant

Id	P_u	
	Orig.	Num. Sort
1	312	123
2	321	123
3	123	123
4	421	124
5	521	125
6	431	134
7	513	135
8	531	135
9	154	145
10	541	145
11	514	145
12	145	145
13	235	235
14	532	235
15	423	245
16	245	245
17	254	245
18	542	245
19	345	345
20	354	345
21	543	345

Inverted index

```
1 -> 1,2,3,4,5,6,7,8,9,10,11,12
2 -> 1,2,3,4,5,13,14,15,16,17,18
3 -> 1,2,3,6,7,8,13,14
4 -> 4,6,9,10,11,12,15,16,17,18,
     19,20,21
5 -> 7,8,9,10,11,12,13,14,15,16,
     17,18,19,20,21
```

Inverted index with differences

```
1 -> 1,1,1,1,1,1,1,1,1,1,1,1
2 -> 1,1,1,1,1,8,1,1,1,1,1
3 -> 1,1,1,3,1,1,5,1
4 -> 4,2,3,1,1,1,3,1,1,1,1,1,1
5 -> 7,1,1,1,1,1,1,1,1,1,1,1,1,1,1
```

Inverted index with differences + Run-Length

```
1 -> (1,12)
2 -> (1,5),8,(1,5)
3 -> (1,3),3,(1,2),(1,1)
4 -> 4,2,3,(1,3),3,(1,6)
5 -> 7,(1,14)
```

Figure 9.1: Example of the induction of runs for plain, differences and run-length encoding of lists. Here $\sigma = 5$, $n = 21$.

value, equivalent to a constant percentile of objects being covered. This simplification reduces the complexity of fine tuning the parameters of the index. Experimental results validating this choice are presented in [Esuli, 2009; Amato and Savino, 2008], and recently by [Tellez et al., 2011a]. The particular value of $K = 7$ is not optimal for all datasets, but for simplicity we used this fixed value, which in particular shows the robustness of the methods. Moreover, using this fixed value enhances the recall quality of some of the state of the art methods, yet it affects both time and space performances.

9.3.1 General Performance

In this section we analyze the result's quality, searching time, and the percentage of the reviewed database in the CoPhIR-10M database. Experimental results are shown for two type of queries: t-threshold queries, and 1-threshold with fixed number of verified objects.

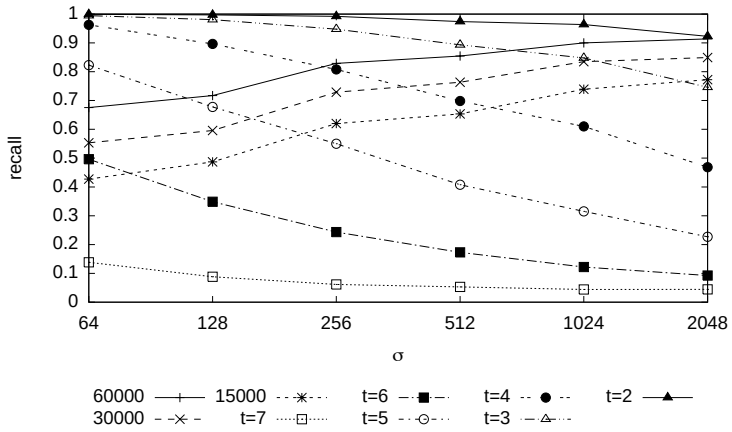
Here, our primary quality measure is the recall: the ratio between relevant results in S and the relevant objects obtained, Section 1.3. Since our queries are 30-nn, the recall is just the number of true 30-nn elements returned, divided by 30. This measure ignores how close the non-relevant objects are from the true 30-nn. In the next section we discuss this point.

Figure 9.2a shows how the recall evolves with the number of references. Methods based on t-threshold show a decreasing recall as function of t; smaller t gives better recall. Smaller σ values (number of references) give better recall, but at the cost of distance computations and time, see Figures 9.2b and 9.2c respectively. Notice that in both figures, the points in each curve are produced by indexes with different σ values. Then, when $t > 1$ the order of σ is descending as the recall increases, and for $t = 1$, σ is in ascending order. We put labels in selected curves of the figures to increase readeability.

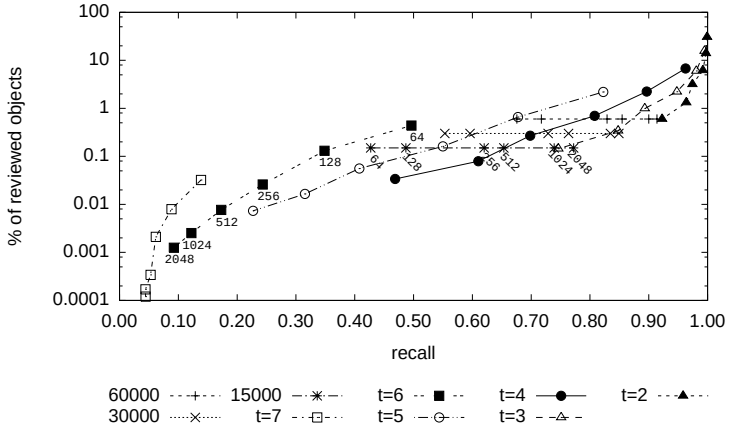
Larger σ values imply faster indexes. The speedup is produced because the Kn objects are split into more inverted lists. We note that the distribution of lengths (of inverted lists) is not Zipfian as in text inverted indexes for natural languages.

All these parameters induce tradeoffs that can be used to effectively tune real applications. For example, for $t = 2$ and $\sigma = 2048$ the index achieves 0.92 of recall, reviewing 0.6% of the database in about 0.4 seconds.

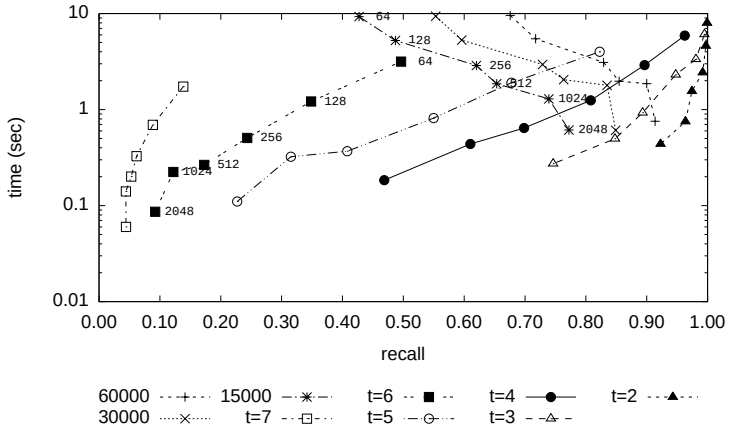
Large t values produce faster searches, since the algorithm skips parts of the input lists, due to *advance* commands in Algorithm 15. Fixing γ, the number of elements to be verified, restricts the percentage of verified ele-



(a) Recall.



(b) Percentage of database reviewed.



(c) Searching time.

Figure 9.2: CoPhIR-10M, $n = 10^7$, $K = 7$. Recall and searching time performance

σ	γ	max-ratio			
		mean	stdev	min	max
64	15000	1.06	0.04	1.00	1.25
128	15000	1.05	0.03	1.00	1.25
256	15000	1.03	0.03	1.00	1.27
512	15000	1.02	0.02	1.00	1.12
1024	15000	1.02	0.01	1.00	1.06
2048	15000	1.01	0.01	1.00	1.10
64	30000	1.04	0.03	1.00	1.19
128	30000	1.03	0.02	1.00	1.16
256	30000	1.02	0.02	1.00	1.26
512	30000	1.01	0.01	1.00	1.11
1024	30000	1.01	0.01	1.00	1.04
2048	30000	1.01	0.01	1.00	1.07
64	60000	1.02	0.02	1.00	1.12
128	60000	1.02	0.02	1.00	1.09
256	60000	1.01	0.02	1.00	1.25
512	60000	1.01	0.01	1.00	1.07
1024	60000	1.01	0.01	1.00	1.04
2048	60000	1.00	0.01	1.00	1.06

Table 9.1: Statistics of the covering radius (30-th nearest neighbor) of CoPhIR-10M

ments of the database and hence bounds the total time. See lines "15000", "30000" and "60000" of Figure 9.2. In this case $t = 1$ and the t-threshold algorithm is equivalent to set *union* (being linear in the number of items in the input lists). Notice that under this configuration, the performance is driven by CND, i.e. the priority queue of Algorithm 15. Based on Figure 9.2c, this strategy (lines named "15000", "30000" and "60000") is useful to control the search time, yet it needs to compute the entire set *union*.

Naturally, a hybrid configuration achieves better control of the performance and quality, i.e. the combination of t-threshold and fixed γ. For example, for $\sigma \geq 1024$ pure t-threshold configurations yields to better times than just fixing the cardinality, see Figure 9.2c. The inverse is true for $\sigma < 1024$.

9.3.2 Proximity Ratio as a Measure of Retrieval Quality

In multimedia information retrieval applications, especially when some relevance feedback is expected from the user, we want to measure how close

σ	γ	max-ratio			
		mean	stddev	min	max
64	100	1.14	0.17	1.00	2.01
128	100	1.11	0.14	1.00	1.87
256	100	1.10	0.17	1.00	2.27
512	100	1.05	0.07	1.00	1.58
1024	100	1.03	0.06	1.00	1.51
2048	100	1.02	0.02	1.00	1.10
64	500	1.08	0.14	1.00	1.86
128	500	1.05	0.10	1.00	1.80
256	500	1.03	0.09	1.00	1.77
512	500	1.01	0.02	1.00	1.12
1024	500	1.01	0.03	1.00	1.22
2048	500	1.00	0.01	1.00	1.07
64	1000	1.05	0.08	1.00	1.62
128	1000	1.02	0.03	1.00	1.14
256	1000	1.01	0.03	1.00	1.14
512	1000	1.01	0.02	1.00	1.12
1024	1000	1.00	0.01	1.00	1.06
2048	1000	1.00	0.01	1.00	1.07

Table 9.2: Radius statistics for Documents dataset.

the reported *non-relevant* objects (the false positives) are from the relevant ones. To this end we show some statistics of the ratio between the covering radius of the 30-th nearest neighbor and the distance given by NAPP in Table 9.1. Note that large σ values produce results that are very close to the real answers, supporting Observation 1, which bounds the distance to the query, not the recall. Actual distances for the 30-th nearest neighbor in our query set have the following statistics: mean=3958.16, standard deviation=930.24, minimum=1418, and maximum=6531. Even on the largest ratios, the retrieved items are in a quite small percentile of the dataset, as shown in Figure 1.1.

The same statistics are given for the database of Documents and Colorshard, respectively in Tables 9.2 and 9.3. Notice the indexes have worse performance on these databases, probably because of the high intrinsic dimensionality of these datasets, especially for Documents, as shown in the histograms of Figure 1.1 and Table 1.1

Note that, in all the experiments, the proximity ratios are very close to 1. Please notice that quality measured as proximity ratio is of use on the majority of real world applications.

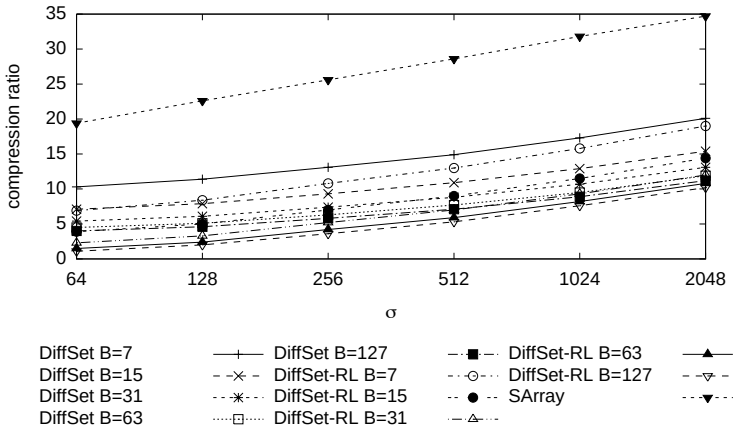
σ	γ	max-ratio			
		mean	stddev	min	max
1000	64	1.05	0.09	1.00	1.70
1000	128	1.03	0.06	1.00	1.28
1000	256	1.03	0.06	1.00	1.25
1000	512	1.02	0.05	1.00	1.24
1000	1024	1.01	0.04	1.00	1.25
1000	2048	1.00	0.02	1.00	1.15
2000	64	1.04	0.09	1.00	1.69
2000	128	1.03	0.06	1.00	1.26
2000	256	1.02	0.05	1.00	1.25
2000	512	1.02	0.04	1.00	1.24
2000	1024	1.01	0.03	1.00	1.25
2000	2048	1.00	0.02	1.00	1.14
3000	64	1.04	0.08	1.00	1.63
3000	128	1.02	0.05	1.00	1.25
3000	256	1.02	0.05	1.00	1.25
3000	512	1.01	0.04	1.00	1.24
3000	1024	1.01	0.03	1.00	1.25
3000	2048	1.00	0.02	1.00	1.14

Table 9.3: Radius statistics for the Colors-hard.

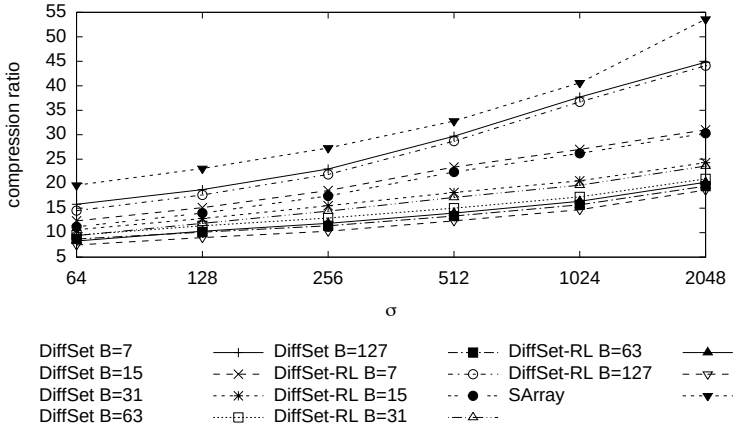
9.3.3 Experimental Results on the Compressed NAPP Inverted Index

Our plain inverted index uses a fixed number of bits. For example, the index for CoPhIR-10M uses 267 MiB, i.e., each object is represented with 224 bits, using integers of 32 bits. The compressed representation uses from 10 to 80 bits per object for CoPhIR-10M, and 15 to 80 bits in Documents. For Colors-hard, we found a behavior similar to CoPhIR-10M, yet requiring some additional bits per object.

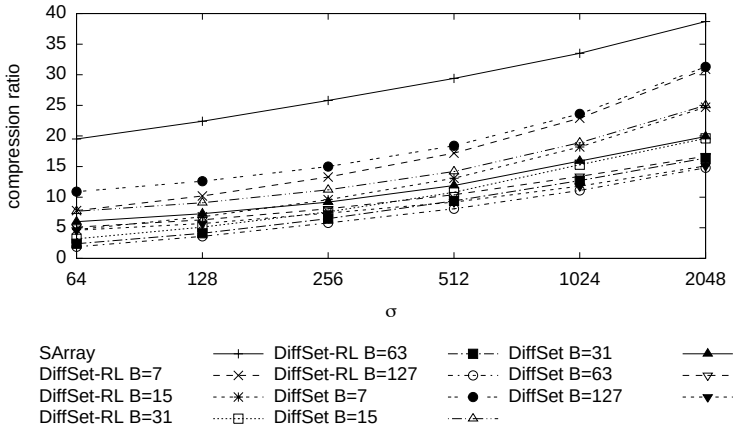
Our experiments confirm that the number of runs is large, and the index is better compressed. The effect is that the smallest index is the run-length based one and the largest compressed index is the *SArray*, as shown in Figure 9.3. Note that even the space gain of *SArray* is considerable. σ is also a crucial parameter for compression. Small σ values produce a small index, yet it needs to review larger portions of the database.



(a) CoPhIR-10M, 10^7 objects, the plain index uses 267.03 MBytes



(b) Documents, 25057 objects, the plain index uses 0.67 Mbytes



(c) Colors-hard, 112682 objects, the plain index uses 3.1 Mbytes

Figure 9.3: Compression ratio as a percentage of the plain inverted index for our experimental data sets.

type of encoding	B	searching time (sec) CoPhIR-10M	Documents	Colors-hard
DiffSet	7	2.57	0.020	0.0057
DiffSet	15	3.34	0.020	0.0058
DiffSet	31	4.81	0.022	0.0061
DiffSet	63	7.69	0.025	0.0066
DiffSet	127	13.50	0.028	0.0075
DiffSet-RL	7	2.57	0.019	0.0054
DiffSet-RL	15	2.73	0.019	0.0055
DiffSet-RL	31	2.75	0.019	0.0056
DiffSet-RL	63	2.71	0.019	0.0054
DiffSet-RL	127	2.64	0.020	0.0055
SArray	-	0.34	0.031	0.0056
plain (*with runs*)	-	0.17	0.024	0.011
plain (*original*)	-	0.42	0.029	0.014

Table 9.4: The average time necessary to search a query in the compressed NAPP inverted index and the plain version. Indexes were configured using $\sigma = 2048$, $(t = 2)$-threshold search. Indexes for CoPhIR-10M $\gamma = 15000$, and $\gamma = 1000$ for Documents and Colors-hard.

