André Belchior Mourão

Mestre em Engenharia Informática

Towards an Architecture for Efficient Distributed Search of Multimodal Information

Dissertação para obtenção do Grau de Doutor em

Informática

Orientador: João Miguel da Costa Magalhães, Prof. Auxíliar, Universidade Nova de Lisboa

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Janeiro, 2018

Towards an Architecture for Efficient Distributed Search of Multimodal Information

Este documento foi gerado utilizando o processador (pdf)LaTeX, com base no template "unlthesis" [1] desenvolvido no Dep. Informática da FCT-NOVA [2]. [1] `https://github.com/joaomlourenco/unlthesis` [2] `http://www.di.fct.unl.pt`

Acknowledgements

I'll start this thesis by expressing my gratitude to my advisor, João Magalhães, for his endless time, continuous support and constructive criticism. His guidance kept me motivated and helped me persevere throughout this long journey.

I'd like to thank my thesis advisory committee members, Francisco Couto and Sérgio Duarte, for raising important questions that helped shape this thesis and to Laurent Amsaleg and Matthijs Douze for the making the viva thesis defence both a challenging and enjoying experience.

I'd also like to thank Fundação para a Ciência e Tecnologia, for funding my research through scholarship SFRH/BD/95064/2013 and Microsoft Research for providing two Microsoft Azure Research Award sponsorships that enabled the evaluation of this thesis work in a real-world large scale distributed setting.

I'm grateful for the support and encouragement I've received from my friends and family. In particular, my friends and colleagues at the Nova LINCS P3/13 Lab for their companionship over the last six years: Flávio Martins, David Semedo, Gustavo Gonçalves, Carla Viegas, Pedro Santos, Rui Madeira, Inês Rodolfo, Serhiy Moskovchuk, Ana Figueiras, Filipa Peleja, Camila Wohlmuth, Carla Nave, Ivo Anjos, Gonçalo Marcelino, Vanessa Lopes, Daniel Gomes and Gonçalo Araújo.

My longtime friends, Rui Assunção, David Santos, Tiago Luís, Pedro Freitas, Marta Martinho, João Rato, Adriana Formiga, Sara Viera, Nuno Barreto, António Matos, Ricardo Marques, Pedro Severino and Eduardo Costa, for all the fun moments we have shared.

To my family, especially my mother, Ana Belchior, my father, Paulo Mourão, my brother Miguel Mourão and my grandmother Maria Helena Belchior, for their love and support.

Abstract

The creation of very large-scale multimedia search engines, with more than one billion images and videos, is a pressing need of digital societies where data is generated by multiple connected devices. Distributing search indexes in cloud environments is the inevitable solution to deal with the increasing scale of image and video collections. The distribution of such indexes in this setting raises multiple challenges such as the even partitioning of data space, load balancing across index nodes and the fusion of the results computed over multiple nodes. The main question behind this thesis is *how to reduce and distribute the multimedia retrieval computational complexity?*

This thesis studies the extension of sparse hash inverted indexing to distributed settings. The main goal is to ensure that indexes are uniformly distributed across computing nodes while keeping similar documents on the same nodes. Load balancing is performed at both node and index level, to guarantee that the retrieval process is not delayed by nodes that have to inspect larger subsets of the index.

Multimodal search requires the combination of the search results from individual modalities and document features. This thesis studies rank fusion techniques focused on reducing complexity by automatically selecting only the features that improve retrieval effectiveness.

The achievements of this thesis span both distributed indexing and rank fusion research. Experiments across multiple datasets show that sparse hashes can be used to distribute documents and queries across index entries in a balanced and redundant manner across nodes. Rank fusion results show that is possible to reduce retrieval complexity and improve efficiency by searching only a subset of the feature indexes.

Keywords: Multimedia retrieval; distributed indexing; rank fusion; sparse hashing

Resumo

A criação de motores de pesquisa para pesquisa em multimédia de larga escala (e.g., mais de mil milhões de documentos) é vital em sociedades digitais onde dados são gerados por biliões de dispositivos ligados em rede. A distribuição de índices de pesquisa para ambientes *cloud* é a solução inevitável para lidar com o aumento da escala das coleções de imagens e vídeos. A distribuição desses índices levanta múltiplos desafios como o balanceamento equilibrado do espaço dos dados, balanceamento de carga entre servidores e a fusão dos resultados de pesquisa calculados em múltiplos servidores. A questão principal desta tese é *como reduzir e distribuir a complexidade computacional da pesquisa de dados multimédia?*

Esta tese estuda a extensão de técnicas de indexação invertida baseadas em reconstrução esparsa para ambientes distributivos. O objetivo principal é garantir que os índices são distribuídos uniformemente pelos servidores, mantendo documentos semelhantes nos mesmos servidores. O balanceamento de carga é realizado ao nível do servidor e do índice de pesquisa, de forma a que servidores com mais documentos não atrasem o processo de pesquisa.

A pesquisa de documentos multimodais é baseada na combinação dos resultados de pesquisa das diferentes modalidades de informação. Esta tese investiga técnicas de fusão de listas de resultados, com um foco especial na redução da complexidade computacional através da seleção apenas listas que melhoram os resultados de pesquisa.

Esta tese propõe contribuições nas áreas de indexação distribuída e fusão de listas de resultados. Resultados experimentais em múltiplos conjuntos de dados mostram que técnicas de reconstrução esparsa podem ser utilizadas para a distribuição balanceada e redundante de documentos e pesquisas. A avaliação de técnicas de fusão de listas de resultados mostra é possível reduzir a complexidade de pesquisa e aumentar a precisão do sistema, através da utilização de um pequeno sub-conjunto das listas produzidas.

Palavras-chave: Pesquisa multimedia; indexação distribuída; fusão de listas de resultados; reconstruçao esparsa

Contents

List of Figures

List of Tables

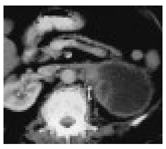
Acronyms

approximate *k*-NN	approximate *k*-Nearest Neighbour search.
B-KSVD	Balanced K-Singular Value Decomposition.
CBIR	content based image retrieval.
CDS	clinical decision support.
DISH	Distributed Indexing by Sparse Hashing.
HoG	Histogram of oriented Gradients.
IR	Information Retrieval.
ISR	Inverse Square Ranking.
k-NN	*k*-Nearest Neighbour.
KSVD	K-Singular Value Decomposition.
L2F	Learning to Fuse.
LETOR	Learning to Rank.
LSH	Locality Sensitive Hashing.
MVP	Multimedia Vertical Partitioning.
OMP	Orthogonal Matching Pursuit.
PRF	pseudo relevance feedback.
RR	Reciprocal Rank.
RRF	Reciprocal Rank Fusion.
SHI	Sparse Hash Index.
SIFT	Scale-Invariant Feature Transform.

CHAPTER 1

Introduction

Multimedia documents, made of a combination of image, video, audio, and text, are taking an ever-growing share of the data we produce. Social-media users upload hundreds of hours of video to YouTube every minute[1]. Hospitals and clinics produce hundreds of gigabytes of medical imaging exams daily [39]. Current search engines search these multimodal documents using textual queries, and some are taking the first steps towards new types of queries. Google Images and TinEye allow for searching images by example: they can retrieve similar images to a query image. The biomedical domain is a prime example of the usefulness of similarity-based multimodal search. Diagnostic Imaging exams (e.g., X-rays) are composed of one or more images representing the examined area and a textual report describing the exam and patient details. Figure 1.1 shows an example of a patient case query, where text and images are complementary: the report explicitly mentions of image regions ("mass on the upper corner") and the image provides a visual model that is difficult to put into words. Making these documents easily discoverable can help medical doctors in making clinical decisions on a daily basis [88].



A 43-year-old man with painless, gross hematuria. Abdominal CT scan revealed a large left renal mass with extension into the left renal pelvis and ureter.

Figure 1.1: Example of a "synthetic" biomedical multimodal query, created for the ImageCLEF Medical 2013 retrieval competition. It contains an image (CT scan) and a textual case description.

[1]https://www.youtube.com/yt/press/statistics.html

1.1 Towards an architecture for effective multimodal search

The goal of the search engine user is to find relevant information about an information need. *Queries* are representations of such information needs and *documents* are the unit of information of the results returned. A *collection* is a set of related documents, meant to be searched together. Documents have evolved from text to multimedia content such as images, audio, and video or interactive and mutable content [60]. *Multimodal* documents and queries are composed of a combination of content from multiple modalities (e.g., text, images, audio and video). Examples of multimedia documents include Web pages, pictures, videos, songs, scientific articles, among many others.

A search engine should return a set of documents that users consider *relevant* to answer the queries. Relevant documents are likely to share a degree of *similarity* with the query. For example, having the same terms in a query and a document is an indicator of similarity and potential relevance. However, similar images may represent the same object while having large differences in their pixel values; Sentences may have the same words but in a different order. Thus, it is not feasible to search documents in their raw forms (e.g., text as a sequence of characters, image as pixels, video as frames). To measure similarity across documents, one must create representations that distill key document characteristics into *features* that can be formally quantified and compared. For images, features represent characteristics such as color, texture, shape, among others. An example of a simple feature is a color histogram, which measures the frequency of colors across an image. Image similarity can be measured by comparing the frequency of colors across images.

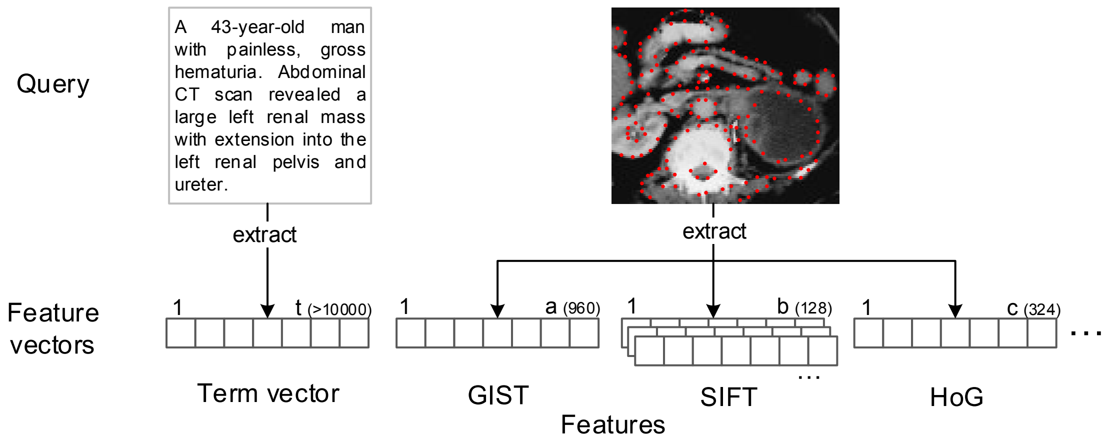

Query
A 43-year-old man with painless, gross hematuria. Abdominal CT scan revealed a large left renal mass with extension into the left renal pelvis and ureter.
extract
extract
Feature vectors
1
t (>10000)
1
a (960)
1
b (128)
1
c (324)
...
Term vector
GIST
SIFT
HoG
Features

Figure 1.2: Example feature vectors for a multimodal query (text and images) and the corresponding features.

On a search index, features are represented as real-valued feature vectors. For such vectors, the notion of *similarity* and *nearest neighbor* can be defined according to a *distance function*. A k-Nearest Neighbour (k-NN) index is a structure designed to efficiently compute a *rank list* with the k most similar feature vectors (i.e., nearest neighbors according to that distance function).

Effective multimedia similarity-based search, i.e., query-by-example, requires using multiple types of features and techniques that increase the complexity of the search process. Figures 1.2 and 1.3 show a set of feature vector for a multimodal (text and image) and a video query respectively. For the text and image query, a feasible set of features are a textual term vector and multiple image features: GIST (image texture), Scale-Invariant Feature Transform (SIFT) (local edges and colors) and Histogram of oriented Gradients (HoG) (edge orientations) for the image. Local features such as SIFT match images by detecting salient key-points on high contrast regions. Each key-point is associated with one feature vector, meaning that hundreds of feature vectors are created for each image. Global features such as GIST can be computed for multiple segments of the image, tilled and pyramid feature extraction. These techniques enable searching by region-of-interest, where a user-selected image region is compared to all other areas of the images in the index. For a video document, a feasible set of features is a set of image features extracted from its key-frames.

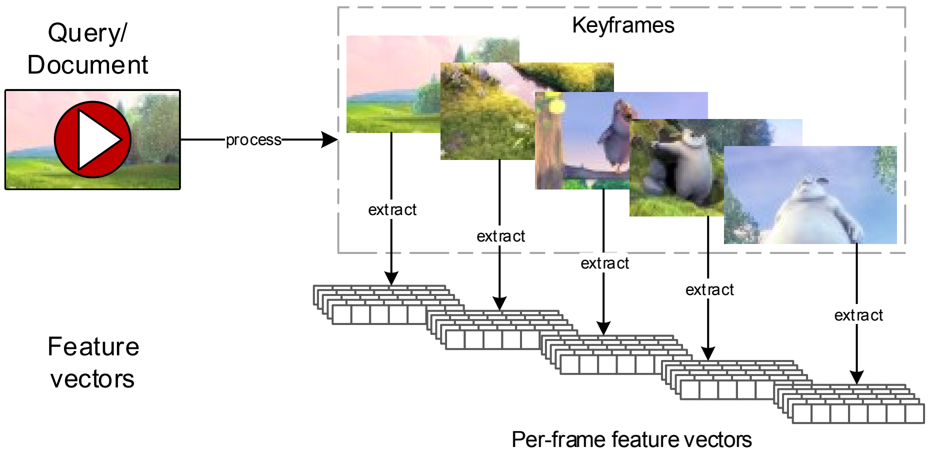


Figure 1.3: Video document with example feature vectors for each of its key-frames

The number of feature vectors extracted from a collection of multimodal documents can easily grow beyond the indexing capabilities of a single computing node. Consider a health-care professional that wants to find articles which are relevant to a patient case in the biomedical literature. The biomedical literature is composed of more than 27 million articles[2], each containing on average more than four images[3], which results in a collection of hundreds of millions of images. Key-point based features such as SIFT compare images by matching key-points extracted across high-contrast regions across images. This process leverages on finding hundreds of key-points and extracting the corresponding hundreds of feature vectors for each image. This process increases the number of vectors to index one-hundred fold, meaning that the number of vectors to index space can reach the billions of vectors for a single feature space.

[2]current size is available at `https://www.ncbi.nlm.nih.gov/pubmed/?term=all%5bsb`

[3]a sample of 70,000 articles contains 300,000 images `http://www.imageclef.org/2013/medical`

1.2 Research question and objectives

Searching on collections with over one billion vectors requires the application of index distribution techniques. Such index should advantage of the characteristics of distributed systems (e.g., cloud environments): parallel processing, hardware redundancy (i.e., enable indexing documents on more than one node), and ability to deploy additional nodes on demand. Thus the question becomes *how to reduce and distribute the multimedia search computational complexity?* In this context, I pursued three research objectives that provide partial answers to the main research question:

- **Multimodal rank fusion**: research rank fusion techniques that work on multiple modalities and features with small amounts of training data;
- **Efficient distribution of vertically partitioned indexes**: create flexible and efficient algorithms for the distribution of multi-feature indexes;
- Balanced load on **horizontally partitioned indexes**: create distribution algorithms that can distribute very large scale indexes with load balancing guarantees at node and index partition level.

Figure 1.4 shows how these objectives address the multimodal index distribution problem at multiple scales. On a multimodal indexing system, documents are indexed on multiple indexes, one index per feature (A, B, C). For small collections (i.e., 1-10 million feature vectors per index), all indexes can be stored on a single node. As collections grow larger (i.e., 10-500 million feature vectors per index), distributing indexes across nodes (i.e., vertical partitioning) becomes an effective partitioning scheme. For very large scales (500+ million feature vectors), individual feature indexes become larger than the computational resources of individual nodes, which means that individual indexes must be partitioned across nodes (i.e., horizontal partitioning).

1.2.1 Rank fusion

Multimodal queries generate multiple rank lists, one per feature type. This thesis studies both unsupervised and supervised rank fusion techniques for feature rank list fusion. The goal is to compute the combination of search results that were computed over a set of distributed feature nodes. In particular, Chapter 3 shows how to increase retrieval effectiveness (e.g., give higher scores to more relevant indexes) and efficiency (e.g., do not query indexes that have low retrieval performance) with a focus on domains with few annotated training data (e.g., biomedical literature retrieval).

1.2.2 Distributing and partitioning multimodal indexes

Distributing a search system to multiple nodes is a challenging task, with potential performance advantages, such as super-linear performance gains with better parallelization.

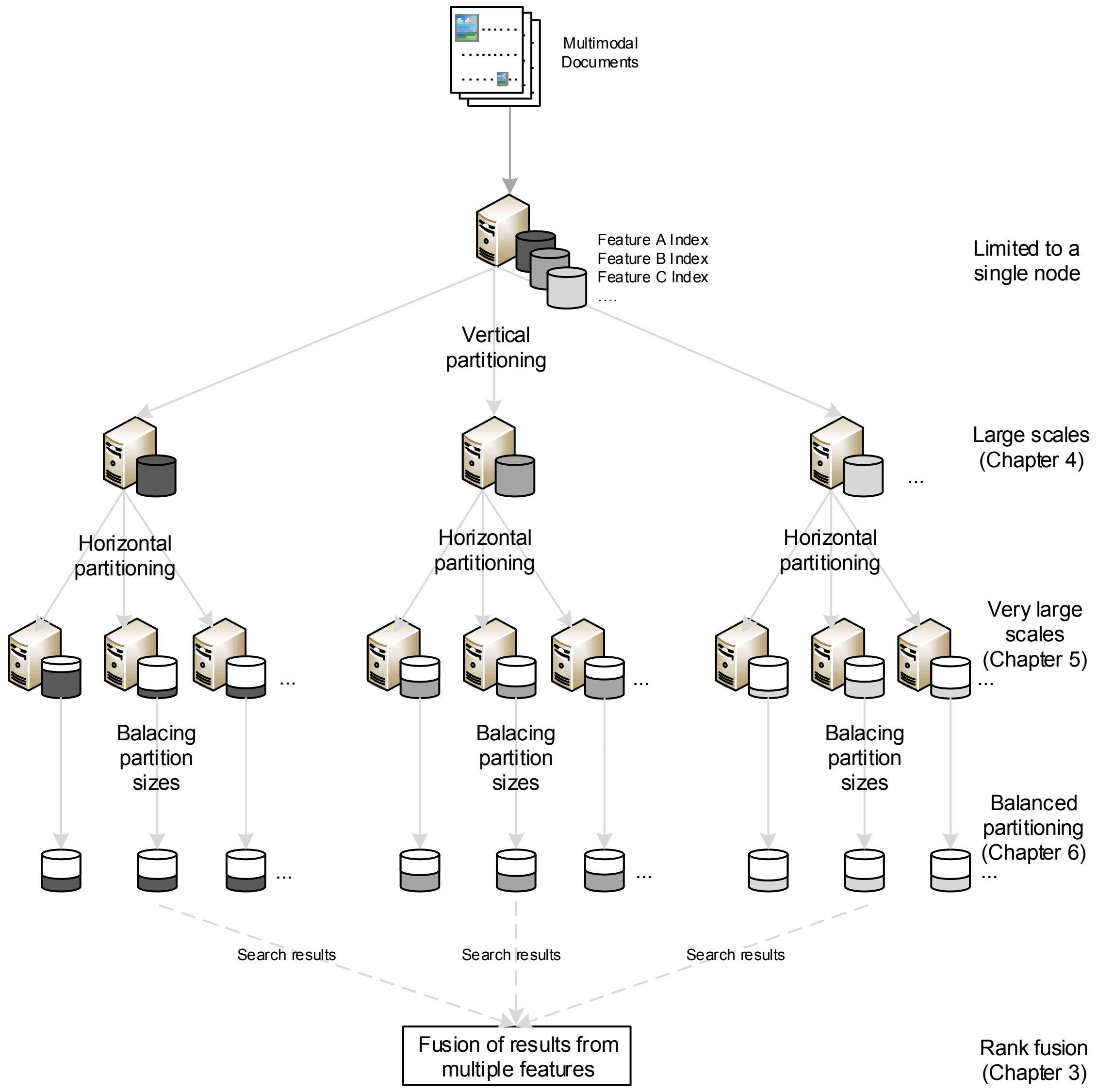


Figure 1.4: Towards an architecture of a distributed multimodal retrieval

The distribution process can greatly influence the retrieval performance: e.g., how to handle node or network failures, slow or inefficient communication protocols and unbalanced load across nodes. Existing k-Nearest Neighbour retrieval systems either deal with distribution as: (i) an extension of single node retrieval systems (e.g., distributed M-tree [14]), overlooking the requirements of distributed systems, or (ii) by employing frameworks such as Map-Reduce [33], which are not very effective on reactive systems that must be ready to answer queries in real-time, as they were designed for bulk data processing [74]. Thus, the problem becomes *how to distribute a search index across nodes effectively?*

Chapter 4 describes the Multimedia Vertical Partitioning (MVP) architecture, which deals with the distribution of documents and combination of search results over a set of indexes distributed over multiple nodes. For very large scales (500+ million feature

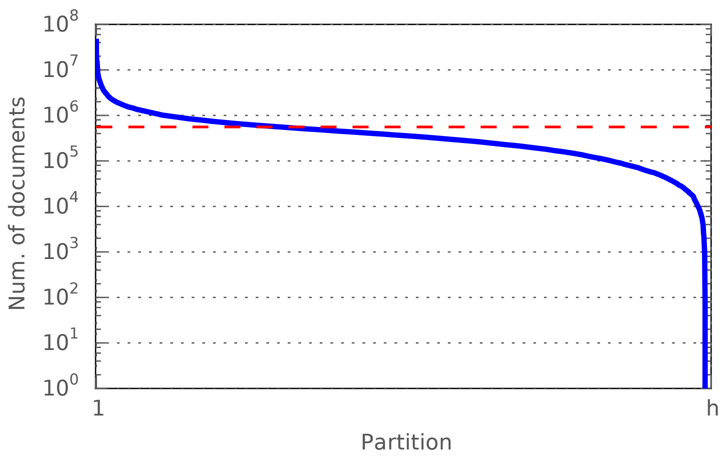


Figure 1.5: Example of an unbalanced partition distribution. The Y-axis is on a logarithmic scale. The red line represents a perfect distribution and the blue line represents the sizes of the created partitions, sorted from largest to smallest.

vectors), individual feature indexes become larger than the computational resources of individual nodes. The solution is to partition index partitions across nodes (i.e., horizontal partitioning), Figure 1.4. Chapter 5 proposes DISH, a horizontally partitioned index which distributes documents across nodes by similarity, such that documents relevant to a query are concentrated across few nodes [56].