Time Performance of the Compressed Index

In the experiment, all compressed indexes were produced with induced runs. For the *plain* index we show the two encodings, with and without induced runs because it affects the retrieval speed. For example, for the CoPhIR-10M index the plain index working with the induced runs is about 2.5 times faster than the original one. This is not surprising since the t-threshold algorithm is instance optimal. For DiffSet and DiffSet-RL bitmaps, the parameter B (section 9.3.3) manages the tradeoff between time and compression. DiffSet and DiffSet-RL are still interesting methods since they achieve low compression ratios, as shown in Figure 9.3. Moreover, as depicted in Table 9.4 for the CoPhIR-10M dataset, the run-length based indexes are just four times slower than the NAPP inverted index (without runs).

This tradeoff is significant for the CoPhIR-10M database (Table 9.4), where the search time increases several times, as compared with the plain representation. The *SArray* coding is quite fast (faster than *plain original*) and still compress significantly. This can be explained because the *SArray* gives constant time access to the i-th element [Okanohara and Sadakane, 2007]. Contrasting with the CoPhIR-10M results, compressed indexes for the Documents database are as fast as the plain representation, and even

faster for some configurations (i.e., for *SArray*), see Table 9.4. Even more, for the smaller Colors-hard dataset, all compressed indexes are twice as fast compared to the original index, and it even surpass the plain index with runs. Remarkably the run-length representation use close to constant time in B.

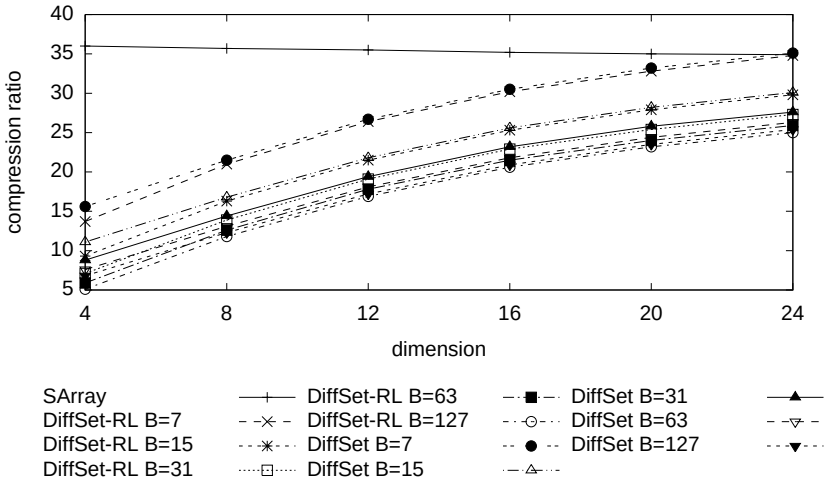
Note that small compressed inverted indexes can fit in the CPU caches. This applies to inverted lists involved in the solution of a particular query. Also notice that the distribution of runs produces easier instances for the t-threshold algorithm, taking advantage of the Barbay-Kenyon t-threshold algorithm. Among the smaller databases, the most important difference is the intrinsic dimensionality, see Figure 1.1, that produces fewer runs and less compression as depicted in Figure 9.3.

9.3.4 The Dimensionality Effect

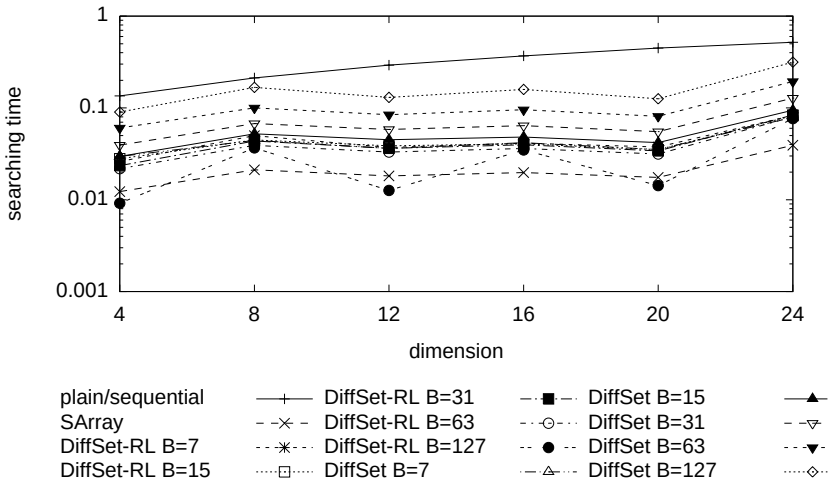
For this experiment, we produce databases in the real unitary hypercube of dimensions 4, 8, 12, 16, 20, and 24; and all coordinates were selected using a uniform distribution. Each dataset contains one million vectors, i.e. $n = 10^6$. All results queries are allowed to review 1000 items, and all indexes were built using 2048 references and $K = 7$. Under this configuration, the number of reviewed objects is fixed to 0.3% of the database, in terms of distance computations. We performed queries for the nearest neighbor search of random vectors not indexed.

We can see the effects of the dimensionality in the compression ratio in the Figure 9.4a. As expected the compression capabilities are significantly diminished as the dimension increases. On searching behavior, we must remark that only 0.3% of the possible number of distances are evaluated, moreover we are achieving close to perfect recall for all configurations. The search time is depicted in Figure 9.4b, we can observe the speedup (up to two orders of magnitude) against sequential search, that is practically necessary for large dimensional datasets (more than 20 dimensions) for the majority of exact metric indexes.

The behavior of the proximity ratio of the nearest neighbor is shown in Table 9.5. The mean is quite small, ranging from 1.0 to 1.02, and a very small standard deviation. The maximum ratio is small too, but exposing a solid increment in the dimension. Random uniform data is free of clusters. Even on this setup, our index shows a good tradeoff among memory, space, speed, recall, and proximity ratio. To put this in perspective, notice that even if the Documents have intrinsic dimensionality higher than RVEC-24-* (Table 1.1), it performs better on compression since its objects tend to form



(a) The effect of the dimension on the compression ratio. $n = 10^6$



(b) Searching time on increasing dimensionality. $n = 10^6$

Figure 9.4: Behavior of the NAPP compressed index as the dimension increases

dim	mean	stddev	min	max
4	1.00	0.00	1.00	1.00
8	1.00	0.00	1.00	1.00
12	1.00	0.02	1.00	1.21
16	1.00	0.02	1.00	1.19
20	1.01	0.04	1.00	1.24
24	1.02	0.04	1.00	1.26

Table 9.5: Proximity ratio as affected by the dimensionality. The NAPP uses a threshold of 2.

clusters.

9.4 Summary

In this chapter we introduced a new representation of the K-nr Jaccard method, called NAPP inverted index. It consist on an inverted index created over the set of regions (references) of each object of S. Also, we compressed the NAPP inverted index yielding to a very efficient metric index with small memory footprint. The plain index uses a few integers per object and the compressed versions use a few bits per object, with a small penalty in search speed for large databases, and a small speed up for small ones.

Our index is capable of achieving high quality results in sub-second queries, even for large metric databases. This is a characteristic inherited by K-nr methods (Chapter 8, yet here is fully exploited using even more heuristics like t-threshold based pruning of the list of candidates.

We provide an extensive set of experimental evidence on both real-world and synthetic datasets. Those results support our claims on quality of results, memory requirements, searching time, robustness to large datasets, and increasing dimensions. Nevertheless, a lot of work is necessary to provide a better understanding of the K-nr technique, since proper theoretical bounds (exact and probabilistic ones) in terms of recall and proximity ratio are currently unknown. Also, K-nr indexes (including the ones studied in this chapter) does not support dynamic operations. A set of effective algorithms and structures are required to efficiently support insertions and deletions on both S and R.

Chapter 10

Conclusions

In this work we study the proximity searching problem from a practical perspective, the idea was to produce algorithms and data-structures working with large databases (millions of objects) on high intrinsic dimensions, and still working with current power of computing.

We encompass both exact and approximate proximity searching techniques. From our practical perspective, we concern about several practical properties of the indexes, namely, preprocessing time, searching time, and storage requirements. Also, on approximate techniques, we care about result's quality. This measure is presented on two fashions: recall and proximity ratio.

We obtained solid improvements to the state of the art, and indeed our indexes can be used in modest hardware setups. So, a set of possibilities arrive since devices with relatively small computing power are becoming more popular every day, i.e. smart phones, tablets, net-books, game consoles, etc. Our techniques bring efficient support for proximity searching in these devices. In general, the majority of our indexes were stored in close to the optimal space of its representation. The compression allows one to efficiently use higher hierarchies of memory (RAM, L2 and L1 cache).[1]

[1]Source code, databases, testbeds, and a couple of demos are freely available from `http://www.natix.org`, and `http://github.com/sadit/natix/`.

10.1 Achievements

10.1.1 Exact Techniques

We introduce Rev-LC (Chapter 3) a fast metric index based on the List of Clusters (LC). It has an efficient preprocessing time while maintain the storage requirements of the LC. Compared with LC, the searching time is a bit larger, yet similar on high intrinsic dimensions. It is specially good for large datasets with high intrinsic dimensions.

We parallelize the LC index in Chapter 4. The objective is to take advantage of the current multi-core hardware. We obtain high core efficiencies and speed ups on preprocessing steps with hard setups, i.e., $n/m = O(1)$, large n, and time-expensive distance functions. On the other side, range searching algorithm is quite good on the same setups, but nn searching algorithm performs worse since dynamic information is shared among threads.

Chapter 5 presents the Polyphasic Metric Index (PMI), our main contribution to exact metric indexes. The idea is to use several backend indexes in order to speed up the searching process. In particular, we use LC as backend index.

Surprisingly, given a λ small value depending on the intrinsic dimension of the dataset, the PMI using λ LC indexes has sub-quadratic preprocessing cost on databases requiring quadratic time by LC. This enhancement is because underlying LC indexes are non-optimal by itself and they use large n/m values. With these setups, the PMI performs less distance computations than the optimal LC, even for high intrinsic dimensions. Contrary to LC, and the majority of the metric indexes, the PMI can gradually adapt to the intrinsic dimensionality of a particular dataset using a fixed preprocessing setup (i.e. fixed large n/m value). Also, it can be adapted to the complexity of a particular query. Both optimizations are performed simply varying λ at query time. We present range and k-nn searching algorithms, the last one is of special interest since it becomes optimal on the number of distances performed. Nevertheless, the PMI's memory cost is multiplied with respect to LC in a λ factor.

We reduce the impact of λ creating compressed representation of a metric index in Chapter 6. If the PMI uses this compressed index as backend, the memory cost reduces from 32λ bits to $\log(m+1)\lambda$ bits per object. So, our PMI becomes $\frac{32}{\lambda \log(m+1)}$ larger/smaller than LC as a setup of m and λ. At the best of our knowledge, this is the first attempt to produce an exact compressed metric index. An additional strong conclusion on the compression of metric spaces is that the intrinsic dimensionality drives the memory

cost per item, that is, datasets with low intrinsic dimensions require small amounts of memory; contrary, high intrinsic dimensional datasets require larger amounts of memory. At the best of our knowledge, we provide the very first prove of this correlation between storage requirements and the intrinsic dimensionality.

10.1.2 Approximate Techniques

Chapter 7 introduces the Locality Sensitive Classification (LSC), a new representation of the Locality Sensitive Hashing. This new representation can be easily represented in close to optimal space without significant speed penalties. The objective applications are those requiring high quality in the result set using multiple LSH indexes. Another case of use is to bring proximity searching techniques into small computing devices like smart phones and tables, and in general devices with relatively small storage capacities.

Chapter 8 introduces a new family of approximate metric indexes. The basis technique is called Neighborhood Approximation (NAPP), since it actually map a metric space into a similarity space using a set of references, and in some way, these references are defining neighborhoods. The mapping has a strict structure allowing faster searches on it. Also, the mapping preserves the proximity between objects, such that comparing items in this transformation gives a hint of its closeness in the original space. A simplification of the technique is presented, called the K nearest references (K-nr). Based on K-nr we describe other state of the art indexes, and create several new metric indexes. In this sense, we introduce a representation based on sequences of symbols where the majority of the K-nr searching methods can be implemented, so a single instance gives support to several searching methods.

Finally, in Chapter 9 we implement one of our best and simple K-nr indexes, with an inverted index. This change allow us to introduce some sophisticated algorithmic techniques allowing more searching variants like those based on the t-threshold set operation. Also, we compress the inverted index producing a fast and small approximate metric index with high quality in the results. Furthermore, we introduced a novel technique able to induce runs in the inverted index, usable in at least two scenarios: for speeding up a plain index, and for inducing compression in compressed indexes. The *SArray* index produces a fast compressed version and can be used with or without induced runs. Differential encoding plus run-length compression achieves high compression rates and at the same time very fast indexes.

We show the behavior of our techniques in three real world datasets, and

a set of six different dimensions of synthetic databases (with three different sizes each one). Experiments on each dataset increase our insight of NAPP indexes for different situations. Real world databases show the performance (time, compression, and retrieval quality) of what we can found for real applications. On the other side, synthetic databases isolates (i) the effect of the dimensionality in our index, that is one of the greatest challenges of the metric indexes; and (ii) the dependency on n, a problem commonly ignored by most metric indexes. In all cases, both our plain and compressed NAPP index, arises as an exceptional good tradeoff between memory, time, and result's quality, making them excellent options for several real world applications.

10.2 Future Work

In general, our indexes work on static collections of objects, and they do not support insertions or deletions of items. A future research should extended our indexes in order to support dynamic operations. Also, our techniques use only main memory to store both the index and the database. Unfortunately, even with compressed representations of the indexes the main memory is relatively small on the current generation of computers. There exists two main approaches solving this issue: (i) disk based compressed indexes, and (ii) distributed indexes across a network of computers. Both solutions are beyond the study of this thesis, but these should lead the next generation of practical proximity indexes.

In the exact proximity searching methods, our study consider mainly compact partition indexes because their innate low memory usage. However, based on our techniques to compress metric indexes, pivot indexes can be greatly improved (reducing its memory requirements). The basic idea is to create really large pivot tables in a very small space. Also, new algorithms for union-intersection (for the PMI) should arise on these new indexes.

On the other side, K-nr indexes should support dynamic operations not just inserting and deleting objects from the database, also, they must support dynamic operations on the set of references. This is important for a high transactional environment. Also, the preprocessing time of K-nr indexes should be improved, a promising scheme is to index the set of references R.

The real time performance of our Compressed NAPP Inverted Index is tightly linked to the underlying t-threshold algorithm, which is primarily designed for uncompressed inverted indexes. So, the creation of faster ad-hoc algorithms for the t-threshold problem for the compressed inverted lists,

and in the optimization and scalability of the technique using parallel and distributed techniques remains as an open problem.

Maybe the weakest part of our NAPP indexes is the lack of tight theoretical guarantees on the result's quality. This is a hard challenge, and will be part of our future research.

Appendix A

Sequences on (Very) Large Alphabets

The majority of the indexes for sequences will fail as σ grows. Unfortunately, our metric indexes (and new representations) are sequences of size n with a very large alphabet Σ ($\sigma = |\Sigma|$). Typically, $\sigma = O(n^\alpha)$ for some $\alpha < 1$ or even $n/\sigma = O(1)$, as is the case of the list of clusters (Chapter 6), or the Locality Sensitive Classification (Chapter 7). Even more, in our indexes we expect a high zero order entropy, i.e. $H_0 \simeq \log \sigma$ (Section 6), that is, there is no a useful deviation on the symbols' frequency that can be exploited to reduce the memory usage. Indexes achieving entropy of higher order (H_h) are promising alternatives to K-nr proximity indexes. Nevertheless other indexes like the List of Clusters cannot take advantage of this property, since we cannot ensure it. Thus, H_h techniques are beyond the scope of this work. This kind of indexes (for text) are nicely surveyed by Navarro and Mäkinen [Navarro and Mäkinen, 2007].

Due to the particularities of our proximity indexes, we pay special attention to indexes of sequences achieving local H_0, which is smaller or equal than the global H_0. On other application's domain, local H_0 is ensured by many other kind of sequences, like those arising on the differences of the ψ function on the Compressed Suffix Array [Navarro and Mäkinen, 2007], the Borrows-Wheeler transform that is the core of the FM-Index [Navarro and Mäkinen, 2007], and our K-nr sequences after the permutation by proximity (Section 9).

Since our metric indexes are highly dependent on indexed sequences, we are committed to improve indexes for large alphabets, while take advantage of local H_0 whenever is possible, since global statistics are not of use. As

shown in Chapters 6, 7, and 9 our algorithms heavily use the Select operation. In particular, we are concerned about a set of $\mathsf{Select}_c(T, r)$ calls with consecutive r.

Section A.1 explores a three new indexed bitmaps, achieving local entropy. Section A.3 studies IoS based on a single permutation. Finally, Section A.4 provides an extensive experimental comparison between our sequence indexes and some state-of-the-art indexes.