1.2.3 Balancing index partitions

DISH indexes can partition very large indexes to multiple nodes while minimizing the differences in the number of documents per nodes. Uneven partition sizes become a problem when partitions are distributed across multiple nodes: Nodes with larger partitions take more time answering queries on scenarios where the system is answering multiple concurrent query streams. Figure 1.5 shows the distribution of documents across h partitions on a GIST feature dataset. The X-axis represents index partitions the Y-axis represents the number of documents per partition (logarithmic scale). Existing techniques generate partition size distributions (red line) that are far from an even distribution of documents across partitions (blue line). The partition size differences happen at both the largest (two orders of magnitude larger than the mean partition size) and smallest partitions (three orders of magnitude smaller than the mean partition size). Balancing partition sizes while preserving similarity may appear contradictory: if the data on the feature space is not uniformly distributed, *how can one guarantee* ***an even partitioning of space*** *in both the densely and sparsely populated regions?* Chapter 6 proposes a solution to this problem by generating similarity-based index partitions that match the distribution of documents in the original space. These techniques improve retrieval time and optimize precision by balancing the size of the partitions on each node, Figure 1.4, which results in a uniform candidate list sizes for each query on each node.

1.3 Contributions

The main contribution of this thesis is a comprehensive architecture for the distribution of multimedia indexes. It explores and evaluates key issues with vertical and horizontal index partitioning and the balancing of load across those two partitioning paradigms, as displayed on Figure 1.4.

On the multimodal index partitioning literature, existing works are either tied to the single node indexing structure [3, 79] or were not tested on scenarios with large indexes (millions of documents) [108]. Thus, this thesis proposes a generic architecture for effective vertical index partitioning. It enables multimodal search when combined with rank fusion techniques, and adds little overhead to the retrieval process. These contributions are presented on Chapter 4 and are a part of the following article:

- André Mourão and João Magalhães, *Scalable multimodal search with distributed indexing by sparse hashing*, paper on the 5th ACM on International Conference on Multimedia Retrieval (ICMR), 2015.

The previously described architecture works while the indexes for individual features fit the memory and processing capabilities of a single node. However, when the number of documents in the index goes beyond the capabilities of a single node, distributing the index to multiple nodes is the natural best step. Existing algorithms are focused either on load-balancing across nodes, without regarding intra-node document similarity or, improving similarity-base indexing algorithm, designed to work on a single node with homogeneous access to the full index. Thus, I propose a similarity-based indexing partitioning approach, DISH that can be distributed to a set of nodes of varying size while keeping the number of documents across nodes uniform: These contributions are presented on Chapter 5 and are a part of the following article:

- André Mourão and João Magalhães, *Towards Cloud Distributed Image Indexing by Sparse Hashing*, **under review** on the ACM Multimedia Conference (ACMMM), 2018.

DISH achieves load balancing by minimizing the standard deviation in the total number of documents per node. However, partitions with significantly different sizes still cause load-balancing issues: nodes will take more time inspecting larger partitions. To generate uniformly sized partitions, my approach was the creation of a sparse hash dictionary that penalizes index partitions that are assigned to a large number of documents: These contributions are presented on Chapter 6 and are a part of the following articles:

- André Mourão and João Magalhães, *Balanced Search Space Partitioning for Distributed Media Redundant Indexing*, paper on the 7th ACM on International Conference on Multimedia Retrieval (ICMR), 2017;

- André Mourão and João Magalhães, *Balancing Search Space Partitions by Sparse Coding for Distributed Redundant Media Indexing and Retrieval*, paper in the International Journal of Multimedia Information Retrieval (IJMIR), 2017.

Rank fusion contributions are focused on two main topics: improve unsupervised rank fusion algorithms and how to learn to select which rank lists for fusion on domains with limited training data. Inverse Square Ranking fusion was created to improve the combination of search results from multiple types of features and modalities: Unsupervised rank fusion algorithms fail when working on rank lists with large differences in retrieval performance. Learning to Rank (LETOR) algorithms solve this problem by selecting which rank lists to use to generate the final rank list, given large amounts of queries with manually annotated results as training data. However, LETOR algorithms can overfit training data when working on domains with limited training data. In this thesis, I propose Learning to Fuse, a supervised algorithm for the selection of rank lists for fusion on such domains. It works by selecting a limited set of rank lists for fusion that maximizes a selected retrieval metric for a small train data sample (less than 100 queries with annotated relevance judgments): These contributions are presented on Chapter 3 and are a part of the following articles:

- André Mourão, Flávio Martins and João Magalhães, *Inverse square rank fusion for multimodal search*, paper at the 12th IEEE International Workshop on Content-Based Multimedia Indexing (CBMI), 2014;

- André Mourão and João Magalhães, *Low Complexity Supervised Rank Fusion models*, **under review** to the 27th International ACM Conference on Information and Knowledge Management (CIKM), 2018.

1.4 Organization

The remainder of this document is organized into six chapters:

Chapter 2 – Background and related work: includes an overview of background work in nearest neighbor search and presents the related work in distributed indexing and rank fusion research;

Chapter 3 – Learning to combine rank lists: explores supervised and unsupervised rank fusion for the combination of results from multiple modalities. Experiments on multiple types of data such as multimodal biomedical retrieval and federated search show that rank fusion works well for various data types with different properties (e.g., high and low overlap in documents between ranks);

Chapter 4 – Multimedia vertical partitioning indexes: describes the creation of an architecture for the vertical distribution of a search index to multiple nodes. It formalizes the indexing and distribution process, providing the foundations for the multimodal search architecture, incorporating partitioning, distribution, and fusion. The evaluation section shows the impact of scale and feature type on the architecture's distribution overhead.

Chapter 5 – Sharding very large-scale multimodal indexes: shows how to partition single feature indexes that do not fit individual node, by high dimensional over-complete, sparse hashing. It proposes an algorithm to balance the total number of documents per node, for unbalanced partition sizes and its impact on redundant, over-complete indexing. Evaluation on a commercial cloud provider shows that sparse hash partitioning scheme achieves good results, with low temporal performance degradation under varying numbers of nodes, concurrent query streams and small precision loss on node failure;

Chapter 6 – Balancing distributed index partitions: builds on top of the previous chapter index partitioning scheme, by forcing index load balancing at partition level. It proposes Balanced K-Singular Value Decomposition (B-KSVD), an algorithm to design codebooks that generate data-driven, balanced partitions. The trade-off between partition size/load balancing and retrieval performance, by providing better retrieval performance with evenly sized partitions, when compared with existing partitioning schemes.

Chapter 7 – Conclusions: summarizes this thesis findings and contributions, with a critical view on the architecture limitations. It also discusses possible improvements, impact, and challenges that arose from this work.

Chapter 2

Background and related work

The goal of this thesis is the study of computational techniques that go towards the reduction and distribution of the computational complexity of a multimedia search architecture. Such goal spans multiple areas of research: high-dimensional indexing, rank fusion and distributed systems. This chapter details existing approaches to high-dimensional multimedia indexing, distribution and rank fusion.

2.1 Definitions

2.1.1 Distance functions

On this thesis, we consider features that are representable as high dimensional (hundreds of dimensions) vectors of real numbers, $x = (x_1, ..., x_d), x_i \in \mathbb{R}$. For k-NN retrieval, the similarity between feature vectors (i.e., finding the nearest neighbors in the feature space) is measured according to a distance. The lower the distance, the higher the similarity. The most common distance function for real numbered feature vectors is the Euclidean distance:

$$euclidean(p,q) = \|p-q\|_2 = \sqrt{\textstyle\sum_{i=0}^{d}(p_i - q_i)^2} \tag{2.1}$$

where p and q are two feature vectors, each with d dimensions.

For binary vectors, the Hamming distance is the most common choice. It counts the number of values in the feature vectors that are different:

$$hamming(p,q) = \|p-q\|_1 = \textstyle\sum_{i=0}^{d} p \oplus q \tag{2.2}$$

where $\oplus$ is the exclusive OR (XOR) operator. When dealing with sparse spaces, it is useful to define the l_0 quasi-norm as the number of non-zero values in the feature vector:

$$l_0 quasi-norm(p,q) = \|p-q\|_0 = \textstyle\sum_{i=0}^{d}((p_i - q_i) \neq 0) \tag{2.3}$$

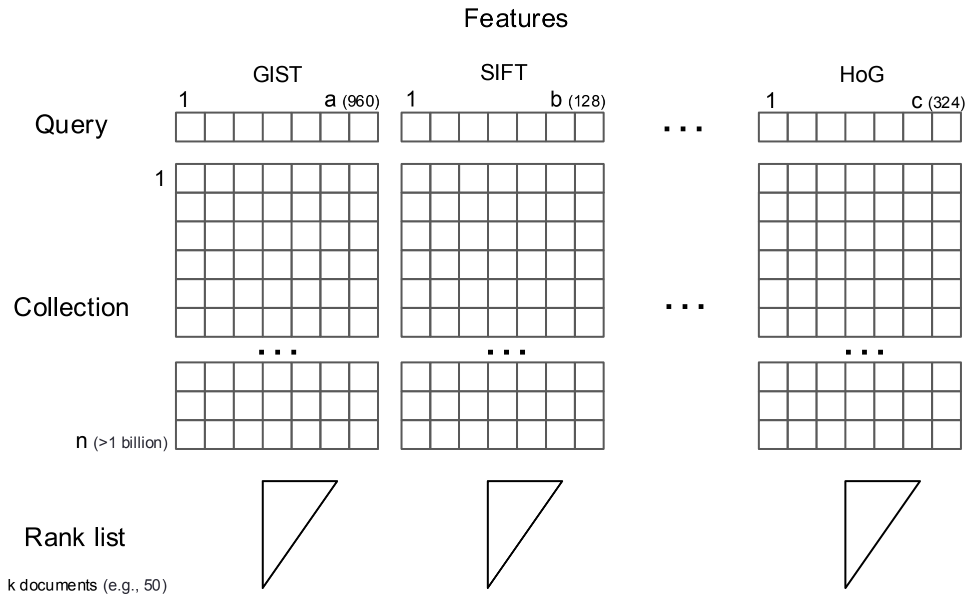


Figure 2.1: An example of a set of query feature vectors and the corresponding collection feature vectors.