A.1 Introduction

There exists several Indexes of Sequences (IoS), since each one puts a different tradeoff between the necessary memory and the complexity of its operations.

The basic problem is as follows. Let $T = s_1 s_2 \cdots s_n$ be a sequence of symbols on the alphabet Σ of size σ, i.e. $s_i \in \Sigma$. Without loose of generality, let Σ be a set of integers, $\Sigma = \{1, 2, \cdots, \sigma\}$. The i-th symbol in T is denoted as T_i.

Let n_c be the number of symbols c in T, then we require at least

$$\log \binom{n}{n_1, n_2, \cdots, n_\sigma} = \log \frac{n!}{n_1! n_2! \cdots n_\sigma!} \text{ bits} \tag{A.1}$$

to represent any instance of T with those statistics. From information theory we can obtain the following formulation, using a fixed code word for each symbol, we require at least $nH_0(T) \leq n \log \sigma$ bits. Where $H_0(T)$ is the order zero empirical entropy of T, i.e.

$$nH_0(T) = n \sum_{c \in \Sigma} p_c \log \frac{1}{p_c} = \sum_{c \in \Sigma} n_c \log \frac{n_c}{n} \text{ bits} \tag{A.2}$$

An IoS provides three basic operations:

— $\mathsf{Rank}_c(T, pos)$ counts how many symbols c occurs in T until position pos, $c \in \Sigma$.

— $\mathsf{Select}_c(T, r)$ returns the smaller position pos such that $\mathsf{Rank}_c(T, pos) = r$.

— $\mathsf{Access}(T, pos)$ retrieves the symbol stored at the position pos in T, T_{pos}.

Notice that an IoS replaces T, since we can reconstruct it using $\mathsf{Access}(T, pos)$, but our notation requires to put T in the arguments even when it is not actually stored.

Most IoS structures are in fact constructed using a cleaver reduction of the problem to the binary case. A binary sequence, $c \in \Sigma = \{0, 1\}$, is dubbed as *(indexed) bitmap*. So, we will present a review of binary sequences, and finally a short review of IoS over larger alphabets.

A.1.1 Indexing Bitmaps

The operations supported over a binary sequence B are the same that those found in larger alphabets, but in this case several interesting relations arise. We only need to implement either Rank_0 or Rank_1 since $pos = \mathsf{Rank}_0(B, pos) + \mathsf{Rank}_1(B, pos)$. This duality is not found for Select_0 and Select_1, but another relation raises since Select_c is computed binary searching over Rank_c, using $\log n + 1$ ranks. In fact, the same technique can be used to solve Rank_c binary searching over Select_c. Better times that logarithmic on the implemented operation can be achieved with additional structures, yet the binary searching is commonly enough for most practical applications. If the indexing structure does not support $\mathsf{Access}(B, pos)$ by itself, it is solved using two Rank operations, i.e. $\mathsf{Rank}_1(B, pos)$ and $\mathsf{Rank}_1(B, pos-1)$ operations, or checking the relation $\mathsf{Select}_b(B, \mathsf{Rank}_b(B, pos)) = pos \iff \mathsf{Access}(B, pos) = b$.

It is possible to solve Rank_c and Select_c (and consequently Access too), in constant time using $n + o(n)$ bits [Jacobson, 1989; Clark, 1996] using n bits. This is useful when the entropy of the bitmap is maximal, i.e. $p_0 = p_1 = 0.5$. In order to solve Rank_1 we split the entire bitmap B in small blocks of $t = \frac{1}{2} \log n$ bits, then we obtain n/t small blocks. The rank inside each block is computed using a precomputed table for all the 2^t possible blocks of size t, and store its t rank responses (one for each bit position), each response requires $\log \log \sqrt{n}$ bits, then we require a table of $\frac{\sqrt{n} \log n}{2} \log \log \sqrt{n}$ bits. So, miniblocks responds relative ranks, and we require absolute samplings, using $\log n$ bits for each one, at the end we require $\frac{n}{t} \log n = 2n$ bits. This is very costly. A solution consist on place absolute samples each $\log n$ mini blocks, this yields to $2n/\log^2 n$ absolute samplings of $\log n$ bits each one, requiring $2n/\log n$ bits. To be able to respond in $O(1)$, we require $\log n$ relative samplings between absolute samplings, being necessary $\log n \log \frac{\log^2 n}{2}$ bits. All those structures (i.e. mini blocks, blocks, and large blocks) sum up to $o(n)$ bits.

Both Select_c can be implemented in $O(1)$ time using $o(n)$ bits, but it requires a complex structure with a large extra space and time, too much for practice. Thus, we considere that it is beyond the scope of this work. The interested reader is referred to [Jacobson, 1989; Clark, 1996]. As previously

commented, in practice it is quite common to simply binary searching on Rank_c. A more simpler and efficient indexed bitmaps for Select_c will be introduced below, but with payment on Rank_c time.

Gonzalez et al. [González et al., 2005] introduce a practical bitmap solving Rank_c and Select_c in $O(\log n)$ time. The main idea is to store the uncompressed bitmap along a table $\mathsf{AbsRank}$ of absolute Rank_1 values computed at fixed positions. Queries are solved first over the $\mathsf{AbsRank}$ and then sequentially inside gaps between absolute samples (processing $\frac{\log n}{2}$ bits at a time). It requires $n + o(n)$ bits. It is really fast on practice, yet it is uncompressed.

In our metric indexes we requiere both uncompressed and compressed bitmaps. A compressed bitmap is represented in space close to $\log \binom{n}{n_1}$ or $nH_0(B)$.

Raman et al. [Raman et al., 2002] showed how to index a bitmap B in $nH_0(B) + o(n) + O(\log \log n)$ bits. The procedure consist on split B into blocks of $t = \frac{1}{2} \log n$ bits. Each block b_i is represented using a tuple (c_i, o_i), then it can be encoded into the *local* zero order entropy. Here local means that the involved probabilities are taken from a smaller region that the whole bitmap, such that it can take advantage of contiguous 0's/1's. The term c_i is the *class* of b_i, that corresponds to the number of enabled bits in b_i. The offset o_i is the position of b_i in a numerically sorted list of all blocks of the same class. Both Rank_c and Select_c can be solved in $O(1)$ time using a small hierarchical structure of directories [Raman et al., 2002].

In practice, the tuple representation of [Raman et al., 2002] can be implemented in a simpler way, as shown by Claude and Navarro [Claude and Navarro, 2008], yet the small term cannot be neglected. The authors fixes the size of the block to $t = 15$, then the bitmap is represented using classes and offsets of the form $\mathsf{Classes} = c_1 c_2 \cdots c_{\lceil n/15 \rceil}$ and $\mathsf{Offsets} = o_1 o_2 \cdots c_{\lceil n/15 \rceil}$, since there are $\log (15 + 1)$ classes c_i requires 4 bits fixing the required space for a fixed n, independently of n or $H_0(B)$. Each entry in the table $\mathsf{Offsets}$ requires a variable length of bits, i.e. $\log c_i$, it is necessary to add synchronization points in $\mathsf{Offsets}.\mathsf{Rank}_c$ and Select_c are solved similarly to Gonzalez et al. [González et al., 2005], saving an entry in $\mathsf{AbsRank}$ at regular positions of B. This information is used to partially solve the problem and finally use sequential scanning blocks between absolute samples. The size of $\mathsf{Offsets}$ is at most $nH_0(B)$, the sampling gaps (in both $\mathsf{Offsets}$ and $\mathsf{AbsRank}$) should maintain the small term as $o(n)$. The index uses precomputed tables to obtain o_i and c_i using b_i, i.e. o_i requires a table of 2^{15} items storing all possible blocks, organized by class and ordered numerically inside each class using 15×2^{15} bits. Also, c_i is computed using a table of all possible values using

2^{17} bits, as shown in [Claude and Navarro, 2008]. The inverse process, i.e. reconstruct b_i using (c_i, o_i), is solved with the same tables.

Okanohara and Sadakane [Okanohara and Sadakane, 2007] introduce *Esp*, *RecRank*, *VCode*, *DArray* and *SArray* bitmaps solving by separate the cases when $n_0 \approx n_1$ and when $n_1 \ll n$. Bitmaps following the former constraint are called *dense*, and *sparse* for the later one. Specially, we focus on SArray (we will interchange its name with OS07 in advance) where $n_1 \log \frac{n}{n_1} + 2n_1 + o(n_1)$ bits are enough to represent a sparse bitmap. The OS07 solves Select_1 in constant time, Rank_c in $O(\log n_1)$ in worst case, and $O(1)$ average time for uniformly distributed 1's. The idea behind the $OS07$ is to store all responses of Select_1, using the division of B into two bitmaps, H and L. L stores the $\log \frac{n}{n_1}$ least significant bits (LSD) concatenated for all $\mathsf{Select}_1(B, r)$ for r from 1 to n_1. The H bitmap cleverly stores the resting $\log n_1$ most significant bits (MSD), i.e. $\log n - \log \frac{n}{n_1} = \log n_1$ bits. Then, we divide the space on at most $2^{\log n_1} = n_1$ different ranges, or divisions of the space. For each range of the space a single 0 is stored in H, followed by many ones as responses of Select_1 are inside that range. Therefore, H has at most n_1 0's and exactly n_1 ones, so, H is classified as *dense* and it is represented using the $DArray$ which uses $n + o(n)$ solving $\mathsf{Select}_1(H, \cdot)$ and $\mathsf{Select}_0(H, \cdot)$ in constant time. $\mathsf{Select}_1(B, r)$ is solved reconstructing the response, i.e. in order to solve $\mathsf{Select}_1(B, r)$ we need to perform $\frac{n}{n_1}\,\mathsf{Rank}_0(H, \mathsf{Select}_1(H, r))$ obtaining the MSD. It requires $\log n_1$ bits. Then, the LSD bits are read from the r-th entry in L (notice that each entry in L has $\log \frac{n}{n_1}$ bits). $\mathsf{Rank}_1(B, pos)$ is solved using Select_0 on H of the MSD of the desired position, then binary search is required between the entries in L corresponding to the range. Bitmaps with uniformly spaced 1's hold a constant number of 1's in each range in average, hence the average cost of $\mathsf{Rank}_1(B, pos)$ is $O(1)$.

Table A.1 wraps up the presented tradeoffs. Summarizing, both RRR02 and CN08 are quite good if the bitmap contains long substrings of a single symbol, GGMN05 is really fast in practice, but the bitmap is uncompressed. On the other side, RRR02 has a small term depending on the length of the bitmap. Consequently, CN08 has the same dependencies, furthermore the small term does not vanishes as n grows. Finally, OS07 solves Select_1 in constant time, with small terms dependent on the number of 1s; on the other hand, Rank_c and Access have $O(1)$ average time if the 1's are uniformly distributed along the bitmap.

Solution	Time		Memory
	Rank	Select	
RRR02 [Raman et al., 2002]	$O(1)$	$O(1)$	$\ell H_0(B) + o(n) + O(\log\log n)$
CN08 [Claude and Navarro, 2008]	$O(\log n)$	$O(\log n)$	$n(H_0(B) + 4/15) + o(n)$
OS07 [Okanohara and Sadakane, 2007]	$O(\log \frac{n}{n_1})$	$O(1)$	$n_1 \log \frac{n}{n_1} + O(n_1) + o(n_1)$
OS07†	$O(1)$	$O(1)$	idem.
GGMN05 [González et al., 2005]	$O(\log n)$	$O(\log n)$	$n + o(n)$

Table A.1: Indexes for Rank, Select and Access for binary sequences. Where n is the length of the bitmap, n_1 is the number of 1's in the sequence, and † means for average in uniformly distributed 1's along the bitmap.

A.1.2 Indexing Sequences with Larger Alphabets

As previously commented, most indexes for alphabets larger than 2 are solved reducing the problem to binary sequences. A simple index for sequences is created using σ (indexed) bitmats. The idea is to simulate a $n \times \sigma$ matrix, with a single one per column. The (i, c)-cell is 1 if the symbol at the i-th position is the c symbol, and 0 otherwise. This kind of IoS will be called a *unraveled* sequence.

For example, considere an unraveled sequence with a RRR02 bitmap per symbol, it can be implemented using $nH_0(T) + \sigma(o(n) + O(\log\log n))$ bits, solving Rank_c and Select_c in constant time, but Access in $O(\sigma)$ time [Navarro and Mäkinen, 2007]. However, the small terms are quite large, and in practice, $o(n)$ do not decrease, as exposed by CN08. On these terms, it is possible to use OS07 to unravel the sequence, since the small terms are depending on n_1, such that it requires $nH_0(T) + O(n)$ bits in total. Solving operations in the time listed in Table A.1. Unfortunately, a hidden space overhead appears on practice, i.e., we require additional $O(\sigma \log n)$ bits to store the headers of all bitmaps (one per row).

An elegant solution, with a small space overhead, and $O(\log \sigma)$ time for all operations is the Wavelet Tree (WT) [Grossi et al., 2003; Navarro and Mäkinen, 2007]. The WT is a balanced binary tree. Each inner node w stores a bitmap B_w of size n_w. At the root, the i-th bit of B_w is 0 if T_i is on the first half of Σ, and 1 if it is on the second half. Then, the left child is a WT node storing all symbols marked with zero, and the right one storing all symbols marked with one. This process is recursively applied adjusting Σ to each node, the division is stopped when a single symbol is found, creating a leaf node. B_w is indexed to support Rank_c, Select_c, and Access.

$\mathsf{Rank}_c(T, pos)$ is decomposed into bitmap operations, in the first step we retrieve the bit in the requested position, $a = \mathsf{Access}(B_{root}, pos)$, then pos

is updated to $\mathsf{Rank}_a(B_{root}, pos)$. If $a = 0$ then we recurse over the left child, and to the right child if $a = 1$. The procedure finalizes when it arrives to a leaf node, reporting pos as the desired rank. $\mathsf{Access}(T, pos)$ is similarly solved to Rank but returning the number of reached leaf instead.

The $\mathsf{Select}_c(T, r)$ starts performing the inverse operations, it goes to the parent of the c-th leaf, then we update r to $\mathsf{Select}_a(B_w, r)$ where w is the parent of the r-th leaf, and a is 0 or 1 if b-th leaf is left or right child, respectively. This process is recursively applied until we reach the root, the procedure returns the r value at the root node.

Then, WT solves Rank_c, Select_c, and Access in $O(\log \sigma)$. Since it is binary and balanced by definition and there exists only σ leafs. These costs suppose that the bitmap operations are $O(1)$.

Since the WT is a balanced tree of $\log \sigma$ levels, (at each level there exists n bits), thus if no compression is made, the bitmaps are stored in $n \log \sigma + o(n \log \sigma) + O(\sigma \log n)$ bits. The last term is for pointers and satellite data in nodes. As expected, any kind of indexed bitmaps can be used, then RRR02 achieves local $nH_0(T)$ as main term. Claude and Navarro [Claude and Navarro, 2008] shown how to implement the WT without pointers removing the $O(\sigma \log n)$ term, the main idea is to exploit the fixed shape of the WT, i.e., paths from the root leafs are isomorphic to binary code.

The WT is a general concept, and the shape is not necessary to be balanced, i.e. it supports any prefix code instead binary code. For example, if a Huffman shape is used then the WT achieves $n(1+H_0(T))+O(\sigma \log n)+ o(n(1 + H_0(T)))$, even without compressed bitmaps. Here the $O(\sigma \log n)$ term also includes the table of symbols. In this case, if underlying bitmaps are supporting $O(1)$ operations, the WT with Huffman shape has an average cost of $O(H_0(T) + 1)$ per operation. The worst case can be guaranteed to be $O(\log \sigma)$ if after a $\gamma_{\text{huff}} \log \sigma$ levels the Huffman tree is balanced, where γ_{huff} is a positive constant, such that WT's has $O(\log \sigma)$ levels. On large σ values, this cost can be of worth.

Golinsky et al. [Golynski et al., 2006] presents an uncompressed index requiring $n \log \sigma + o(n \log \sigma)$ bits, designed primarily for large alphabets. The technique unravels the sequence and partitioning each σ positions, transforming T into $\frac{n}{\sigma}$ blocks. The i-th block is a $\sigma \times \sigma$ binary matrix with σ enabled bits (a single 1 per column), represented with the tuple (π_i, M_i). Where π_i is a permutation and M_i is a bitmap of 2σ bits. They are created as follows. We review the matrix in row-major order, we append to π_i the positions holding an enabled bit, we place a 0 in M_i whenever we start a row, and add as many 1's as the number of 1's in that row. π_i is represented with the indexed permutations presented by Munro et al. [Munro et al., 2003],

being able to represent permutation and its inverse using $(1+\epsilon)\sigma \log \sigma$ bits, while solve $\pi_i(j)$ in $O(1)$, and $\pi_i^{-1}(j)$ in $1/\epsilon$ for some $0 < \epsilon \leq 1$. For practical purposes, we concatenate all M_i bitmaps in $M = M_1 M_2 \cdots M_{n/\sigma}$. Additionally, we store a bitmap $L = L_1 L_2 \cdots \mathrm{L}_\sigma$ where each bitmap L_c is created traveling the entire unraveled sequence in row-major order storing a 0 whenever we enter a block followed by the number of 1's in the current block of the c-th symbol. Both M and L must support Rank, Select, and Access queries. Then, bitmaps M and L requires$2n + o(n)$ bits for each one, and all the permutations requires $n \log \sigma$ bits plus $\frac{n}{\sigma} o(\sigma \log \sigma)$ bits to store a Y-fast trie, necessary to solve Rank_c in its promised time. Access requires to inverse of the permutation [Munro et al., 2003]. Both Rank_c and Access are solved in $O(\log \log \sigma)$ time. Also, $\mathsf{Select}_c(T, r)$ is solved in constant time. This index is important for us since our metric indexes hold large alphabets and Select_c is the most frequent operations.