2.1.2 Exact *k*-nearest neighbor

Exact k-nearest neighbor search goal is to find the exact set of k feature vectors that minimize the distance to a query feature vectors. Formally, the goal is to retrieve the $k \in \mathbb{N}^+$ nearest feature vectors $S \in \mathbb{R}^{k,d}$ to a query feature vector $q \in \mathbb{R}^d$ from an collection $Y \in \mathbb{R}^{n,d}$:

$$\forall y_i \in Y, \forall x_j \in S - Y, f(q, y_i) \leq f(q, x_j), s.t. \|s\| = k \tag{2.4}$$

where f is a distance function. An alternative to restricting retrieval to the k nearest feature vectors is to include all feature vectors that are closer to the query than a radius r:

$$\forall y_i \in Y, y_i \in S \text{ if } f(q, y_i) < r \tag{2.5}$$

Exhaustive k-NN search algorithms select candidates by comparing query feature vectors with all the collection feature vectors in the original vector space. These techniques do not scale effectively, as they rely on inspecting and ranking all the vectors against the query. This process has a complexity of $\mathcal{O}(n \times d)$, which grows linearly with both index and feature vector size. Figure 2.1 shows an example of the set of feature vectors for a query and document collection, and the per-feature rank lists containing the nearest neighbors. Multimodal, multi-feature search amplifies this problem: on Figure 2.1, the search process must be performed for all feature types, for a total complexity of $\mathcal{O}((a + b + c) \times n)$. Thus, researchers have developed alternative techniques to deliver search on large document collections, by relaxing the requirement to retrieve the exact set of nearest neighbors (i.e., approximate k-nearest neighbor).

2.1.3 Approximate *k*-nearest neighbour

Approximate k-Nearest Neighbour search (approximate k-NN) algorithms increase temporal performance by relaxing the guarantee for the retrieval of the exact k closest neighbors. Possible solutions include the reduction in the number of feature vectors to inspect per query or by the creation of alternative feature representations that can be compared more efficiently.

A radius-based approximated algorithm can be defined as finding the set of c-approximate r-near neighbors:

$$if\, \exists d(q,p_1) < r, p_1 \in Y \Rightarrow p_2 \in Y, d(q,p_2) < cr \tag{2.6}$$

for any query $q \in \mathbb{R}^d$, a multiplier c find a feature vector p_2 in an collection $Y \in \mathbb{R}^{n,d}$. An approximated version of this problem returns the feature vector p_2 with a probability of $1-\delta$, with $\delta \in [0,1]$. The δ is usually a constant value closer to one than to zero. Figure 2.2 shows how these approaches select candidates in a two-dimensional feature space. k-NN selects the k closest feature vectors to the query point, radius-based selects all the points inside the r radius sphere and c-approximate radius-based selects candidates inside the cr sphere with a probability of δ.

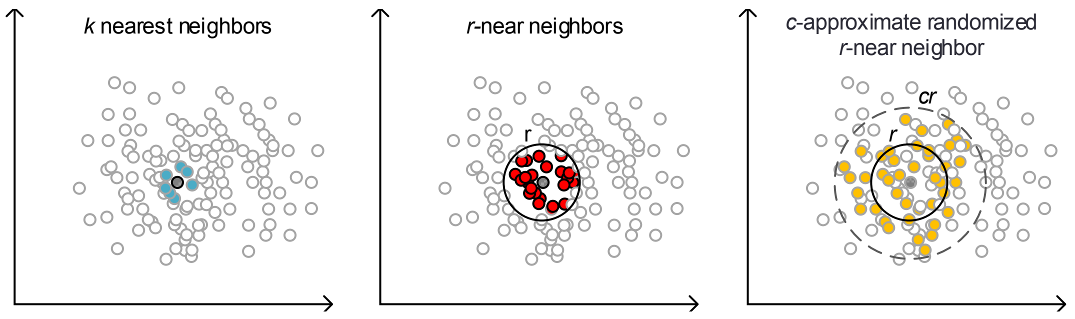


Figure 2.2: Types of k-NN search: k-nearest neighbors, radius-based and c-approximate radius-based

2.1.4 Efficiency and effectiveness

Retrieval systems are evaluated from a result quality and temporal perspective, *effectiveness* and *efficiency* respectively.

Effectiveness

Effectiveness can be defined as how well do results answer the user query. On k-NN search, the relevance of a document to a query is measured according to a set metric. The k feature vectors closer to the query (k nearest neighbors) are assumed to be relevant. However, feature vectors are an approximated representation of the documents: documents may be close to the query on a feature space and not be relevant to the user of the search engine, or vice-versa. A solution is to evaluate the quality of the results using the Cranfield model [29]: have a set of experts manually assess the relevance of results retrieved by search engines for a representative set of queries. Retrieval effectiveness can then be measured using those relevance assessments.

Precision is the ratio of retrieved relevant documents over the total number of documents retrieved. *Recall* is the ratio of retrieved relevant documents over the total number of relevant documents. Both these measures can be calculated at different levels of retrieved documents (e.g., P@k means precision at the k^{th} retrieved document). Considering C as a set of results and R as the set of relevant results (for Information Retrieval (IR) systems, all relevant documents; for k-NN systems the first k^{th} neighbours), the metrics are defined as:

$$P@k = \frac{|C_{1,\ldots,k} \cap R|}{k}, R@k = \frac{|C_{1,\ldots,k} \cap R|}{|R|} \tag{2.7}$$

These values are usually averaged (arithmetic mean) over multiple queries to get more consistent results: *average precision (avgP)*, *average recall (avgR)*.

An alternative definition of recall (*recall at R*) is used to evaluate some k-NN systems. It measures whether the first nearest neighbor is ranked in the top R positions of the returned results. As with *avgP* and *avgR* it is averaged over multiple queries. This metric is useful for matching points in 3-d spaces and other use cases where only the single closest nearest neighbor is necessary.

Efficiency

Efficiency can be defined as how long does the system take to answer user queries and what computational resources were used. Distributed systems research is focused on measuring architectural-level performance such as query throughput, load distribution quality, performance over growing number of requests, and resistance to nodes and network failure. This section defines some temporal and system metrics:

- **Query time** *(milliseconds)*: the time it takes to make a single query, from the moment the user submits a query to the moment results are returned. This metric is usually averaged over multiple queries.

- **Throughput** *(queries per second)*: rate at which requests can be processed. This metric measures how the system handles processing queries in parallel. It can be either estimated or measured using at peak or average rate on a system under real load.

2.2 Background: Search on high-dimensional dense spaces

k-NN search evolved from brute-force linear searches to sophisticated techniques that transform the feature vectors into new representations and reduce the percentage of the collection to inspect.

2.2.1 Tree-based methods

Tree-based algorithms have been an active area of study for decades [28, 103, 109]. These algorithms divide the search space by grouping similar feature vectors into sub-trees. At query time, efficient retrieval is achieved by pruning the non-relevant sub-trees. Trees mostly differ in their space partitioning techniques and heuristics.

The k-d tree [16] partitions the search space into two parts at every non-leaf node (h dimensional space). At each depth level l, the search space is partitioned at the $l \bmod h$ dimension, and the point selected to partition the search space will be at the median of the candidates. An example of a simple 2-d tree with the vectors projected on a 2-d space is on Figure 2.3. One can see the generated vertical (A and D) and horizontal (B, C, and E) partitions, and the tree structure that was produced.

For nearest neighbor search, the process is to move down the tree until one reaches a leaf node, keeping the current best candidate c as the node that minimizes the distance l to the query. After reaching the leaf node, the tree is transversed in reverse, checking if there could be a closer "nearest neighbor" on the unexamined side trees: it is inspected if an hyper-sphere centered on c with radius l crosses that node's hyperplane. The process is repeated until the root of the tree is reached. For k-NN search, the process is similar, but a candidate list of k current best nodes is kept. This process is effective with a low number of dimensions (≤ 100), but it degenerates into linear search efficiency for high dimensional feature vectors. This is due to the large increase in hyperplane intersections, which significantly increases the amount of sub-trees inspections.

The VP-Tree [109] divides the feature space into a set of circles, centered at vantage points and with multiple radii. For each circle, all left children of the node will be inside the circle, while all right children will be outside of the circle. The M-Tree [28] strengths are its ability to dynamically index data on a metric space that satisfy the triangle inequality. When a node grows over a set limit, it splits itself into several nodes that cover metric regions with minimum overlap and volume. It allows the implementation of multiple split policies, which can be tuned depending on application needs (e.g., the trade-off between low indexing time and higher retrieval precision).

The complexity of tree-based algorithms is handled by pruning the search space and will be in the order of $\mathcal{O}(logn \times d)$. As they do not handle dimensionality reduction or transformation, the effectiveness of search space partitioning is reduced when the dimensionality of the vectors to index increases [103].

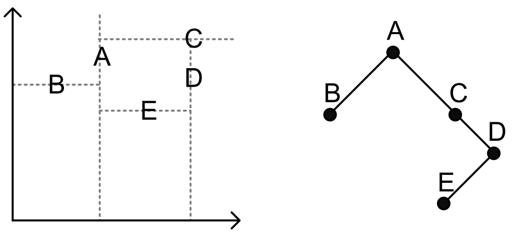


Figure 2.3: *kd*-Tree example

2.2.2 Hamming embeddings

An alternative to tree-based algorithms for k-NN search are hash-based techniques. These techniques can measure similarity very quickly in the binary space using the Hamming distance and special processor instructions. Instead of precision being limited by the number of inspected documents, these algorithms give a probabilistic guarantee that the retrieved k documents are the true nearest neighbors.

The seminal Locality Sensitive Hashing (LSH) algorithm [6] groups similar items into the same buckets (similar to hash table entries) with a high probability. At query time, only buckets with non-zero values are inspected, avoiding exhaustive linear search across all hashes. Figure 2.4 shows an example of a space with three points and the corresponding hash codes and hash table.

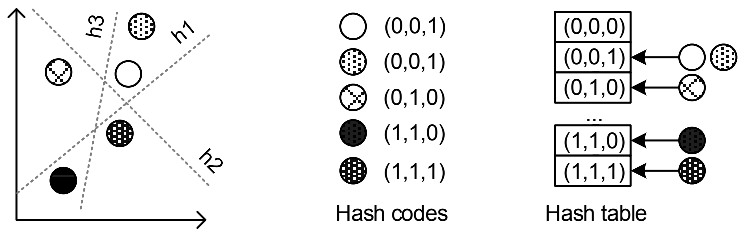


Figure 2.4: LSH space partitioning and indexing example

Consider a family F of hash functions, a threshold r and an approximation factor $c > 1$. For two points p, $q \in \mathbb{R}^h$, a family F of binary hash functions $h(p,q) = [0,1] \in F$ is interesting if:

$$P_1 = P(h(p) = h(q) \,|\, d(p,q) \leq r) \, < \, P_2 = P(h(p) = h(q) \,|\, d(p,q) \geq cr) \qquad (2.8)$$

For k-NN, LSH families can be used to create hash tables that group similar feature vectors into buckets. The algorithm has two parameters: code width h and number of tables L. G is a family of h wide hash codes, obtained by the concatenation h functions from F such that $g(p) = [f_1(p), ..., f_k(p)]$. At retrieval time, the buckets that share non-zero entries with the query are inspected, and a set of k candidates is selected until a stopping condition is met (i.e., distance from candidates below a threshold) or all candidates are inspected. The complexity of LSH is approximately equal to $\mathcal{O}(h \times n^{\frac{logP_1}{logP_2}})$, $h << d$, an Retrieval performance also benefits from using the l_1 norm, which is much faster than the l_2 norm.

LSH is very sensitive to the generated partitions and data distribution. Multiple techniques have been created [27, 32, 86] to improve both temporal and retrieval performance by creating data independent hash function families that divide the space according to a structure (i.e., grid) or document distribution.

2.2.3 Data dependent hashing

Recent works have focused on creating hash functions [44, 47, 48, 61, 97, 104] that try to leverage on the latent relationships in the document collection to construct alternative representations. Spherical hashing [42] is a data-depended variant of LSH that partitions the high-dimensional query space in hyperspheres instead of hyperplanes. Torralba et al. [97] propose a method to learn small binary feature vectors from real-valued vector features using Boosting [89]. Hinton [44] generate compact hash representations using an auto-encoder based on Restricted Boltzmann machines. Binary codes generated by Spectral Hashing [104] are calculated by thresholding a subset of Eigenvectors of the Laplacian of the similarity graph. Grauman and Fergus [40] authored a comprehensive review of the state-of-the-art on binary hash indexing techniques.

The approaches based on binary codes and Hamming distances make it possible to search millions of multimedia documents in less than a second. Scalability is tackled by working in a space where the features are much smaller than the original space feature vectors, and the norm (Hamming distance) is much faster to compute than the original l_2 norm. The main loss in precision in these techniques is related to the quality of the hashes.

2.2.4 Space partitioning through clustering

Clustering techniques are one of the most used space partitioning techniques, with applications on image indexing and retrieval [48, 67]. The search space is partitioned by generating a set of centroids; vectors are assigned to the closest centroid according to a metric (e.g., Euclidean distance). k-means, a popular clustering technique, aims at finding the set of centroids that minimizes the sum of squares within-cluster distances. Lloyd [63] proposed a local search solution that is still widely applied today. On the original formulations, the initial seed centroids are selected randomly from the training data, which may greatly increase the convergence time. k-means++ [7] is a centroid selection technique that estimates a good set of seed centroids, by analyzing the distribution of the seed centroids and the training data distribution. Fuzzy c-means clustering/soft clustering [17] techniques extend the assigning of documents to multiple clusters, by keeping membership information for documents to clusters (e.g., the ratio of the distance to the centroids). Clustering techniques such as DBSCAN [35], do not set the number of centroids as a parameter, focusing instead on cluster density and points per cluster.

Clustering techniques are some of the best performing nearest neighbor search algorithms. Jégou et al. [48] proposed the Inverted File System with the Asymmetric Distance Computation (IVFADC), an index that divides the space into a set of Voronoi cells through k-means based vector quantization. Document feature vectors are indexed according to the distance to the closest centroid. The number of documents to inspect is reduced by inspecting only documents that are assigned similar centroids. Nearest neighbors are then estimated using the vector-to-centroid distances, Further works improve candidate distance

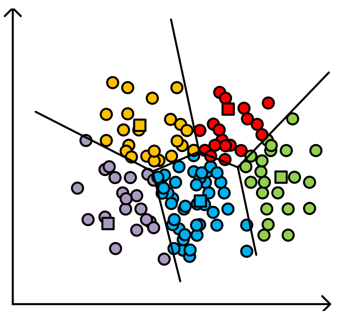


Figure 2.5: Example of k-means partitioning and the corresponding Voronoi cells produced.

computation [49] and feature vector quantization [54]. Tavenard et al. [94] proposed a technique for balancing k-means cluster size, by shifting cluster boundaries into parallel boundaries. Their experiments showed that balanced k-means reduced the variability in the number of candidates retrieved per query. Babenko and Lempitsky [10] created Inverted Multi-Index (IMI), which generalizes IVFADC by using product codebooks for individual cell construction. Babenko and Lempitsky [11] relaxed orthogonality constraints of IMI (Non-Orthogonal IMI), to better fit data distribution. Their experiments show that such partitioning still results in a large number of empty cells (60%-80% for IMI and 40%-50% for NO-IMI). Figure 2.5 shows an example of the partitions produced using k-means and of the produced Voronoi cells.

An alternative idea comes from MSIDX [96]: feature vectors' dimensions with higher cardinality have higher discriminative power than lower cardinality dimensions. The algorithm multi-sorts the indexed feature vectors according to their feature cardinalities. Indexed vectors are re-ordered with feature vectors' higher values on the most discriminative dimensions being most important. The querying algorithm follows the same multi-sort principle.

2.3 Sparse hashing for efficient partitioning and retrieval

An alternative to dense binary hashes are large, sparse, real value hashes [19]. Sparse hashing transforms dense d dimensional feature vectors into a sparse h dimensional vector, where $d << h$ and with s non-zero coefficients, $s << d << h$. In other words, compress the initial signal into a small number of coefficients (much smaller than the original space dimensionality) on a higher dimensional space.

Consider a vector $x \in \mathbb{R}^h$; it is called $s-$sparse if it has s or less non-zero values. A set of vectors $X \in \mathbb{R}^{n,h}$ is sparse if all its vectors are sparse:

$$\begin{aligned} \text{sparse(x)} &\Rightarrow \|x\|_0 \leq s \\ \text{sparse(X)} &\Rightarrow \forall i = 1, \cdots, n \ \text{sparse}(\text{x}_\text{i}) \end{aligned} \tag{2.9}$$

The goal is to find the sparse hash x that, combined with the dictionary D minimizes the error of reconstructing the original vector y, $\|Dx - y\|_2$. Equation 2.10 shows how

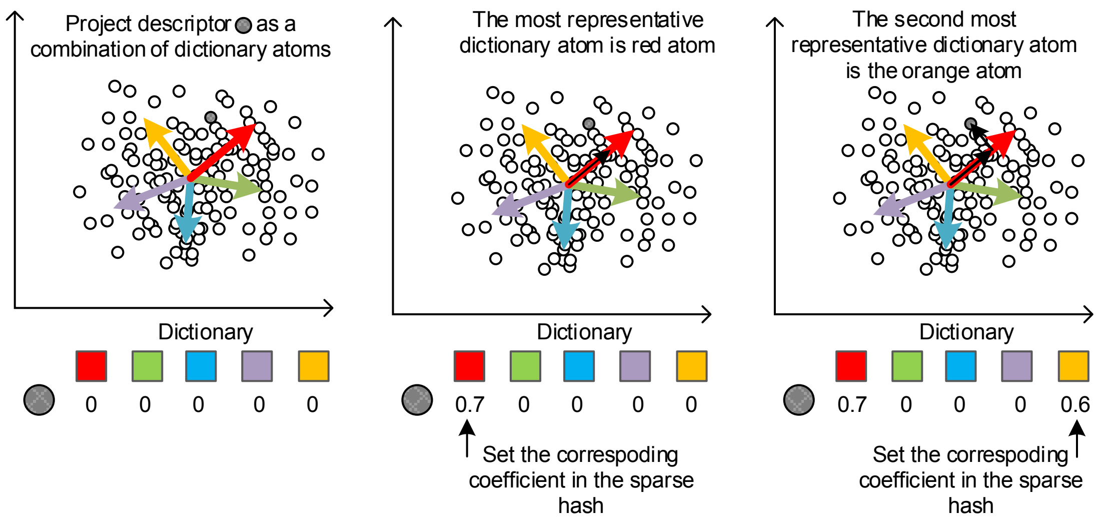

Project descriptor ● as a combination of dictionary atoms
Dictionary
0 0 0 0 0
The most representative dictionary atom is red atom
Dictionary
0.7 0 0 0 0
Set the correspoding coefficient in the sparse hash
The second most representative dictionary atom is the orange atom
Dictionary
0.7 0 0 0 0.6
Set the correspoding coefficient in the sparse hash

Figure 2.6: Computing a sparse hash using the previously computed dictionary.

to generate a sparse hash $x \in \mathbb{R}^h$ for a vector $y \in \mathbb{R}^d$, based on an existing dictionary $D \in \mathbb{R}^{d \times h}$ and s is the sparsity coefficient:

$$\begin{gathered} \arg\min_x \|Dx - y\|_2, \\ \text{subject to} \\ \|x\|_0 \leq s, \end{gathered} \tag{2.10}$$

Multiple techniques were developed to generate high dimensional sparse hashes. These techniques differ on the type of regulation applied to the hashes: l_0 penalty (e.g. Orthogonal Matching Pursuit (OMP) [84]), l_1 penalty (e.g. Lasso [93]), l_2 penalty with coefficient thresholding (e.g. Ridge [45]) or a combination of the l_1 and l_2 penalties (e.g. Elasticnet [111]). OMP controls sparsity by greedily selecting the most correlated coefficient at each iteration with the current residual (l_0 pseudo-norm penalty). Lasso does sparse selection by applying the l_1 penalty, Elasticnet's penalty is a mixture of l_1 penalties with l_2 penalties, having both the sparsity properties of l_1 penalty and the limited coefficient magnitude of the l_2 penalty.

OMP [84] solves the l_0 regression, by selecting the most correlated atom with the residual vector (the difference between the vectors Y and their reconstructions DX) at each iteration. It chooses the atom that better minimizes the reconstruction error of the hash computed in the previous iteration; computed coefficients may not change on the following iterations. OMP's greediness is tied with the retrieval process, where candidate selection starts at the partition with the highest reconstruction coefficient. Figure 2.6 illustrates this process.