Claude and Navarro [Claude and Navarro, 2008] report a practical implementation of the GMR06 index, using a cyclic representation by Munro et al. [Munro et al., 2003], and binary searching to solve Rank_c. Thus, Access achieves a worst case time of $O(1/\epsilon)$, Rank_c in $O(\log \sigma)$, and Select_c still need $O(1)$ time. Basically, this implementation uses less memory due to the deletion of the Y-fast trie. However, we observed a low locality in our access pattern (mostly based on iterate Select consecutively, Section 6.3.3) in the index of Golynski et al. We perform an extensive set of experimental evidence supporting our claims in Section A.4.

The following sections expose our contributions. Firstly, we describe our indexed bitmaps, they are a fundamenta brick in our family of indexed sequences, to be exposed later.

A.2 Sets as Lists of Differences

A set $L \subseteq \{1, 2, 3, \cdots, n\}$ is straightforwardly represented using an indexed bitmap. In a similar way, all bitmap primitives are easily represented with a set and a few basic operations. Let the set L be stored as a sorted list of integers. $\mathsf{Select}_1(L, r)$ is equivalent to access to the r-th item on L with $\mathsf{Select}_1(L, 0) = 0$. $\mathsf{Rank}_1(L, p)$ is computed searching in L the insertion position of p with $\mathsf{Rank}_1(L, 0) = 0$. Others operations are computed using the existing relations with these primitives. In other words, the sorted list is seen as the response to all possible Select_1 calls, as SArray [Okanohara and Sadakane, 2007].

We create three bitmaps based on represent a sorted list in a compressed

way. The first approach dubbed DiffSet (DSet), is a simple compression of the sorted list using differences plus absolute values at each B positions of the sorted list. The second one enhances DiffSet under the premise of existence of *runs* in the sorted list (zones of consecutive items). This regions are specially encoded with a run-length scheme. This approach is named DiffSet-RL. Finally, we assume that *runs* can be quite large and frequent.We dubbed this bitmap as DiffSet-RL2.

All of these bitmaps shares its worst case complexities, that is Select_0, Rank_c, and Access in $O(B + \log \frac{n}{B})$; Select_1 is solved in at most $O(B)$ steps, $B \geq 1$. Finally, Select_1 over s consecutive arguments is solved in $O(B + s)$ steps. So, each call is solved in $O(\frac{B+s}{s})$ steps, in amortized time over s. If $B = O(s)$ we obtain $O(1)$ amortized time for s consecutive Select_1 calls. This time is obtained using pointers to previous Select_1 results.

Their worst case memory consumption is $nH_0(L)+2n_1 \log\log n/n_1)$ bits. Our run-length bitmaps are *opportunistic*, in the sense that they take speed up operations when small local H_0 is found. Also, they achieve smaller spaces than the worst case.

A.2.1 DiffSet bitmap

A sorted list encoded with differences is just the list of differences between consecutive entries, as shown in the example of Figure 9.1. Each difference is encoded using a prefix free integer encoding, like Elias-γ or Elias-δ [Elias, 1975]. Elias-γ is a variable-length integer encoding using $2\log(x+1)$ bits to encode an integer x. The idea is to represent x with $\log(x+1)$ bits, also we need to represent the length of the code in unary. It uses less space than fixed-length binary encoding (which uses $\log n$ bits) if $x \leq \sqrt{n/2}$. We have s integers in the range 1 to n ($s = n_1$ for bitmaps). In the worst case each difference is n/s and we need twice the optimal number of bits, $2s \log n/s + 2s$. This worst case can be of worth on large integers. For these cases, Elias-δ encodes an integer x in $\log(x+1) + 2\log\log(x+1)$. x is represented in $\log(x+1)$ and the number of bits is encoded with Elias-γ.

All encoded integers are concatenated in a memory area L', and a pointer to L' is set every B positions of the original list L. So, we need $(s/B)\log n$ bits for these pointers. Let $B = \Theta(\log n)$. Accessing the i-th integer then costs $O(\log n)$ decodings. Also, L' needs $s/\Theta(\log n) = o(s)$ absolute samples. Each sample needs $\log n$ bits, if samples are explicitly represented. Summing up $2s$ bits for pointers (on L') and absolute samplings. Notice that L' and absolute samplings are sorted lists too, so, they can be represented in close to optimal space using SArray.

Finally, using Elias-δ encoding, our DiffSet bitmap requires $n_1 \log \frac{n}{n_1} + 2n_1 \log\log \frac{n}{n_1} + 2n_1 + o(n_1) = n_1 \log \frac{n}{n_1} + O(n_1 \log\log \frac{n}{n_1})$ bits, in the worst case (with ones uniformly distributed along the bitmap).

A.2.2 DiffSet + Run-Length

In a way, differential encoding of a sorted list represents *runs* with unary coding, thus for long runs this method is suboptimal. A better option is to encode the length of the run using either Elias-γ or Elias-δ code, yielding to DiffSet-RL. As in DiffSet, we use regular samplings to get fast access to the i-th integer. Figure 9.1 shows an example of run-length encoding of the inverted index, where only differences of 1's are run-length encoded as a tuple (1, length). Since we always decode from left to right it is simple to mix differences with run-length encodings. If an absolute sample falls inside a sample, the run is cut. This is suboptimal in space, but allows decompression without binary searching to locate the actual position.

A natural optimization is introduced as follows: if the j-th and the $(j+1)$-th absolute samples are separated by exactly B positions we say that the range is *filled*, and no representation of the data inside the range is necessary; just the sampling data is stored. We dubbed this variant as DiffSet-RL2.

Notice that if the i-th integer lies in a filled range, it is decoded in constant time. In the same way, even when the worst case requires B decompressions of items, the average time is way smaller if runs are found in the road to the desired integer. On both DiffSet-RL and DiffSet-RL2, a run being in the road means to advance its length in constant time.

The run-length increments the necessary space in 1 bit when runs are not present (adding n_1 bits to the space cost in the worst case).

A.3 Indexing Sequences with a Single Permutation

Let us start with an unraveled sequence, thus each symbol is a row in a $n \times \sigma$ matrix. Rank_c and Select_c are solved using the bitmap (row) corresponding to symbol c. As detailed in Section A.1.2, the biggest problem comes from Access since it needs $O(\sigma)$ calls to row-bitmap's Access. Furthermore, in practice $O(\sigma \log n)$ bits are necessaries to store data-structure's headers.

Our index is dubbed as Extra Large Bitmap (XLB) by reasons that will be evident below. XLB is based on the well known observation that a

row major order traversal on Select_c produces a permutation Π of $[1,\ldots,n]$. Items corresponding to a single row are numerically sorted. This permutation can be encoded into a single bitmap $P[1,\sigma n]$ with n ones replacing each entry of Π by $cn+\Pi(i)$. Using this technique, we do not need to store a header per symbol. However, other problems arise since P has the following memory requirements under global statistics.

$$\log\binom{\sigma n}{n} = n\log\sigma + n\log e \tag{A.3}$$

$$= n\log\sigma + O(n) \text{ bits} \tag{A.4}$$

In particular, if P is indexed using SArray the necessary space is $n\log\sigma + O(n)$ bits. Also, DiffSet produces $n\log\sigma + O(n\log\log\sigma)$ bits in the worst case. Nevertheless, bitmaps achieving local entropy as DiffSet, DiffSet-RL, and DiffSet-RL2 in fact achieve $nH_0(T)+O(n\log\log\sigma)$ since they adapt to the local statistics ignoring the global ones.

In general P should be indexed with a compressed bitmap achieving local entropy, and with an space dependent on the number of ones, not in the length of the bitmap in both major and minor terms. If the bitmap only achieves global entropy, then the major term is at least of $n\log\sigma$.

Until now, we have encoded the matrix in a single large bitmap. With this machinery, Rank_c is solved as follows.

$$\mathsf{Rank}_c(T,p) = \mathsf{Rank}_1(P,cn+p) - \mathsf{Rank}_1(P,cn-1) \tag{A.5}$$

On the other hand, Select_c has a similar procedure.

$$\mathsf{Select}_c(T,r) = ((\mathsf{Select}_1(P,r+\mathsf{Rank}_1(P,cn-1))-1) \mod n)+1 \tag{A.6}$$

As with unraveled sequences, the main problem is Access, but since Π is present, the solution can be stated in terms of Π^{-1} (similarly to Golynski et al. [Golynski et al., 2006]).

$$\mathsf{Access}(T,p) = \left\lceil \frac{\mathsf{Select}_1(P,\Pi^{-1}(p))}{n} \right\rceil \tag{A.7}$$

However, Π and Π^{-1} are not directly available, but they are computed in the following way.

$$\Pi(i) = \mathsf{Select}_1(P,i) \mod n \tag{A.8}$$

We compute Π^{-1} using the cyclic structure of the permutation with the structure of Munro et al. [Munro et al., 2003]. Thus, we require at most

$\frac{n}{t}\log n$ bits to solve Π^{-1} in at most t accesses to Π, with $t \geq 1$. Finally, if $t = \Theta(\log n)$ then the memory cost becomes $o(n)$ while solves $\Pi(i)$ in $O(\log n)$ calls to $\mathsf{Select}_1(P, \cdot)$.

A.3.1 Efficient Access on Unraveled Sequences

The previous technique has a straightforward application to Unraveled Sequences. For both Rank_c and Select_c the computation is simplified since instead of compute the number of ones until a symbol region, the involved bitmap (row in a matrix) is selected and the desired operation is directly performed on it.

Access needs Π and Π^{-1} to be solved. Π^{-1} is solved using the same idea of XLB, that is, taking advantage of the ciclic structure of the permutation. Π is computed with an additional bitmap X, created in a row-major traversal: each time we enter into a row we append a 0 into X, then we write 1 as many 1's are found in that row. X requires $n+\sigma+o(n+\sigma)$ bits or $\log\binom{n+\sigma}{n}$ bits if a compressed bitmap is applied. $\Pi(p)$ is computed using X and the unraveled sequence I (indexed by symbol) as follows:

$$\Pi(p) = \mathsf{Select}_1(I_{\mathsf{sym}}, p - \mathsf{Select}_0(X, \mathsf{sym}) + \mathsf{sym}) \tag{A.9}$$

Where $\mathsf{sym} = \mathsf{Select}_1(X, p) - p$. Finally, Access is detailed in Equation A.10.

$$\mathsf{Access}(T, p) = \mathsf{Rank}_0(X, \mathsf{Select}_1(X, \Pi^{-1}(p))) \tag{A.10}$$

This technique becomes unfeasible for sequences with very large alphabet, e.g., $\frac{n}{\sigma} = O(1)$. The complication arises because of data-structure's headers. Notice that this breaking point varies depending on the underlying bitmaps.

A.4 Experimental Results

We tested several generated sequences with $n = 2^i$ for $i = 20, 22, 24, 26$; and $\sigma = 2^j$ for $j = 6, 8, 10, 12, 16, 18, 20$. With these setups we generate two kinds of random sequences.

- Uniform random sequences. For each position, a randomly selected symbol is placed.
- Skewed random sequences. For each position, a symbol is selected with probability p_{prev} to be equal to the previous symbol (the first

symbol is uniformly selected). The objective is to induce *runs* into the sequence. We generate sequences for p_{prev} of 0.9 and 0.99.

Each query uniformly selects symbols and arguments from the valid ranges, i.e., $pos \in [1, \ldots, n]$, $rank_c \in [1, \ldots, \mathsf{Rank}(T, n)]$, and $symbol \in [1, \ldots, \sigma]$. Our time performances were averaged with 50000 calls. The same setup is valid for symbols on consecutive Select calls, yet rank values are consecutive, and the number of consecutive arguments is the entire set of possible arguments of Select.

Our main set of results are exposed on terms of n and σ. In the first set of experiments, we plot the operation's time cost against the memory of the indexes (implicit n). We show figures for three different σ values, each row fixes σ. On the second part, we expose the performance in the same explicit terms, operation's time cost and memory, but memory cost is implicitly showing σ. On this set of experiments we show three different values of n, one fixed value per row.

All experiments were performed in a workstation with Intel(R) Xeon(R) CPU E5462 @ 2.80GHz, with eight cores (two quad-core processors), and 2GiB of main memory. The workstation runs the 9.8.0 Darwin Kernel. All indexes and databases were stored in main memory. Our implementations was written in the C# programming language and on the Mono framework, `www.mono-project.com`.

We presented an experimental comparison among our index and several state of the art alternatives. Also we present some new variants of these alternatives. The indexes being under review and comparison are the following ones.

— **GMR06**. The index of Golynski et al. [Golynski et al., 2006] with the following variants.

  – **GMR06**. The original index, as implemented by Claude and Navarro [Claude and Navarro, 2008].
  – **GMR06-RL**. A variant of GMR06 with an special encoding of the runs. The idea is to include an additional bitmap marking runs, such that only run's header is stored.
  – **GMR06-32**. A variant of GMR06 using 32 bit integers for each entry in the permutations instead of $\log \sigma$.

— **Wavelet Tree (WT)** with the following flavors.

  – **WT GGMN05**. WT using GGMN05 as backed bitmap.

  - **WT RRR02**. WT using RRR02 as backed bitmap.
  - **WT RRR02v2**. WT using new variant of RRR02, the idea is to represent the class's identifier as a variable length code. The classes are now sorted by its cardinalities. Pointers to starting positions are stored synchronously with Offsets.

— **XLB**. This is our main contribution, we study specializations for **DiffSet**, **DiffSet-RL**, **DiffSet-RL2**, and **SArray** bitmaps. In the second experimental part, we fix $B = 31$ for DiffSet-based bitmaps. So, we obtained a fast index but with large memory requirements, as shown in the previous set of experiments. Even with these extra cost our index performs quite good.

Notice that bitmaps SArray, DiffSet, DiffSet-RL, and DiffSet-RL2 were implemented supporting fast consecutive Select values storing pointers to previous values such that the next call of Select is efficiently computed in $O(1)$ time. It is well known that WT can store pointers to previous values, yet the technique's implementation is quite sophisticated, contrasting with the simple technique required by XLB. The reason is that instead of a complex index of sequences like WT, our XLB is just a large bitmap. On the other side, GMR06 (and its variants) does not suffer any modification since it already requires $O(1)$ time for Select.

A.4.1 XLB with DiffSet Bitmaps

Preprocessing Cost

Firstly we describe properties and performances of XLB with DiffSet like bitmaps. These experiments show both the behavior of the DiffSet bitmap (and its variants) as a unit, and the XLB sequences with a bitmap reaching local entropy. Figure A.1 shows the construction time. Uniform datasets are shown in left column. Each row has a fixed σ value. These bitmaps have not runs, so all DiffSet variants are performing similar. The performances are grouped by block size B. Notice that increasing σ does not imply a change on the construction performance. On the other extreme, Skewed 0.99 (rightmost column) is a bitmap with many runs inside. Here, plain DiffSet bitmap has the most expensive construction. This is behavior is expected since DiffSet-RL and DiffSet-RL2 are handling runs efficiently in both space and time terms. Again, σ changes are not noticeable in the construction time. The middle column, Skewed 0.9, exposes a transition

performance between Uniform and Skewed 0.99. So, the cost dependency of DiffSet-RL and DiffSet-RL2 on the number of runs is noticeable.

Performance of Access

The average time cost of Access operation and the necessary memory of the index are shown in Figure A.2. Each point corresponds to the average time required to solve Access and the size in KiB of the index solving the operation. The size is related to n. Uniform sequences, left column, are directly depending on B, ignoring σ. DiffSet curves are consistently faster than run-length based bitmaps since the decoding operation is simpler. Also, the required memory is quite similar among all due to our log scale.

Column corresponding to Skewed 0.9 show a great speed and memory cost reduction. This is because the existence of small runs. The performance of all methods are quite similar, since runs are not too large to be of use for run-length based bitmaps.

On the other side, Skewed 0.99, a large improvement for run-length based indexes is found (rightmost column, Figure A.2). Here DiffSet have the smallest speed up, and DiffSet-RL2 the biggest one. The memory cost is reduced significantly for both DiffSet-RL and DiffSet-RL2, due to the better encoding of runs.