2.3.1 Creating dictionaries for the generation of sparse hashes

The generation of sparse hashes requires a dictionary which transforms vectors from the dense real space into the sparse space. Figure 2.7 shows the generation of such dictionary

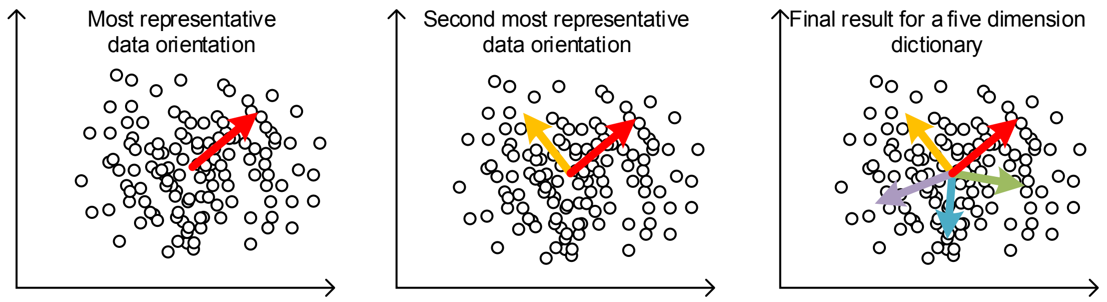

Most representative data orientation
Second most representative data orientation
Final result for a five dimension dictionary

Figure 2.7: Creating a data-driven dictionary for sparse hash generation.

using an iterative algorithm. The goal is to find the directions that better represent a collection of d-dimensional points. Each colored vector represents a direction; the red vector is the most important direction, the yellow vector is the second most important directions. The remaining directions were iteratively computed in decreasing order of importance. Computing the dictionary requires solving the following optimization problem: find the dictionary D and set of sparse hashes X that minimizes the reconstruction error against a set of vectors Y:

$$\begin{gathered}\arg\,\min_{D,X} \|DX - Y\|_2, \\ \text{subject to} \\ \|x\|_0 \leq s, \\ \text{for } x \in X\end{gathered} \tag{2.11}$$

where $Y \in \mathbb{R}^{d \times n}$ are the original document vectors (one per column), $D \in \mathbb{R}^{d \times h}$ is a dictionary, learned from the data, $X \in \mathbb{R}^{h \times n}$ are the sparse hashes and x is a sparse hash vector (row of X).

Solving for both D and X is NP-hard. Dictionary computation algorithms such as K-Singular Value Decomposition (KSVD), estimate the dictionary atoms (i.e., D column vectors) that minimize the reconstruction error. KSVD [2] offers an approximate solution to this problem, by iteratively alternating between (1) sparse coding of the training examples using the current dictionary and (2) updating the dictionary atoms to fit training data better. Stochastic gradient descent techniques (e.g., [83]) update each example per iteration, to minimize reconstruction error. Cherian et al. [26] presented an index based on hashes created using l_1 regression and the Newton Descent for codebook learning. Borges et al. [19] presented an index based on sparse hashes created using l_0 regression and a codebook learned through KSVD.

2.3.2 Inverted indexes for distributed indexing and retrieval

Inverted index structures are a widely studied for indexing and retrieval on a single machine, being an essential part of the IR curriculum [22, 68]. Most of the literature is focused on indexing text on inverted index structures. On classical text indexing systems, each *document* is represented by the set of words that are on it. A possible visualization of multiple documents with multiple words is a *matrix*, where the rows are all the words in all documents and the columns are the documents. Each entry of the matrix is the number of times a word appears on a document, The set of words on all index documents is called the *dictionary*. This matrix is highly *sparse*, and not adequate for processing at scale.

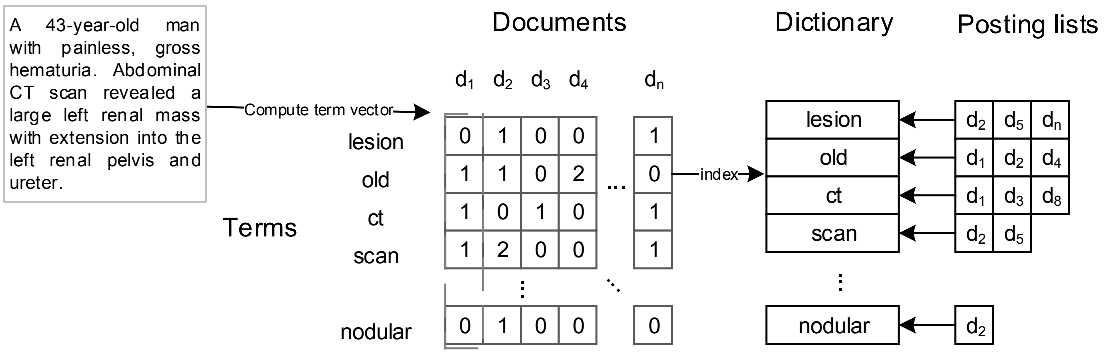


Figure 2.8: Indexing text using posting lists

A solution is to store only the non-zero entries on an *inverted index*, where each index entry corresponds to a word and contains a list of all documents which contain that term (*posting list*). At query time, the postings lists that correspond to the query terms are inspected. Figure 2.8 shows a simple example using biomedical terms. A textual document is decomposed into a set of term vectors. These term vectors are then indexed on the posting lists that match the terms that appear in the document.

Consider now a real-valued feature vector: each *feature* corresponds to a *word*, meaning that documents and feature vectors can be represented as a matrix. The key difference is that regular feature vectors are dense, meaning that representing the feature vector matrix as a set of posting lists will not be effective. If the hashes are sparse, one can create inverted indexes with similar structure and proprieties.

2.3.3 Discussion

This section described existing solutions to the k-NN retrieval problem that can achieve good retrieval effectiveness by inspecting a small subset of the indexed feature vectors, However, these techniques assume that the index fits a single node and that the time it takes to access indexed feature vectors is uniform (i.e., all feature vectors are in the main memory (RAM)). For example, the NO-IMI algorithm Babenko and Lempitsky [11] generated a large number of empty partitions even after relaxing orthogonality constraints. Such unbalances become a problem if the indexing process is performed in situations where the access to resources is not uniform (e.g., the index is distributed across multiple nodes). The following sections detail how existing works deal with the distribution of the index to multiple nodes.

2.4 Scaling out indexing and retrieval

The algorithms described in the previous section assume that all the data is in the main memory (RAM). However, RAM is limited to the hundreds of GB per node, which is not enough to deal with multimodal documents with dozens of features. Thus, one must find techniques to enable search at these scales effectively.

2.4.1 Scaling indexes to disk

The first place to store indexes that do not fit into memory is the secondary memory (e.g., hard drive or solid state drive). This problem has been studied in areas as varied as database systems, text indexing, with few examples in multimedia retrieval [15, 41, 59].

The M-Tree variants (LHI-Tree [41] and PM-Tree [92]) reduce vector dimensionality by indexing vectors according to their distances to randomly selected pivot vectors. The LHI-Tree applies a hashing function to the vectors in the new dimensional space, and store points with nearby hashes to the same disk region to increase the probability of finding nearest neighbors in the same disk read. The NV-Tree [59] re-projects feature vectors into an uni-dimensional space so that nearest neighbors are likely stored in that disk area, minimizing disk access. The LSH Forest [15] is based on the LSH algorithm with variable width codes, stored in prefix B-Trees. Their disk-based representation transverses the codes through their prefixes (stored nearby on disk) to find candidates and then re-rank them on the original space. The solutions on the literature rely on re-projections, clustering or hashing to store likely nearest neighbors to the same disk region. Also, increasing index size limit without an increasing parallelization leads to disk I/O becoming a disproportionate portion of retrieval time.

2.4.2 Distributed indexing and retrieval

As data reaches the processing limits of a single processing node, scaling out indexes across multiple nodes becomes the inevitable next step. Multimodal multi-feature indexes pose a challenge to feature-to-node allocation policies: *how to distribute indexed feature vectors across nodes?*

There are two main index partitioning techniques to divide an index across m nodes effectively [22]:

- divide features across multiple nodes (horizontal partition or sharding);
- divide feature vector dimensions across multiple nodes (vertical partition).

Figures 2.9 and 2.10 give a visual representation of such partition techniques. In sharding, there are two main approaches to distribute feature vectors across shards:

- a two-tier search process - searching a small index with a subset documents from all shards and select shards with the most results on the subset [12];

- select shards probabilistically by comparing individual shard document statistics with the query statistics [4].

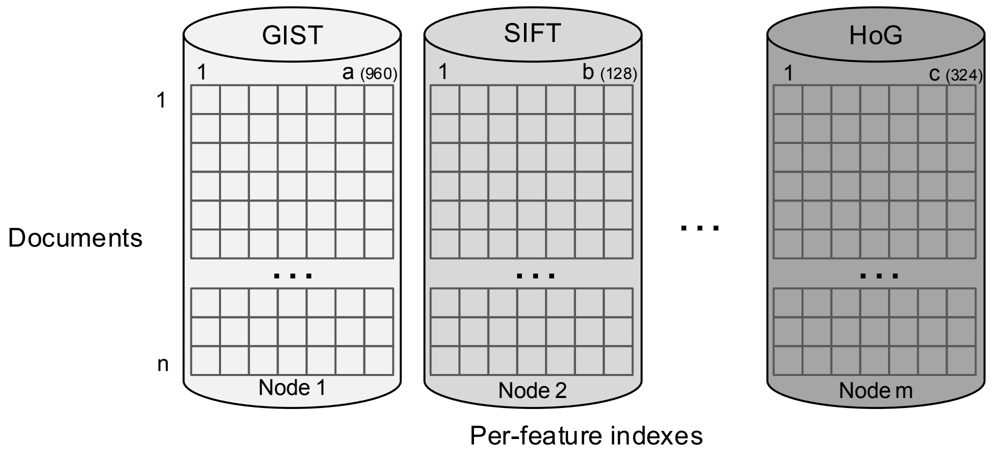


Figure 2.9: Vertical index partitioning

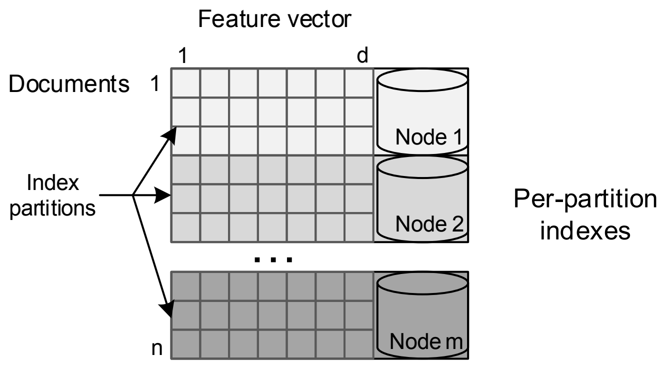


Figure 2.10: Horizontal index partitioning

2.4.3 Distributed text indexing

Distributed text indexes offer retrieval precision comparable to a full index inspection by querying a small subset of the full set of indexing nodes [56]. Kulkarni and Callan [56] compared three document sharding policies: random, source (URL-based) and topic (content-based clustering), over large textual document datasets (500 million documents with 381 billion words). They showed that Topic-based sharding offers the retrieval precision in line with exhaustive search while inspecting only 20% of the document collection. Dai et al. [31] builds on top of this work by using query logs as the seeds for the cluster centroid creation. Topic-based sharding closely matches sparse hash partitioning: extracting principal dimensions of the data is similar to topic-creation on text indexes. Their experiments show that vertical sharding offers the best temporal performance in a distributed setting, without an increase in load imbalance.

The bulk of work in distributed text indexes (sparse, very high dimensional term vectors) contrasts with the equivalent work done in the multimedia domain (dense document feature vectors). Multimedia document distribution is a less explored area, where document allocation policies are either random or based on existing partitions of single node algorithms. This is particularly challenging due to the data high-dimensionality and its unknown underlying structure.

Vertical partitioning can work at two levels for systems with multiple indexes: distribute indexes across nodes or partition individual feature vector dimensions across nodes. As textual feature vectors are sparse, they can be easily partitioned per term (partitioning dimensions across nodes): index documents in partitions with non zero values. For dense feature vectors, there is no simple technique to apply vertical partitioning in their original form.

2.4.4 Distributed multimedia indexing

The distribution problem is an interesting area of study that must balance between two factors: *how to design a system that provides a good distribution of documents and queries across machines while giving locality guarantees (vectors on the same machine have some degree of similarity) for efficient retrieval?*.

One of the works that address the efficient distribution of a media index is by Ji et al. [50]. They designed a distributed near-duplicate image retrieval architecture and tested multiple horizontal and vertical sharding indexing techniques, based on Vocabulary Tree model quantization.

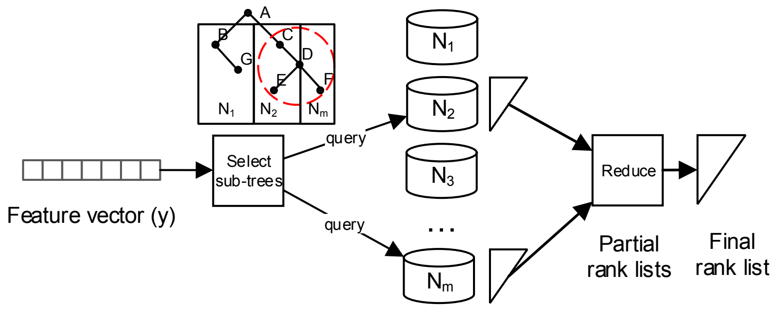


Figure 2.11: Sub-tree based sharding

Batko et al. [14] describe a large scale (50 million images from Flickr) distributed content based image retrieval (CBIR) system capable of sub-second querying time. Their system includes distributed crawling, feature extraction, and retrieval. Local node indexing is based on an enhanced M-tree [92] and data distribution between nodes is based on an M-Chord [81] peer-to-peer search network. Their distribution architecture is intertwined with the intra-node indexing algorithm, making it hard to generalize for alternative indexers.

Aly et al. [3] describe a distributed kd-tree algorithm that distributes sub-trees across multiple nodes, Figure 2.11 Their benchmarks show that their approach leads to better results than creating independent kd-trees on all the machines, their results show very high single query time (about eight seconds) on an index with million documents. The effectiveness of sub-tree based index partitioning is reduced when the dimensionality of the vectors to index increases [103], meaning that more nodes have to be queried.

Lee [57] offer a full distributed image search engine with manual relevance feedback. The indexing algorithm is based on clustering using self-organizing tree maps and distributing the clusters across machines, Figure 2.12 The paper mentions efficiency information by mapping precision vs. number of clusters inspected.

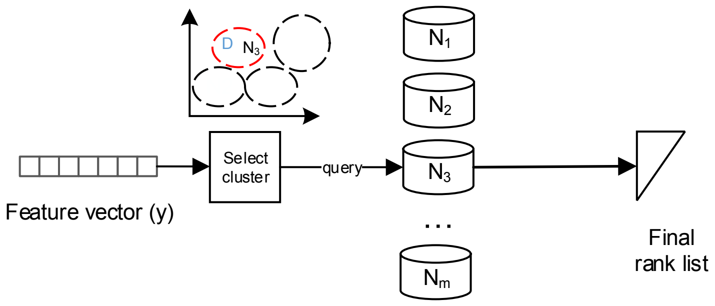


Figure 2.12: Cluster-based sharding

The MapReduce model [33] is widely deployed to process large amounts of data on a machine cluster. It is divided into a Map procedure, which is the processing function to apply to data and a Reduce procedure that aggregates the results from the Map procedure. Parallel data processing is achieved by running Map and Reduce on multiple machines in parallel with disjoint data partitions, Figure 2.13 Data distribution and communication are managed by the MapReduce framework.

Yang et al. [108] present a small scale image search engine, with distributed feature extraction and search on a distributed inverted index. The authors report experiments with increasing number of nodes (1, 2, 4) and 2000 feature vectors, and show that time spent on the feature extraction decreases linearly with the increase in nodes, while search time decreases sub-linearly.

Moise et al. [74] tested hierarchical cluster partitioning using Map-Reduce on a 100 million image index. Experiments show that the overhead behind the Map and Reduce operations is considerable (e.g., copying data to Hadoop Distributed File Systems), as it is only optimized for massive batches of queries.

Muja and Lowe [79] recently tested several approximated nearest neighbor search algorithms on a distributed setting with four nodes and 80 million feature vectors. Their approach is based on disjoint data partitioning and a map-reduce operation where all indexes are searched and their results combined by a master node. Mohamed and Marchand-Maillet [73] improved Map-Reduce for media retrieval by executing the Map and Reduce stages in parallel using the Message Passing Interface (MPI) protocol. They show that, although it improves performance versus regular Map-Reduce, it is still worse than a manual MPI-based implementation. As Map Reduce requires Map and Reduce nodes to be data agnostic, indexes must either query all nodes for all queries (e.g. [79]), which does not meet our efficiency goal, or have all nodes have access to the full index ([74]), which is limited by the time the node takes to fetch the relevant index subset.

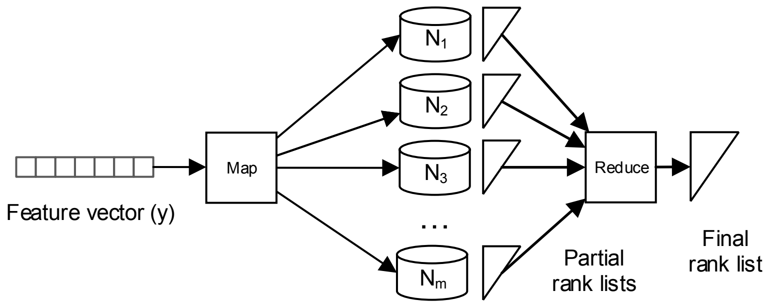


Figure 2.13: Map-Reduce based sharding

As Map Reduce requires Map and Reduce nodes to be data agnostic, indexes must either:

- query all nodes for all queries [79], which reduces efficiency, or
- have all nodes accessing the full index [74], which is limited by the time it takes to fetch the relevant index subset.

As Map-Reduce was designed for batch data processing, it is optimized for working with massive batches of queries at a time, instead of individual queries.

2.4.5 Consistent hashing for load balancing

Assigning documents and requests to nodes in a consistent manner (i.e., the same request will always be assigned the same node) is an area of study on distributed systems research [55, 95]. Consistent hashing [55] and Rendezvous hashing [95] are consistent distribution techniques for distributed systems, designed to improve cache hit rate by mapping similar requests to the same node while keeping the load balanced across nodes. Both algorithms are based on assigning both documents and nodes to an uni-dimensional space, where documents are assigned to the closest nodes. Consistent hashing is based on mapping the distribution space into a numeric angle value. Nodes are assigned multiple pseudo-random angle values. Documents are assigned to nodes by computing the angle value from a hash and selecting the node with the closest angle value. Rendezvous or Highest Random Weight hashing is based on generating node indexing priorities based on hashing the concatenation of the document and the node identifiers. Recently, Mirrokni et al. [72] developed Consistent hashing with bounded load, that improves load distribution by capping the maximum number of documents per node. Documents that would have been assigned to that shard are forwarded to the next node with available capacity. Although it improves load balancing compared to regular, consistent hashing, it still designed for exact matches. These techniques were designed for load balancing on static content nodes or caches; similarity is measured by cryptographic hash functions (e.g., SHA256, MD5) designed for exact hash matches and to explicitly produce very different hashes for near-duplicate or similar documents.

2.4.6 Commercial and open-source solutions

Outside academia, web search and social media companies are the key players for distributed processing of large amounts of media documents with strict time constraints. Pinterest researchers [52] revealed the architecture behind their image recommendation system (related pins), that recommends images that are related to user posts, and *Similar Looks*, that recommends clothing articles from online stores that look like to the ones on an image. Image feature vectors are extracted using the Caffe deep learning framework, *salient color signatures* and *local features for object detection* which are called *Incremental Fingerprints*. The image collection is divided into epochs by upload date and feature type. Indexes and features are sharded by image signature (MD5 hash of the fingerprint) and assigned to machines in the cluster using Hadoop. The system uses two indexes: a disk-based partially memory cached for the fingerprints for candidate selection and an image annotation and user information graph for re-ranking Static Performance evaluation is only reported as the performance on a single machine, and the Live Performance evaluation was based on user engagement (number of pins) instead of time and precision.

Existing open-source and commercial projects such as Apache SOLR[1], Elastic Search[2] or Sphinx[3] offer a fully featured search platforms for text. These platforms offer tools to set a schema for the index, choose which algorithms and features to use for processing and offer REST APIs that enable easy website integration.

SOLR and Elasticsearch offer index distribution through horizontal sharding using Apache Lucene[4] as the backend for text indexing and retrieval. LIRE (Lucene Image Retrieval) [64] is a library based on Lucene that enables indexing image feature vectors as postings lists.

2.4.7 Discussion

Efficient nearest neighbor search requires effective, similarity-based search space partitioning techniques. k-NN search architectures should take advantage of the characteristics of new distributed systems. The current state-of-the-art [73, 74, 79, 108] builds on adaptations of existing generic distributed platforms to handle the problem of efficient multimedia index distribution. Sparse hashing techniques have the potential to create better similarity-based index partitions due to their similarities to existing solutions developed for distributed textual indexes [31, 56].