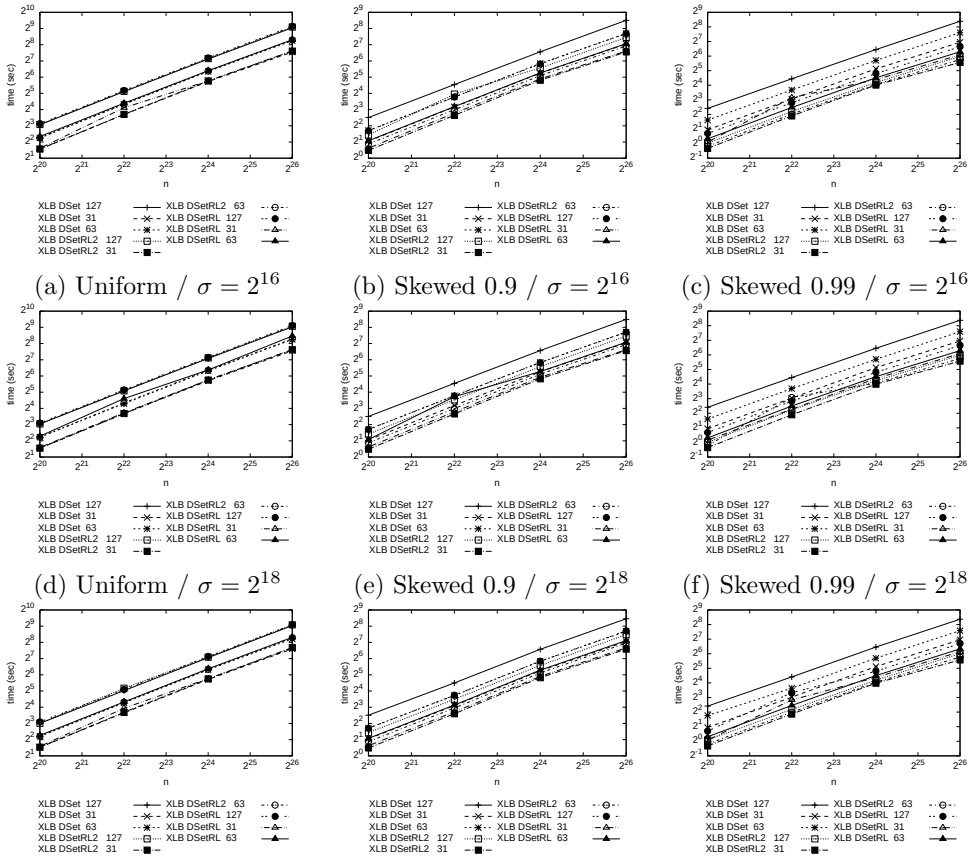
Performance of Rank

Figure A.3 shows the performance of Rank_c. Uniform datasets are barely affected by σ, and have the worst performances since the bitmaps do not contain runs. Column showing Skewed 0.9 datasets have a much better performances than the first column, here smaller indexes are produced by large B values, while better times are achieved by small B values and run-length based indexes.

Finally, Skewed 0.99 sequences improve their time and memory performances using run-length bitmaps. The improvements are particularly important on time, being several times faster than the plain XLB DiffSet.

Performance of Select

Figure A.4 shows the average time per select operation on several sequences of increasing size. Figures with the uniform dataset show a time performance dependent on B, yet a clear speed up arise as σ grows (on small n). The cause of this behavior is that the number of induced partitions by symbol increases

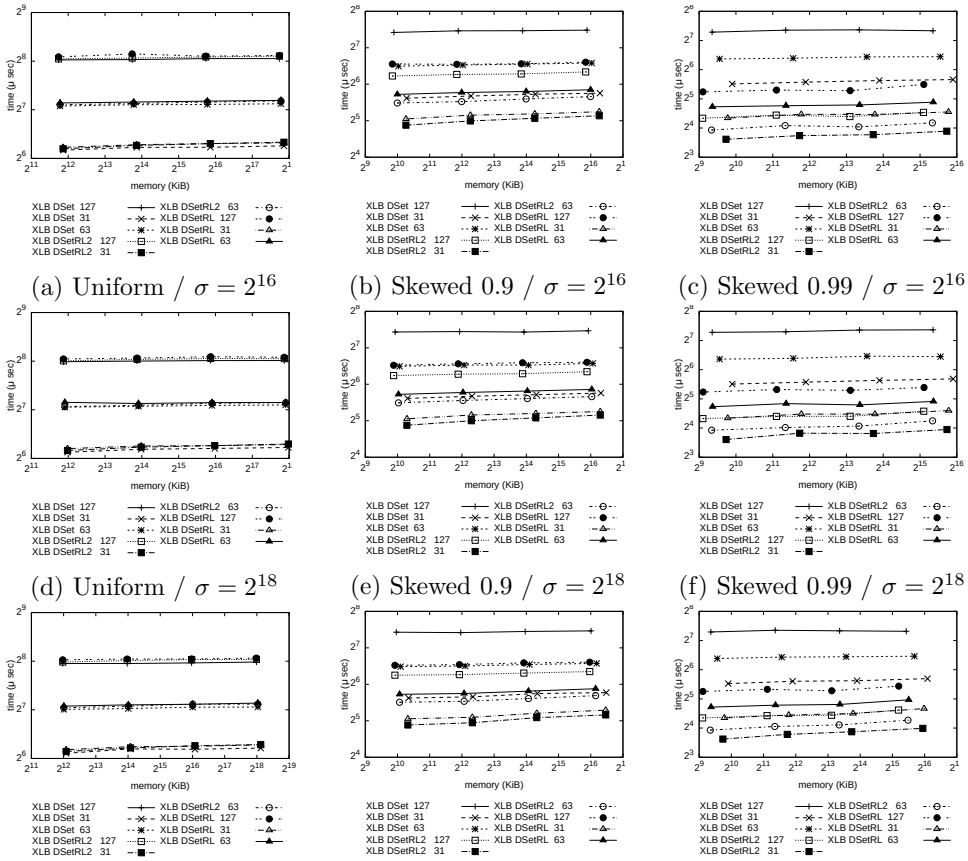


(a) Uniform / $\sigma = 2^{16}$ (b) Skewed 0.9 / $\sigma = 2^{16}$ (c) Skewed 0.99 / $\sigma = 2^{16}$

(d) Uniform / $\sigma = 2^{18}$ (e) Skewed 0.9 / $\sigma = 2^{18}$ (f) Skewed 0.99 / $\sigma = 2^{18}$

(g) Uniform / $\sigma = 2^{20}$ (h) Skewed 0.9 / $\sigma = 2^{20}$ (i) Skewed 0.99 / $\sigma = 2^{20}$

Figure A.1: Construction time on sequence indexes for several (n, σ) setups.

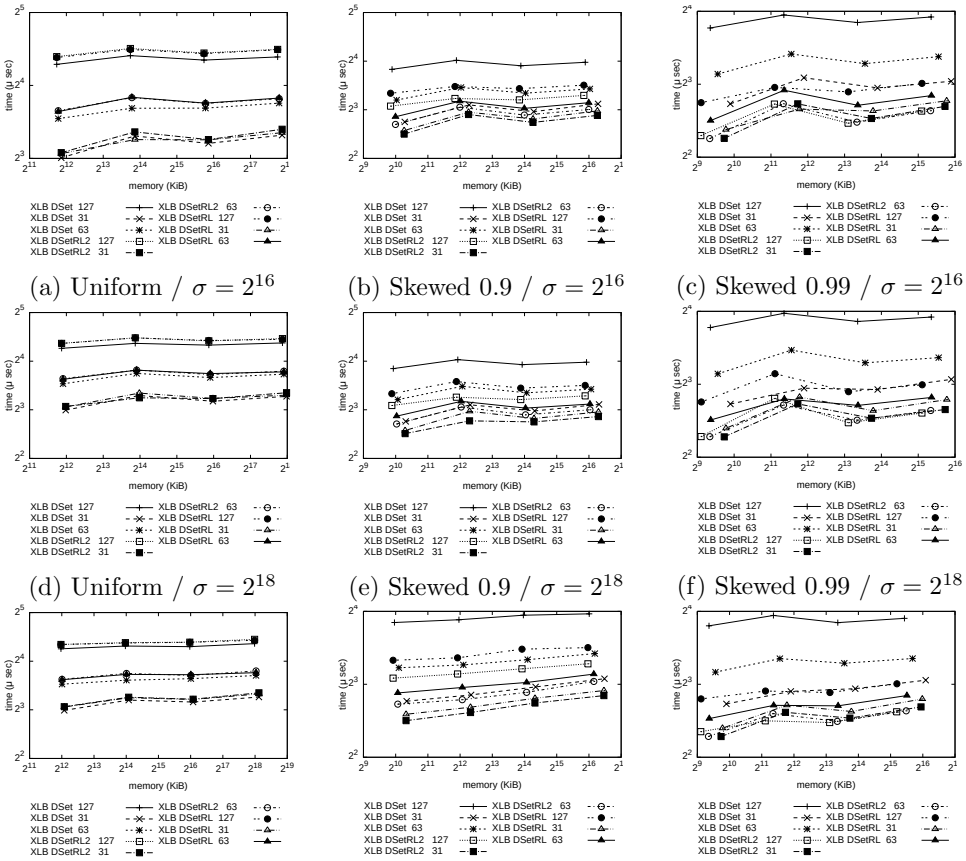


(a) Uniform / $\sigma = 2^{16}$ (b) Skewed 0.9 / $\sigma = 2^{16}$ (c) Skewed 0.99 / $\sigma = 2^{16}$

(d) Uniform / $\sigma = 2^{18}$ (e) Skewed 0.9 / $\sigma = 2^{18}$ (f) Skewed 0.99 / $\sigma = 2^{18}$

(g) Uniform / $\sigma = 2^{20}$ (h) Skewed 0.9 / $\sigma = 2^{20}$ (i) Skewed 0.99 / $\sigma = 2^{20}$

Figure A.2: The cost of Access on sequence indexes for several (n, σ) setups.

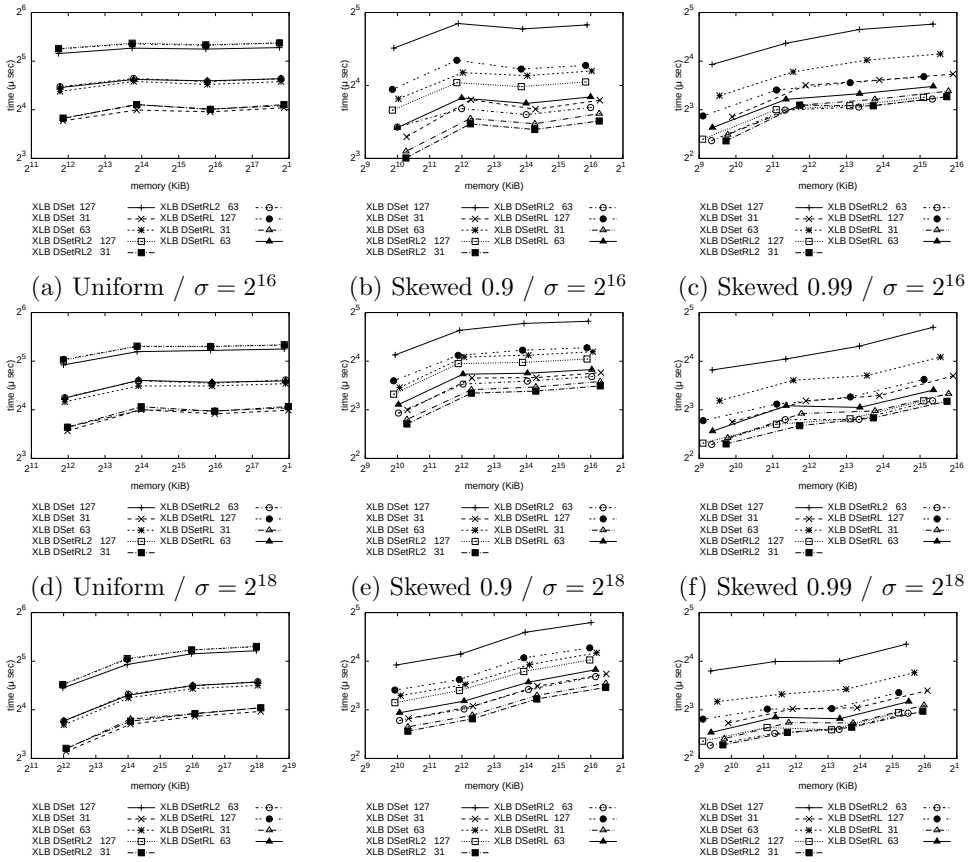


(a) Uniform / $\sigma = 2^{16}$ (b) Skewed 0.9 / $\sigma = 2^{16}$ (c) Skewed 0.99 / $\sigma = 2^{16}$

(d) Uniform / $\sigma = 2^{18}$ (e) Skewed 0.9 / $\sigma = 2^{18}$ (f) Skewed 0.99 / $\sigma = 2^{18}$

(g) Uniform / $\sigma = 2^{20}$ (h) Skewed 0.9 / $\sigma = 2^{20}$ (i) Skewed 0.99 / $\sigma = 2^{20}$

Figure A.3: The cost of Rank on sequence indexes for several (n, σ) setups.



(a) Uniform / $\sigma = 2^{16}$ (b) Skewed 0.9 / $\sigma = 2^{16}$ (c) Skewed 0.99 / $\sigma = 2^{16}$

(d) Uniform / $\sigma = 2^{18}$ (e) Skewed 0.9 / $\sigma = 2^{18}$ (f) Skewed 0.99 / $\sigma = 2^{18}$

(g) Uniform / $\sigma = 2^{20}$ (h) Skewed 0.9 / $\sigma = 2^{20}$ (i) Skewed 0.99 / $\sigma = 2^{20}$

Figure A.4: The cost of Select on sequence indexes for several (n, σ) setups.

with fixed n. In other words, in many cases the expected $\mathsf{Rank}_c(T, n) = n/\sigma$ is smaller than B.

The time cost of Skewed 0.9 sequences are close to an half of the uniform ones. Memory is several times smaller. Here the speed up is given mostly by B, however run-length bitmaps are always faster.

The last column shows the behavior for Skewed 0.99 sequences. Both speed up and memory performances are greatly improved, however the differences are more noticeable than those found Skewed 0.9 column. The effect of σ in the performance is barely noticeable.

Performance of Select with Consecutive Arguments

The performance of Select with consecutive arguments is shown in Figure A.5. Here, a huge time improvement is achieved compared to random arguments on Select.

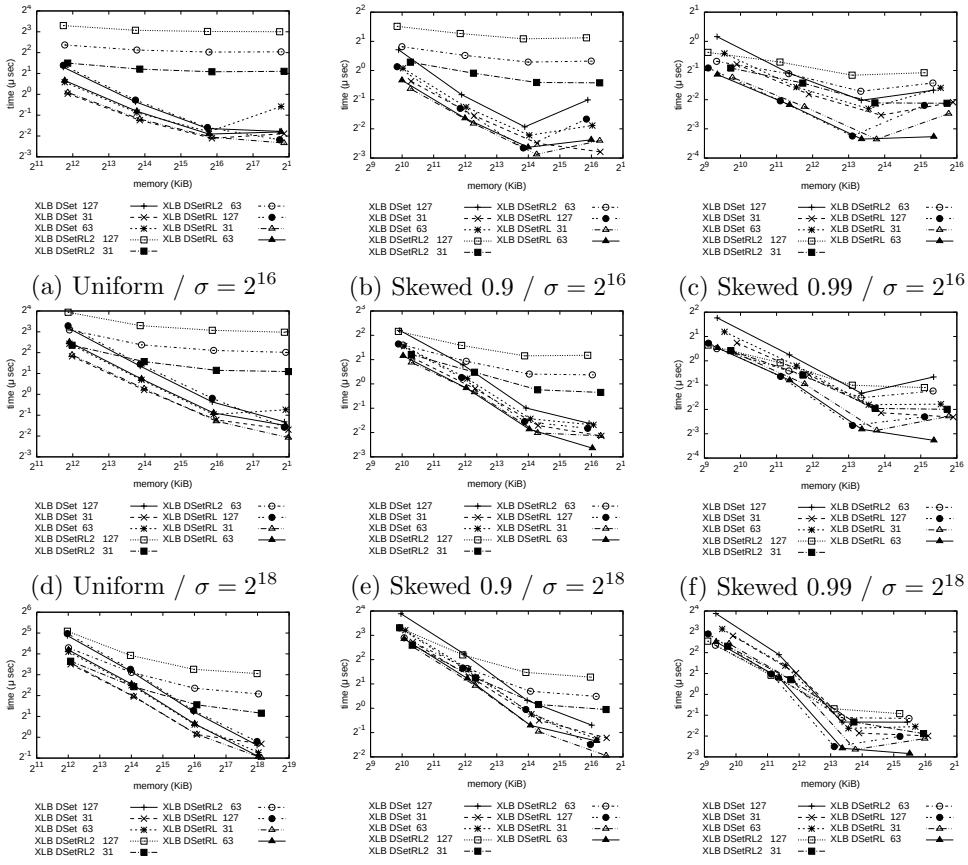
Please remember that consecutive arguments for Select in XLB indexes is easily implemented in amortized time. For a fixed σ (row), we obtain better times on large n values because of the cost is amortized. In contrast to random arguments, the effect of σ is inversely on small n values, i.e., occurrences of a symbol do not provide enough support to amortize calls.

A.4.2 Comparison Against other Techniques

We have reviewed the behavior of DiffSet, DiffSet-RL, and DiffSet-RL2 with XLB. However, the former study give us an isolated understanding of technique, we learned about B and the technique's behavior on n and large σ. Now we will discuss the performance among our index, GMR06, and WT techniques.

Performance of the Preprocessing Step

The construction time for sequences of several lengths is presented in Figure A.7. Here, WT based indexes have the slower constructions, and the more robust to n are GMR06 based indexes. On the other side, XLB-SArray has the best performance for Uniform and Skewed 0.9. On the Skewed 0.99 column, the best times are performed by XLB with run-length bitmaps, yet XLB-SArray performs quite good.

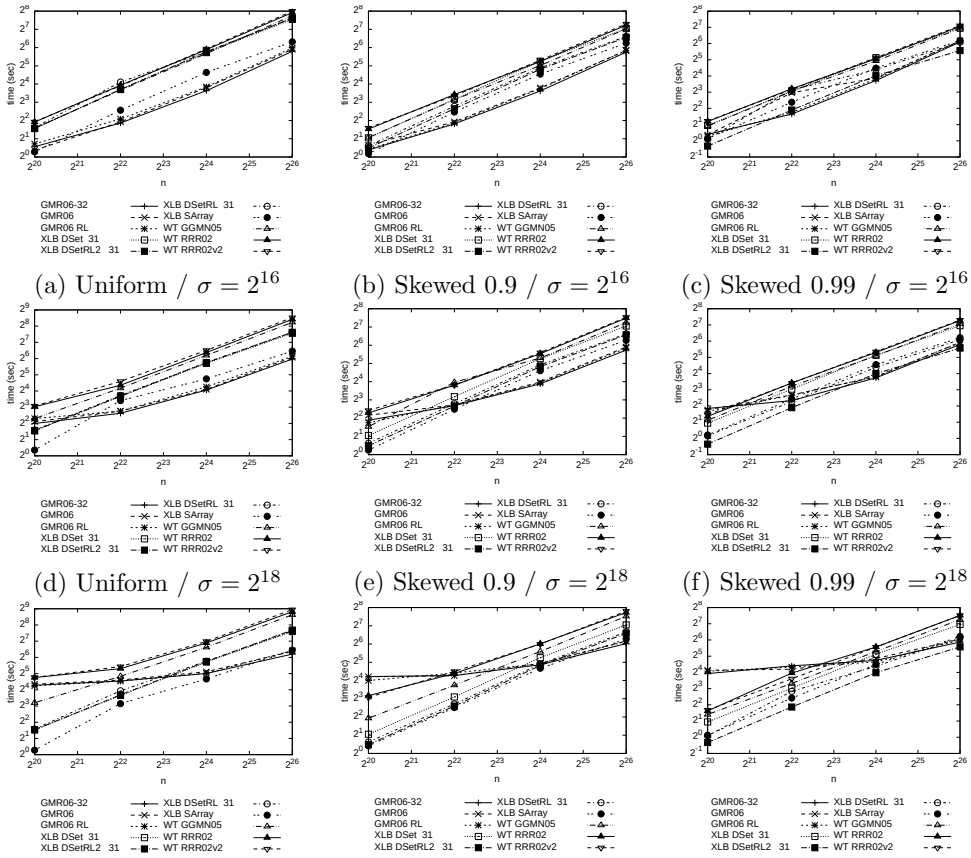


(a) Uniform / $\sigma = 2^{16}$ (b) Skewed 0.9 / $\sigma = 2^{16}$ (c) Skewed 0.99 / $\sigma = 2^{16}$

(d) Uniform / $\sigma = 2^{18}$ (e) Skewed 0.9 / $\sigma = 2^{18}$ (f) Skewed 0.99 / $\sigma = 2^{18}$

(g) Uniform / $\sigma = 2^{20}$ (h) Skewed 0.9 / $\sigma = 2^{20}$ (i) Skewed 0.99 / $\sigma = 2^{20}$

Figure A.5: The cost of consecutive Select on sequence indexes for several (n, σ) setups.

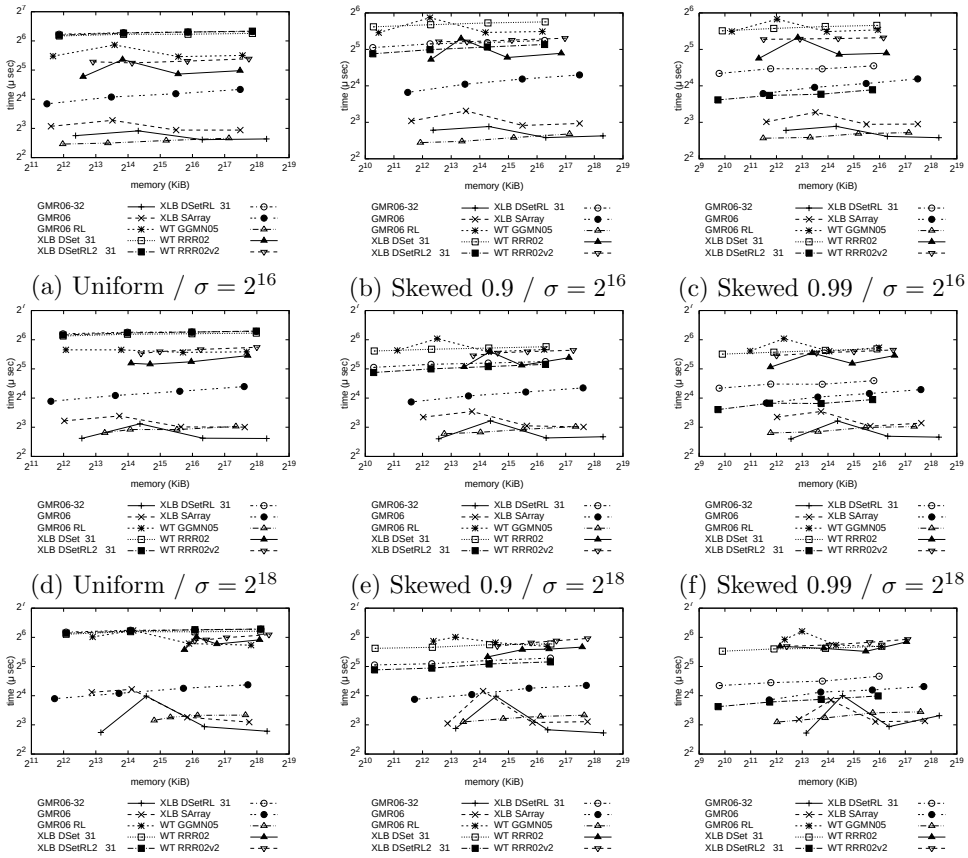


(a) Uniform / $\sigma = 2^{16}$ (b) Skewed 0.9 / $\sigma = 2^{16}$ (c) Skewed 0.99 / $\sigma = 2^{16}$

(d) Uniform / $\sigma = 2^{18}$ (e) Skewed 0.9 / $\sigma = 2^{18}$ (f) Skewed 0.99 / $\sigma = 2^{18}$

(g) Uniform / $\sigma = 2^{20}$ (h) Skewed 0.9 / $\sigma = 2^{20}$ (i) Skewed 0.99 / $\sigma = 2^{20}$

Figure A.6: Construction time



(a) Uniform / $\sigma = 2^{16}$ (b) Skewed 0.9 / $\sigma = 2^{16}$ (c) Skewed 0.99 / $\sigma = 2^{16}$

(d) Uniform / $\sigma = 2^{18}$ (e) Skewed 0.9 / $\sigma = 2^{18}$ (f) Skewed 0.99 / $\sigma = 2^{18}$

(g) Uniform / $\sigma = 2^{20}$ (h) Skewed 0.9 / $\sigma = 2^{20}$ (i) Skewed 0.99 / $\sigma = 2^{20}$

Figure A.7: Access time

Performance of Access

Figure A.7 shows the average time of Access. The XLB-SArray produces the smaller indexes for uniformly distributed sequences and still hold a competitive time cost. On the same column, the best times are performed by GMR06-32, GMR06, and WT-GGMN05. XLB with DiffSet-RL and DiffSet-RL2, and WT with RRR02 and RRR02v2 are performing quite bad on sequences with uniform distribution. The reason behind bad performances are that these methods reach their worst cases on the uniform distribution.

The right most column, Skewed 0.99, shows a great improvement of XLB with run-length, and WT with RRR02 and RRR02v2 methods, in both time and memory requirements. Nevertheless, the enhancements are larger for XLB methods. Notice that GMR06 methods have a low sensitivity to the distribution, yet GMR06-RL reduces its size in a similar behavior to other run-length indexes.

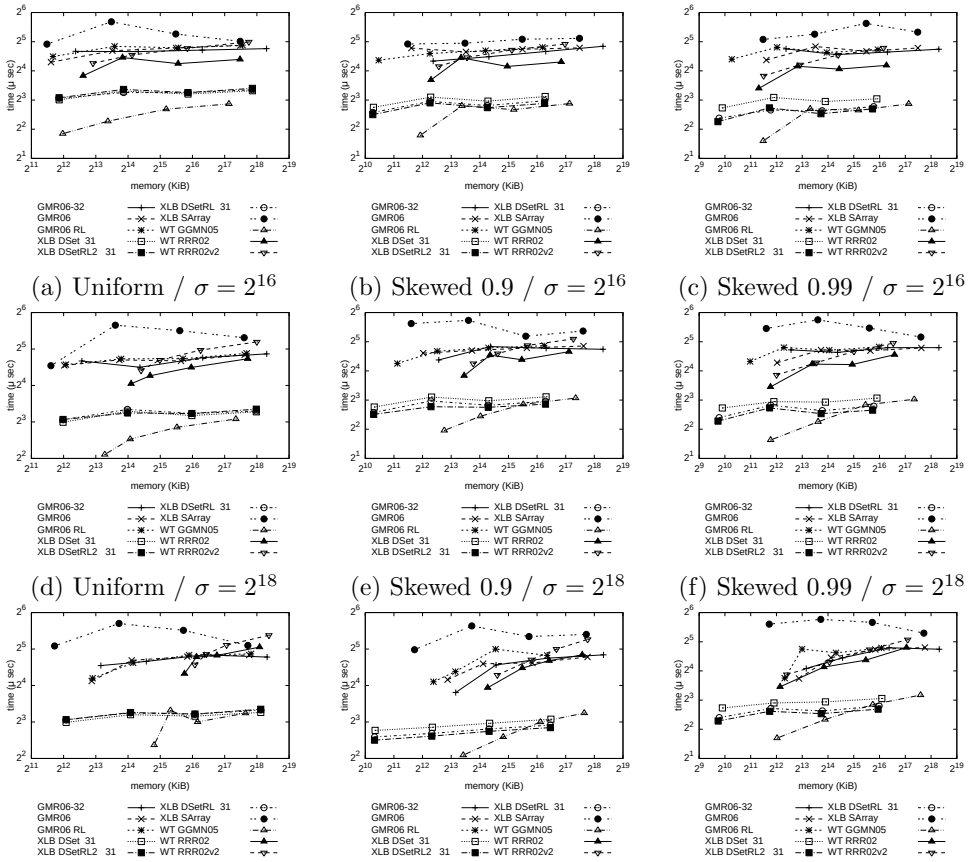
The middle column basically is a transition between the commented extremes. Here, XLB-SArray remains as an equilibrated alternative to the smaller run-length indexes and the faster GMR06 methods.

Performance of Rank

Figure A.8 shows the average time of Rank. Here, XLB-SArray is the slower alternative, but produces the smallest indexes on the uniform column. For any other setup, XLB with DiffSet (and its run-length variants) is the best choice: they are the smaller and faster indexes, only WT-GGMN05 is faster for small n.

Performance of Select

Select's performance is presented in Figure A.9. Note that GMR06 based methods and XLB methods are barely sensitive to n, contrasting to WT methods. Here, XLB with DiffSet methods are the faster and smaller indexes. Nevertheless, we observed that WT can be really fast for small sequences. As in other operations, XLB-SArray performs quite good on uniform distributions.

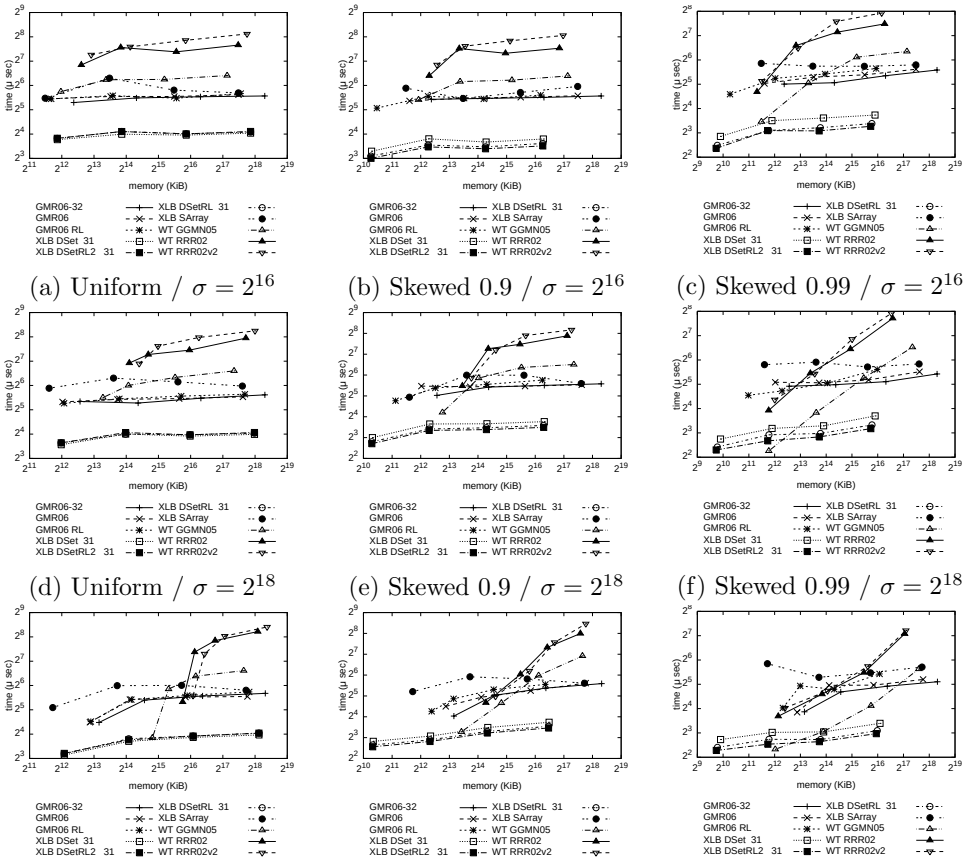


(a) Uniform / $\sigma = 2^{16}$ (b) Skewed 0.9 / $\sigma = 2^{16}$ (c) Skewed 0.99 / $\sigma = 2^{16}$

(d) Uniform / $\sigma = 2^{18}$ (e) Skewed 0.9 / $\sigma = 2^{18}$ (f) Skewed 0.99 / $\sigma = 2^{18}$

(g) Uniform / $\sigma = 2^{20}$ (h) Skewed 0.9 / $\sigma = 2^{20}$ (i) Skewed 0.99 / $\sigma = 2^{20}$

Figure A.8: Rank time



(a) Uniform / $\sigma = 2^{16}$ (b) Skewed 0.9 / $\sigma = 2^{16}$ (c) Skewed 0.99 / $\sigma = 2^{16}$

(d) Uniform / $\sigma = 2^{18}$ (e) Skewed 0.9 / $\sigma = 2^{18}$ (f) Skewed 0.99 / $\sigma = 2^{18}$

(g) Uniform / $\sigma = 2^{20}$ (h) Skewed 0.9 / $\sigma = 2^{20}$ (i) Skewed 0.99 / $\sigma = 2^{20}$

Figure A.9: Select time

Performance of Select with Consecutive Arguments

Figure A.10 shows the performance of Select with consecutive arguments. Here, XLB methods are the smaller and faster indexes. The worst performances are found for compressed WT sequences. Also, GMR06 based indexes kept a similar performance for all setups.

A.4.3 Dependency on the Alphabet's Size

Until now, we have exposed the behavior of our XLB indexes for relatively large alphabets and several setups on n. In this set of experiments we will use a bigger range of alphabet's sizes, covering smaller alphabets. We use the same indexes of Section A.4.2. Our main variable in figures is σ, encoded as the size of the index. Also, figures in a row shares a fixed n (2^i for $i = 22, 24, 26$), and columns are denoting randomly generated sequences with the same parameters. The memory axis wrap σ values corresponding to $\sigma = 2^j$ for $j = 6, 8, 10, 12, 16, 18, 20$.

Performance of the Preprocessing Step

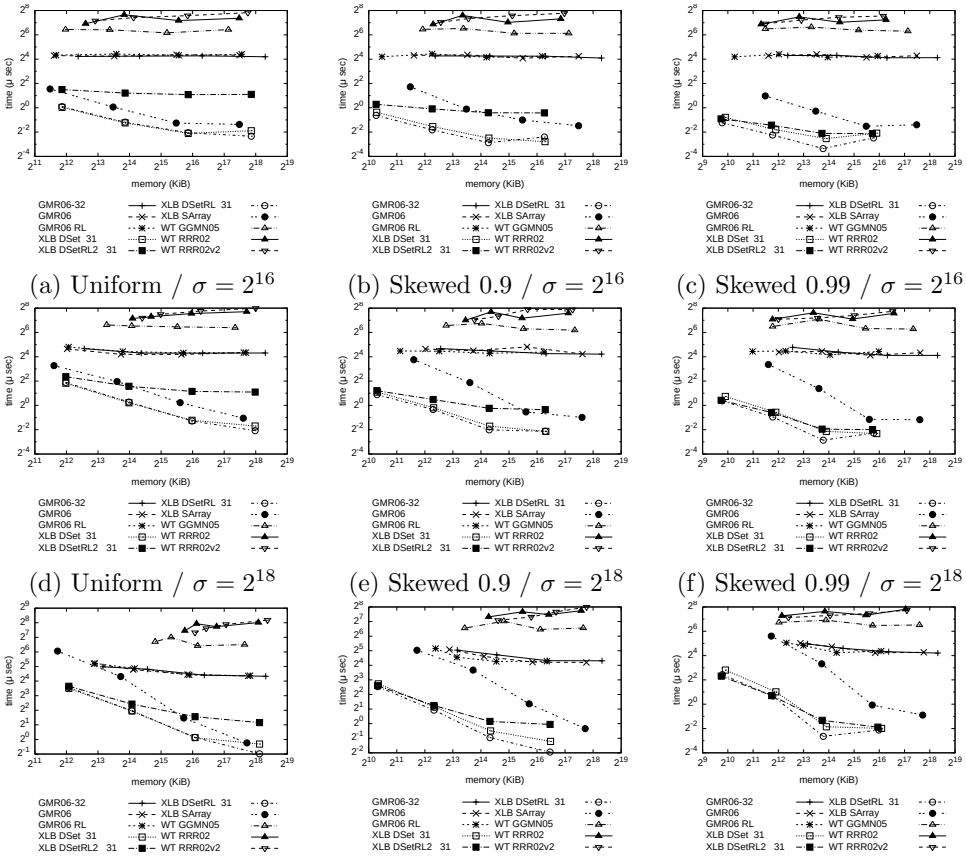
XLB methods have its construction time barely affected by σ. This contrast with GMR06 and WT based indexes, that are quite dependent on σ, they are pretty fast for small alphabets and slower on large ones. This is true for small sequences, for large ones XLB methods become faster. The speed of GMR06 and WT based methods rapidly degrades as n grows.

Notice that indexes with run-length based bitmaps are faster as the number of runs increase (Skewed 0.9 and 0.99).

Performance of Access

Figure A.12 compares Access performance for our testing methods. The best performance for Uniform sequences are the WT indexes. Particularly, the compressed WT indexes are quite good for small σ, and they degrade their performance as σ grows.

The existence of runs has a big impact on XLB DiffSet methods, as is shown in the middle column of Figure A.12. Here, both time and memory are barely varying with σ. On this column, WT-GGMN05 performs quite faster than the majority. WT-RRR02 and WT-RRR02v2 are small and fast indexes, specially for sequences with small σ. A similar performance is found in the last column, Skewed 0.99.

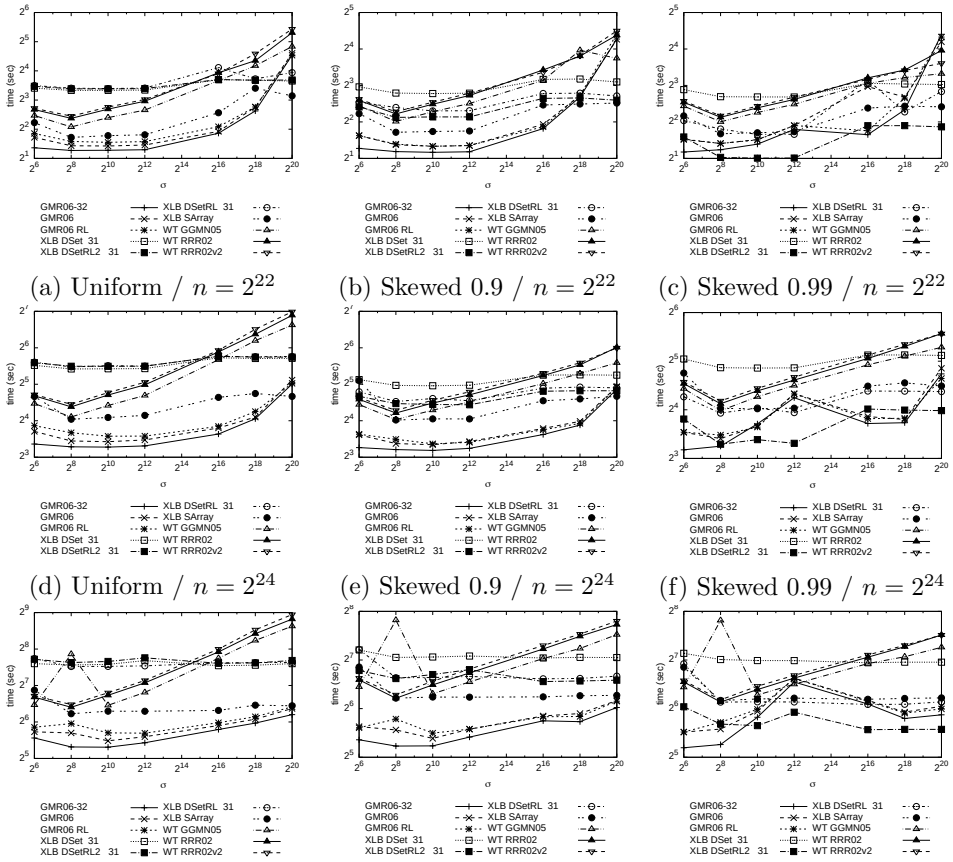


(a) Uniform / $\sigma = 2^{16}$ (b) Skewed 0.9 / $\sigma = 2^{16}$ (c) Skewed 0.99 / $\sigma = 2^{16}$

(d) Uniform / $\sigma = 2^{18}$ (e) Skewed 0.9 / $\sigma = 2^{18}$ (f) Skewed 0.99 / $\sigma = 2^{18}$

(g) Uniform / $\sigma = 2^{20}$ (h) Skewed 0.9 / $\sigma = 2^{20}$ (i) Skewed 0.99 / $\sigma = 2^{20}$

Figure A.10: Select time on consecutive arguments

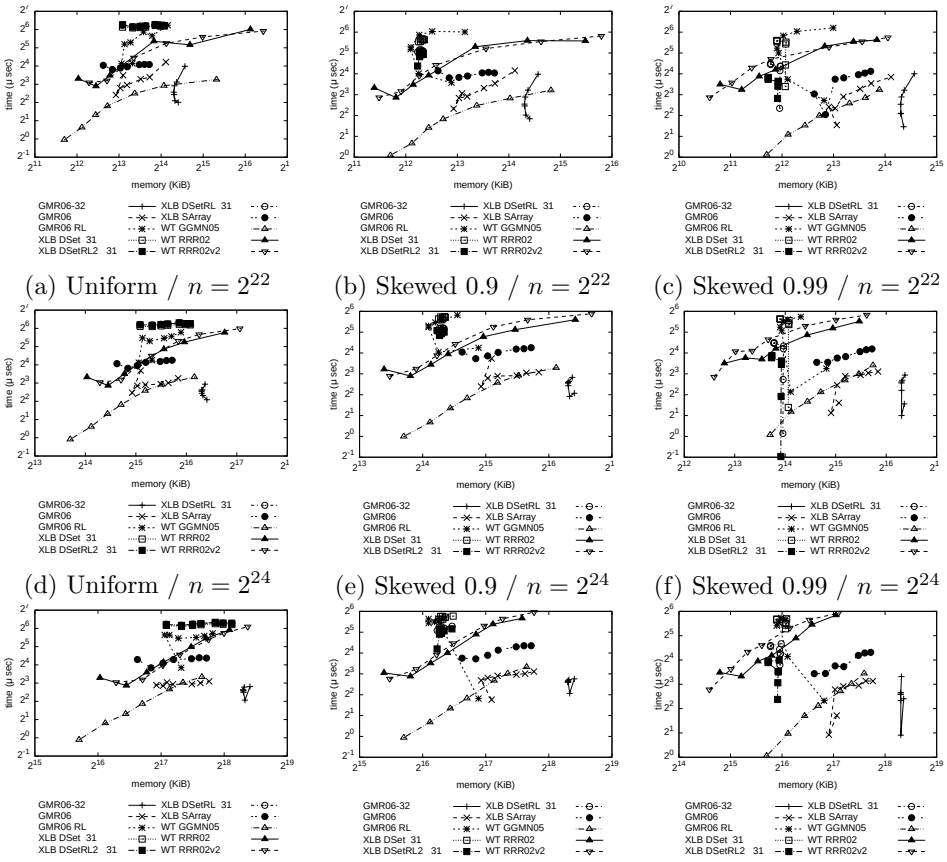


(a) Uniform / $n = 2^{22}$ (b) Skewed 0.9 / $n = 2^{22}$ (c) Skewed 0.99 / $n = 2^{22}$

(d) Uniform / $n = 2^{24}$ (e) Skewed 0.9 / $n = 2^{24}$ (f) Skewed 0.99 / $n = 2^{24}$

(g) Uniform / $n = 2^{26}$ (h) Skewed 0.9 / $n = 2^{26}$ (i) Skewed 0.99 / $n = 2^{26}$

Figure A.11: Construction time on varying σ.



(a) Uniform / $n = 2^{22}$ (b) Skewed 0.9 / $n = 2^{22}$ (c) Skewed 0.99 / $n = 2^{22}$

(d) Uniform / $n = 2^{24}$ (e) Skewed 0.9 / $n = 2^{24}$ (f) Skewed 0.99 / $n = 2^{24}$

(g) Uniform / $n = 2^{26}$ (h) Skewed 0.9 / $n = 2^{26}$ (i) Skewed 0.99 / $n = 2^{26}$

Figure A.12: Access time on varying σ.

On most configurations, GMR06 methods and XLB-SArray are showing a small time dependency on σ, enforcing our previous conclusions.

Performance of Rank

Rank's performance is shown in Figure A.13. For small alphabets we observed a behavior similar to Access, i.e., excellent times for WT-GGMN05 and smaller indexes for WT-RRR02 and WT-RRR02v2. Nevertheless, as σ grows, XLB and GMR06 indexes improve its performance. Also, on sequences with many runs like Skewed 0.9 and Skewed 0.99, our XLB indexes with DiffSet (and its run-length variants) are faster and smaller for large alphabets, and very robust to σ size.

Performance of Select

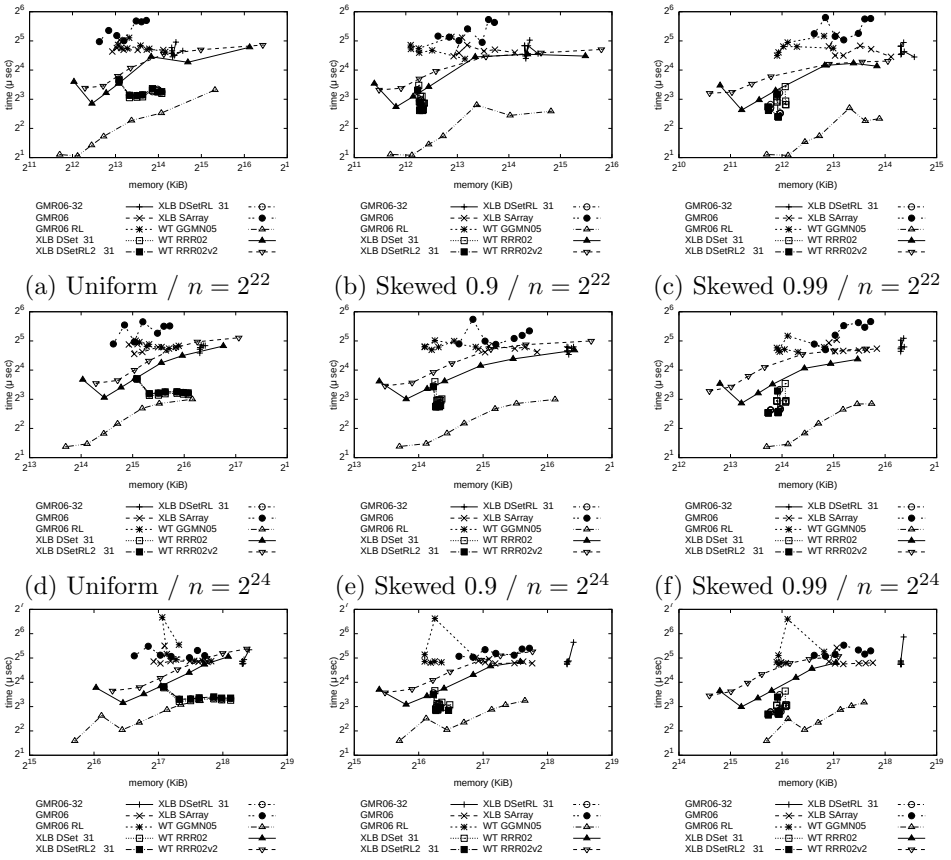
Figure A.14 shows the performance of Select on varying σ. Here, the faster indexes are the XLB DiffSet-family of indexes. For small σ, the least memory cost family is for the compressed WT (RRR02 and RRR02v2). The memory cost on larger σ are clearly dominated by XLB methods. The behavior of GMR06 and its variants is quite good for the majority of the setups.

Performance of Select with Consecutive Arguments

Figure A.15 shows the performance of Select with consecutive arguments on varying σ. Here, the time of XLB methods is several times smaller than other alternatives. As before, for small σ, the smaller indexes are those produced by WT with compressed bitmaps. Yet this is rapidly surpassed by XLB methods as σ grows.

A.5 Perspectives

Our XLB indexes are efficient alternatives to traditional sequences. They perform pretty good on very large alphabets, with excellent performance in sequences with many runs. In particular, our XLB provides efficient Access operations. In general, provides excellent tradeoffs on memory and time for Rank, Access, and Select operations. Also, our index is unbeatable on Select operations in both XLB-SArray for non-compressible sequences; and XLB-DiffSet(*) for compressed sequences. So, they have a promising application domain on text information retrieval, as a replacement of inverted indexes and the text itself.

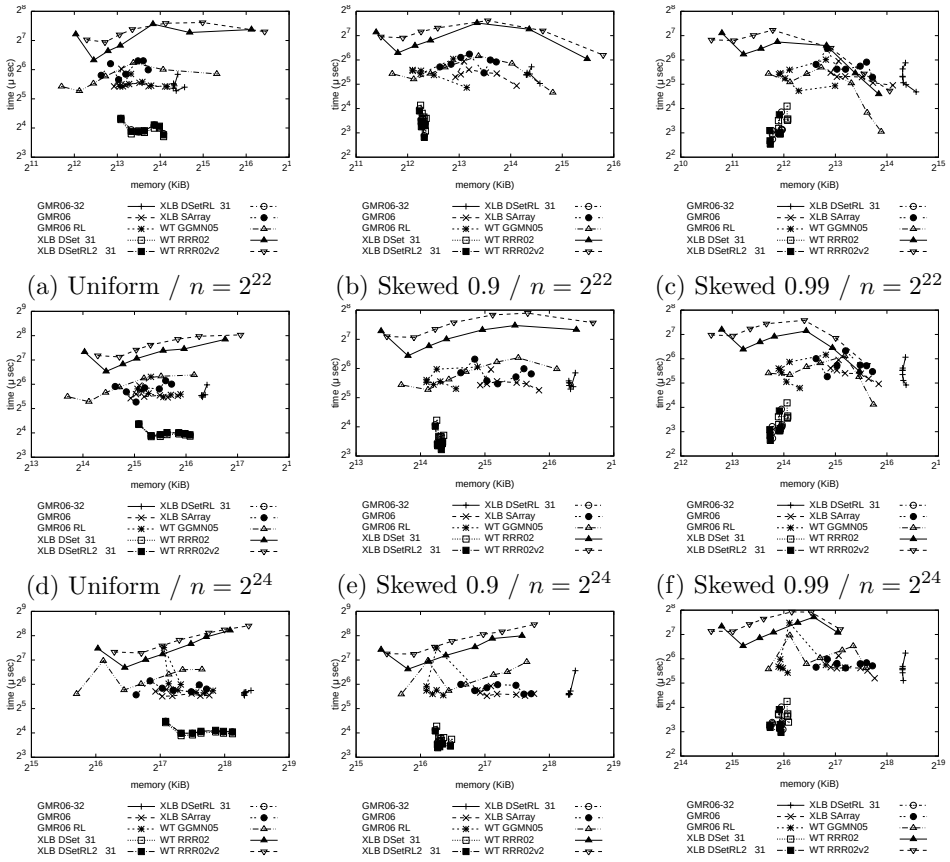


(a) Uniform / $n = 2^{22}$ (b) Skewed 0.9 / $n = 2^{22}$ (c) Skewed 0.99 / $n = 2^{22}$

(d) Uniform / $n = 2^{24}$ (e) Skewed 0.9 / $n = 2^{24}$ (f) Skewed 0.99 / $n = 2^{24}$

(g) Uniform / $n = 2^{26}$ (h) Skewed 0.9 / $n = 2^{26}$ (i) Skewed 0.99 / $n = 2^{26}$

Figure A.13: Rank time on varying σ.

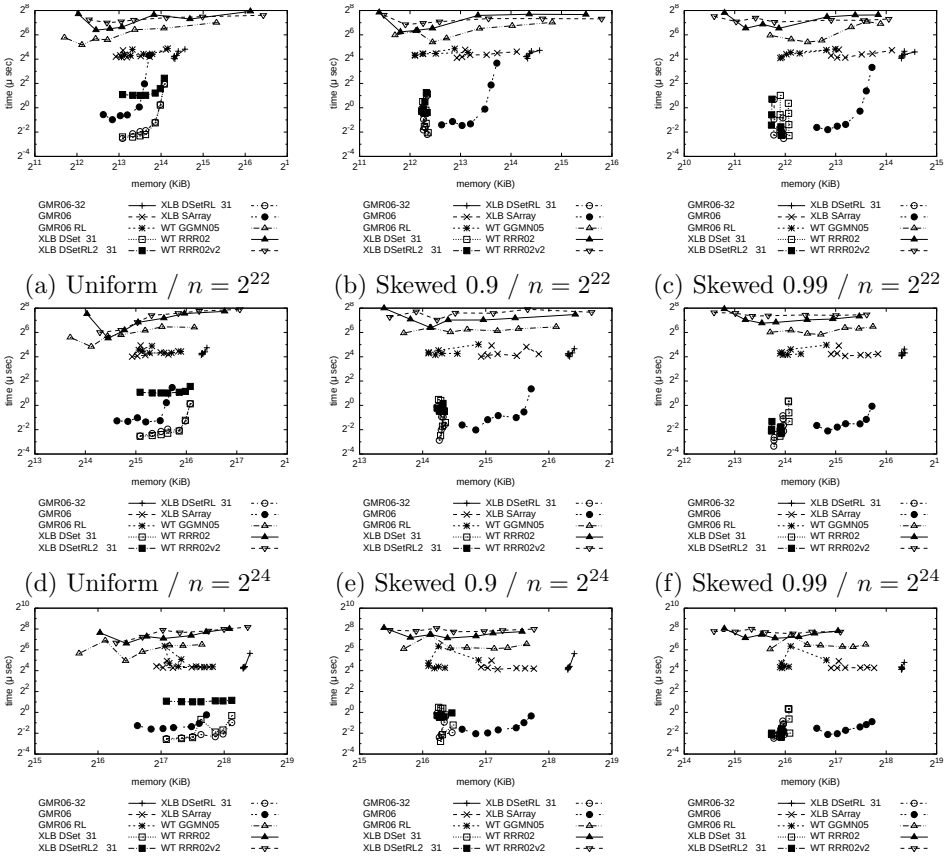


(a) Uniform / $n = 2^{22}$ (b) Skewed 0.9 / $n = 2^{22}$ (c) Skewed 0.99 / $n = 2^{22}$

(d) Uniform / $n = 2^{24}$ (e) Skewed 0.9 / $n = 2^{24}$ (f) Skewed 0.99 / $n = 2^{24}$

(g) Uniform / $n = 2^{26}$ (h) Skewed 0.9 / $n = 2^{26}$ (i) Skewed 0.99 / $n = 2^{26}$

Figure A.14: Select time on varying σ.



(a) Uniform / $n = 2^{22}$ (b) Skewed 0.9 / $n = 2^{22}$ (c) Skewed 0.99 / $n = 2^{22}$

(d) Uniform / $n = 2^{24}$ (e) Skewed 0.9 / $n = 2^{24}$ (f) Skewed 0.99 / $n = 2^{24}$

(g) Uniform / $n = 2^{26}$ (h) Skewed 0.9 / $n = 2^{26}$ (i) Skewed 0.99 / $n = 2^{26}$

Figure A.15: Select time with consecutive arguments on varying σ.

Due to its good performance on sequences with many/large runs, our XLB-DiffSet(*) indexes can be of use to FM-Indexes [Navarro and Mäkinen, 2007], the target is to index T^{BWT} when σ is very large.

Remarkably, our XLB family hold the best performance on large alphabets and large n for the majority of operations, specially on Select with random and consecutive arguments. Further, our index becomes the faster compressed index for Rank. On the other hand, Access is our slower operation, still exposing good times when compared against compressed indexes. Also, on large σ values our indexes with a high number of runs, our run-length based techniques (XLB and GMR06-RL) produce the smaller set of indexes.

We introduce several alternatives for well known bitmap and sequences indexes, as listed below.

— **RRR02v2**. A bitmap taking advantage of small local entropy values, even more than RRR02. The idea is to reduce the class's identifiers for classes with a few number of blocks. The WT with RRR02v2 improves the performance for small alphabets and large number of runs.

— **GMR06-RL**. This sequence takes advantage of the number of runs, since it avoids the storage of full permutations. When a run is found, it is marked in a special bitmap and the header of the run is stored. Again, we can observe in figures of Skewed 0.99, a dramatic reduction of the memory requirements.

Finally, we add efficient support to Access on Unraveled Sequences, using a simple variant of our technique. Thus, sequences with small to medium sized alphabets can be improved too. So, with unraveled sequences a whole set of bitmap's mixtures are possible with our approach. For example, dense bitmaps should be indexed either with fast uncompressed bitmaps or with bitmaps taking advantage of local entropy, if this property is detected. On the other hand, sparse rows can be indexed with efficient compressed bitmaps. Also, the speed of operations can be optimized, for example frequently retrieved symbols can be *promoted* to faster bitmap implementations, even if they suppose a memory waste.

Bibliography

Amato, G. and Savino, P. (2008). Approximate similarity search in metric spaces using inverted files. In *InfoScale '08: Proceedings of the 3rd international conference on Scalable information systems*, pages 1–10, ICST, Brussels, Belgium, Belgium. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

Andoni, A. and Indyk, P. (2008). Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Communications ACM*, 51:117–122.

Baeza-Yates, R. and Navarro, G. (1998). Fast approximate string matching in a dictionary. In *Proc. 5th International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 14–22. IEEE CS Press.

Baeza-Yates, R. and Salinger, A. (2010). Fast intersection algorithms for sorted sequences. In Elomaa, T., Mannila, H., and Orponen, P., editors, *Algorithms and Applications*, volume 6060 of *Lecture Notes in Computer Science*, pages 45–61. Springer Berlin / Heidelberg.

Baeza-Yates, R. A. and Ribeiro-Neto, B. A. (1999). *Modern Information Retrieval*. ACM Press / Addison-Wesley.

Barbay, J. and Kenyon, C. (2002). Adaptive intersection and t-threshold problems. In *Proceedings of the 13th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 390–399. ACM-SIAM, ACM.

Barbay, J., López-Ortiz, A., Lu, T., and Salinger, A. (2009). An experimental investigation of set intersection algorithms for text searching. *J. Exp. Algorithmics*, 14:3.7–3.24.

Böhm, C., Berchtold, S., and Keim, D. A. (2001). Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. *ACM Computing Surveys*, 33(3):322–373.

Bolettieri, P., Esuli, A., Falchi, F., Lucchese, C., Perego, R., Piccioli, T., and Rabitti, F. (2009). Cophir: a test collection for content-based image retrieval. *CoRR*, abs/0905.4627v2.

Brin, S. (1995). Near neighbor search in large metric spaces. In *Proceedings of the 21th International Conference on Very Large Data Bases*, VLDB '95, pages 574–584, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Burkhard, W. and Keller, R. (1973). Some approaches to best-match file searching. *Communications of the ACM*, 16(4):230–236.

Bustos, B. and Navarro, G. (2004). Probabilistic proximity search algorithms based on compact partitions. *Journal of Discrete Algorithms*, 2(1):115–134.

Chávez, E. and Figueroa, K. (2004). Faster proximity searching in metric data. In *MICAI*, pages 222–231.

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

Chávez, E., Marroquin, J., and Navarro, G. (2001). Fixed queries array: A fast and economical data structure for proximity searching. *Multimedia Tools and Applications (MTAP)*, 14(2):113–135.

Chávez, E. and Navarro, G. (2003). Probabilistic proximity search: Fighting the curse of dimensionality in metric spaces. *Information Processing Letters*, 85:39–46.

Chávez, E. and Navarro, G. (2005). A compact space decomposition for effective metric indexing. *Pattern Recogn. Lett.*, 26:1363–1376.

Chávez, E., Navarro, G., Baeza-Yates, R., and Marroquín, J. L. (2001). Searching in metric spaces. *ACM Comput. Surv.*, 33(3):273–321.

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

Clark, D. R. (1996). *Compact pat trees*. PhD thesis, University of Waterloo, Waterloo, Ontario, Canada.

Claude, F. and Navarro, G. (2008). Practical rank/select queries over arbitrary sequences. In *Proc. 15th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 5280, pages 176–187. Springer.

Cormen, T. H., Leiserson, C., Rivest, R. L., and Stein, C. E. L. C. (2001). *Introduction to Algorithms*. McGraw-Hill, Inc., New York, NY, USA, second edition edition.

Culpepper, J. S. and Moffat, A. (2010). Efficient set intersection for inverted indexing. *ACM Trans. Inf. Syst.*, 29:1:1–1:25.

Elias, P. (1975). Universal codeword sets and representations of the integers. *Information Theory, IEEE Transactions on*, 21(2):194 – 203.

Esuli, A. (2009). Pp-index: Using permutation prefixes for efficient and scalable approximate similarity search. In *Proceedings of the 7th Workshop on Large-Scale Distributed Systems for Information Retrieval (LSDS-IR'09)*, pages 17–24, Boston, USA.

Figueroa, K., Chavez, E., and Navarro, G. (2009). Technical description of the metric spaces library. Technical Report. SISAP Project.

Figueroa, K. and Frediksson, K. (2009). Speeding up permutation based indexing with indexing. In *Second International Workshop on Similarity Search and Applications*, pages 107–114. IEEE Computer Society.

Gionis, A., Indyk, P., and Motwani, R. (1999). Similarity search in high dimensions via hashing. In *Proceedings of the 25th International Conference on Very Large Data Bases*, VLDB '99, pages 518–529, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Golynski, A., Munro, J. I., and Rao, S. S. (2006). Rank/select operations on large alphabets: a tool for text indexing. In *Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm*, SODA '06, pages 368–373, New York, NY, USA. ACM.

González, R., Grabowski, S., Mäkinen, V., and Navarro, G. (2005). Practical implementation of rank and select queries. In *Poster Proc. Volume of 4th Workshop on Efficient and Experimental Algorithms (WEA)*, pages 27–38, Greece. CTI Press and Ellinika Grammata.

Grossi, R., Gupta, A., and Vitter, J. S. (2003). High-order entropy-compressed text indexes. In *Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*, SODA '03, pages 841–850, Philadelphia, PA, USA. Society for Industrial and Applied Mathematics.

Grossman, D. A. and Frieder, O. (2004). *Information Retrieval: Algorithms and Heuristics*. Springer.

Hjaltason, G. R. and Samet, H. (2003). Index-driven similarity search in metric spaces (survey article). *ACM Trans. Database Syst.*, 28(4):517–580.

Ibarrola, A. C. and Chávez, E. (2006). A robust entropy-based audio-fingerprint. IEEE.

Indyk, P. (2004). *Handbook of Discrete and Computational Geometry, J.E. Goodman and J. O'Rouke, editors.* CRC press, Boca Raton, FL., 2 edition.

Jacobson, G. (1989). Space-efficient static trees and graphs. In *30th Annual Symposium on Foundations of Computer Science, 1989*, pages 549–554. IEEE.

Kalantari, I. and McDonald, G. (1983). A data structure and an algorithm for the nearest point problem. *IEEE Transactions on Software Engineering*, 9(5):631–634.

Knuth, D. E. (1998). *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2nd ed edition.

Kyselak, M., Novak, D., and Zezula, P. (2011). Stabilizing the recall in similarity search. In *Proceedings of the Fourth International Conference on SImilarity Search and APplications*, SISAP '11, pages 43–49, New York, NY, USA. ACM.

Micó, M. L., Oncina, J., and Vidal, E. (1994). 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.

Munro, J. I., Raman, R., Raman, V., and Rao, S. S. (2003). Succinct representations of permutations. In *Proceedings of the 30th international conference on Automata, languages and programming*, ICALP'03, pages 345–356, Berlin, Heidelberg. Springer-Verlag.

Navarro, G. (2001). A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31–88.

Navarro, G. (2009). Analyzing metric space indexes: What for? In *Proc. 2nd International Workshop on Similarity Search and Applications (SISAP)*, pages 3–10. IEEE CS Press. Invited paper.

Navarro, G. and Mäkinen, V. (2007). Compressed full-text indexes. *ACM Computing Surveys*, 39(1).

Okanohara, D. and Sadakane, K. (2007). Practical entropy-compressed rank/select dictionary. In *Proceedings of the Workshop on Algorithm Engineering and Experiments, ALENEX 2007*, New Orleans, Louisiana, USA. SIAM.

Pestov, V. (2007). Intrinsic dimension of a dataset: what properties does one expect? In *Proc. 20th Int. Joint Conf. on Neural Networks, Orlando, FL, 2007*, pages 1775–1780.

Pestov, V. (2008). An axiomatic approach to intrinsic dimension of a dataset. *Neural Networks*, 21(2-3):204–213.

Pestov, V. (2010a). Indexability, concentration, and vc theory. In *Proceedings of the Third International Conference on SImilarity Search and APplications*, SISAP '10, pages 3–12, New York, NY, USA. ACM.

Pestov, V. (2010b). Intrinsic dimensionality. *SIGSPATIAL Special*, 2:8–11.

Puglisi, S. J., Smyth, W. F., and Turpin, A. H. (2007). A taxonomy of suffix array construction algorithms. *ACM Comput. Surv.*, 39.

Raman, R., Raman, V., and Rao, S. S. (2002). Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 233–242, San Francisco, CA, USA. ACM/SIAM.

Samet, H. (2006). *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, 1 edition.

Shaft, U. and Ramakrishnan, R. (2006). Theory of nearest neighbors indexability. *ACM Trans. Database Syst.*, 31:814–838.

Skopal, T. (2007). Unified framework for fast exact and approximate search in dissimilarity spaces. *ACM Transactions on Database Systems*, 32(4).

Skopal, T. (2010). Where are you heading, metric access methods?: a provocative survey. In *Proceedings of the Third International Conference on SImilarity Search and APplications*, SISAP '10, pages 13–21, New York, NY, USA. ACM.

Skopal, T. and Bustos, B. (2011). On nonmetric similarity search problems in complex domains. *ACM Computing Surveys*, 43(4):34:1–34:50.

Tellez, E. S. and Chavez, E. (2010). On locality sensitive hashing in metric spaces. In *Proceedings of the Third International Conference on SImilarity Search and APplications*, SISAP 2010, pages 67–74, New York, NY, USA. ACM.

Tellez, E. S. and Chavez, E. (2012). Revisiting the list of clusters. In *4th Mexican Congress on Pattern Recognition, MCPR 2012*. Springer Verlag, Lecture Notes in Computer Science.

Tellez, E. S., Chavez, E., and Camarena-Ibarrola, A. (2009). A brief index for proximity searching. In *Proceedings of 14th Iberoamerican Congress on Pattern Recognition CIARP 2009*, Lecture Notes in Computer Science, pages 529–536, Berlin, Heidelberg. Springer Verlag.

Tellez, E. S., Chávez, E., and Figueroa, K. (2012). Polyphasic metric indexes: Throught practical limits of proximity search on metric spaces. In *Proc. 5th International Conference on Similarity Search and Applications, SISAP 2012*. ACM Press.

Tellez, E. S., Chavez, E., and Graff, M. (2011a). Scalable pattern search analysis. In *3rd Mexican Congress on Pattern Recognition, MCPR 2011*. Springer Verlag, Lecture Notes in Computer Science.

Tellez, E. S., Chávez, E., and Navarro, G. (2011b). Succinct nearest neighbor search. In *Proc. 4th International Conference on Similarity Search and Applications, SISAP 2011*. ACM Press.

Uhlmann, J. K. (1991). Satisfying general proximity/similarity queries with metric trees. *Information Processing Letters*, 40(4):175–179.

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

Volnyansky, I. and Pestov, V. (2009). Curse of dimensionality in pivot based indexes. In *Proceedings of the 2009 Second International Workshop on Similarity Search and Applications*, SISAP '09, pages 39–46, Washington, DC, USA. IEEE Computer Society.

Witten, I. H., Moffat, A., and Bell, T. C. (1999). *Managing Gigabytes: Compressing and Indexing documents and images*. Morgan Kaufmann Publishing, second edition edition.

Zezula, P., Amato, G., Dohnal, V., and Batko, M. (2006). *Similarity Search - The Metric Space Approach*, volume 32. Springer, Series: Advances in Database Systems, 1st edition edition.