[1]https://lucene.apache.org/solr/
[2]https://www.elastic.co/
[3]http://sphinxsearch.com/
[4]https://lucene.apache.org/

2.5 Rank fusion for multimodal retrieval

Multimedia features provide an approximation to what humans consider relevant in multimodal documents. In images, features can represent colors, edges, texture, among other information. For text, features may represent the words or abstractions in the form of topics. As we are dealing with an approximation of the world, there is no canonical order in which to combine the results for individual indexes. Rank fusion algorithms bridge the gap between the different types of similarities measured by each feature. The goal is to combine the set of rank lists into a rank list that adequately models similarity across features. For example, documents that are highly ranked across feature rank lists should be ranked higher on the final rank. Rank fusion techniques follow the effects described by Vogt and Cottrell [102] when ranking documents:

- **The Skimming Effect:** different retrieval approaches retrieve different relevant items
- **The Chorus Effect:** relevant items will be present on the ranks of several retrieval approaches;
- **The Dark Horse Effect:** retrieval approaches may rank a particular document unusually more (or less) accurately than average for a query.

Figure 2.14 gives an overview of the search process: (i) Documents in the rank lists are grouped by a unique identifier, id. (ii) The score of the document in the final ranking is computed as a function of its score and ranking across rank lists.

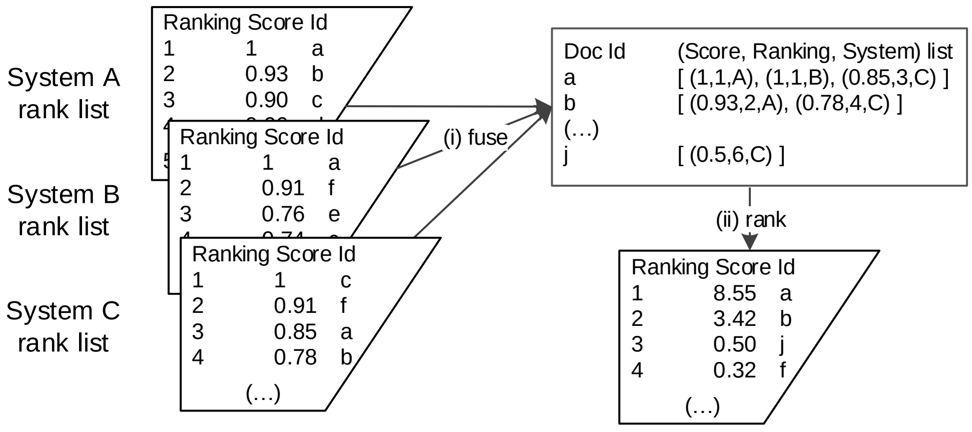


Figure 2.14: The rank fusion process

2.5.1 Score-based rank fusion

Shaw and Fox [90] introduced score based fusion (CombSUM, CombMAX, CombMNZ and other variants). Score-based approaches are based on the scores returned by the retrieval systems. The best performing approaches are CombSUM, CombMAX, and CombMNZ.

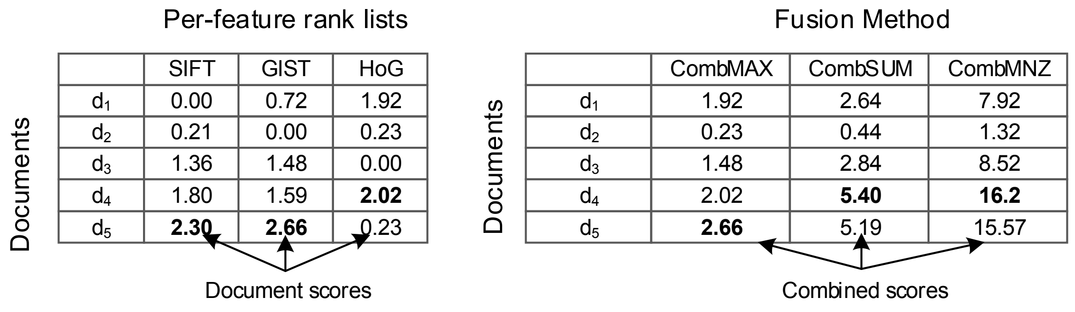
For each document i, the score after fusion can be computed as:

$$\text{CombSUM(i)} = \sum_{k=1}^{nrl(i)} S_k(i), \tag{2.12}$$

$$\text{CombMAX(i)} = \max(S), \forall S \subset D_i, \tag{2.13}$$

$$\text{CombMNZ(i)} = nrl(i) \times \text{CombSUM}(i), \tag{2.14}$$

where $S_k(i)$ is the score of the i document on the rank list k. *nrl(i)* is the number of rank lists document i appears. *nrl(i)* varies between 0 (the document i does not appear on any rank) and the total number of ranks (the document i appears on all ranks). *D(i)* are the ranks that contain the document i. These approaches are widely researched and evaluated [58, 75]; however, they have a key limitation: rank lists must give meaningful scores to documents. Score distribution and normalization dramatically influence performance [75]; effective fusion can only be achieved if scores follow similar distributions across rank lists. Figure 2.15 shows an example of the application of score-based rank fusion techniques on a set of three rank lists (as produced by SIFT, GIST and HoG features). The leftmost table contains the score of each document for each feature type (the higher, the better). The rightmost table contains the score of the combination of the rank lists using multiple score-based techniques.


Per-feature rank lists
Fusion Method
Documents
Documents
Document scores
Combined scores

	SIFT	GIST	HoG
d_1	0.00	0.72	1.92
d_2	0.21	0.00	0.23
d_3	1.36	1.48	0.00
d_4	1.80	1.59	**2.02**
d_5	**2.30**	**2.66**	0.23

	CombMAX	CombSUM	CombMNZ
d_1	1.92	2.64	7.92
d_2	0.23	0.44	1.32
d_3	1.48	2.84	8.52
d_4	2.02	**5.40**	**16.2**
d_5	**2.66**	5.19	15.57

Figure 2.15: Score based rank-fusion examples

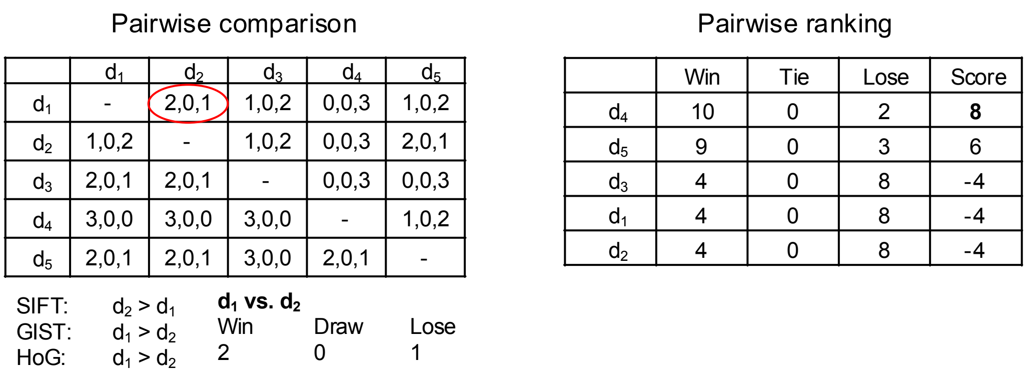
2.5.2 Rank-based rank fusion

Rank-based methods use the position of documents in rank lists as scores. They remove the need for scores, which enables the combination of rank lists from more sources (e.g., from commercial web search engines).

Voting algorithms are based on election theory. Documents are candidates, and each rank list is a voter. The presence of documents in a rank list counts as a vote. Popular approaches include Bordafuse [9] and Condorcet Fuse [76], based on Condorcet voting.

Bordafuse is a voting algorithm based on the positions of the candidates. It was invented by Jean-Charles de Borda in the eighteen century and adapted for rank fusion by Aslam and Montague [9] For each rank list, the document gets a score corresponding to its (inverse) position on the rank. The fused rank list is based on the sum of all per-rank scores.

Condorcet is a voting algorithm introduced by Marquis de Condorcet in the eighteen century and also adapted by Montague and Aslam [76] for rank fusion. It is based on a majoritarian method which uses pairwise comparisons for ranking. Consider a set of documents D and a set of rank lists R. For each pair of documents $d_i, d_j \in D$, the algorithm compare the number of times d_i ranks above d_j over all rank lists R. Candidates are then sorted in descending order according to the number of times they rank above other documents minus the times they rank below other documents. Figure 2.16 shows an example of the application of Condorcet. The leftmost table contains the pairwise comparisons between all documents as (wins, ties, losses) triples. The rightmost table contains the combined score of all pairwise comparisons and the final rank.



Pairwise comparison

	d_1	d_2	d_3	d_4	d_5
d_1	-	2,0,1	1,0,2	0,0,3	1,0,2
d_2	1,0,2	-	1,0,2	0,0,3	2,0,1
d_3	2,0,1	2,0,1	-	0,0,3	0,0,3
d_4	3,0,0	3,0,0	3,0,0	-	1,0,2
d_5	2,0,1	2,0,1	3,0,0	2,0,1	-

		d_1 vs. d_2		
SIFT:	$d_2 > d_1$	Win	Draw	Lose
GIST:	$d_1 > d_2$	2	0	1
HoG:	$d_1 > d_2$			

Pairwise ranking

	Win	Tie	Lose	Score
d_4	10	0	2	**8**
d_5	9	0	3	6
d_3	4	0	8	-4
d_1	4	0	8	-4
d_2	4	0	8	-4

Figure 2.16: Condorcet pairwise comparisons and ranking

Rank and Voting-based approaches do not require normalization and are more stable to differences in rankings between lists.

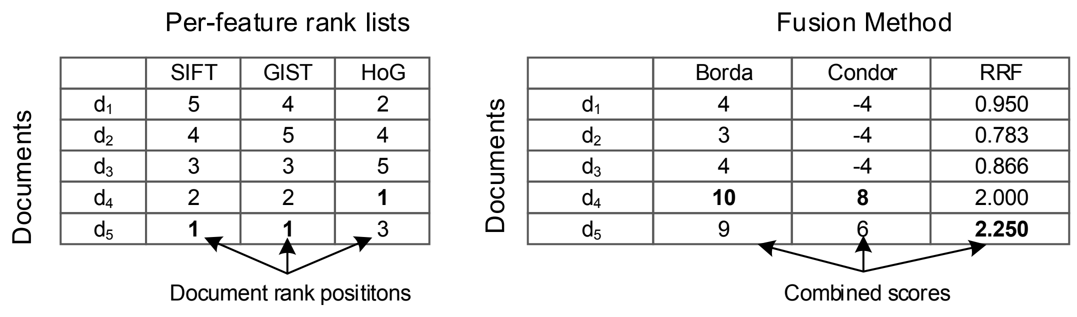
$$\mathrm{RR(i)} = \sum_{k=1}^{nrl(i)} \frac{1}{R_k(i)}, \tag{2.15}$$

$$\mathrm{RRF(i)} = \sum_{k=1}^{nrl(i)} \frac{1}{h + R_k(i)}, \text{ with } h = 60. \tag{2.16}$$

where $R_k(i)$ is the rank of document i on the k rank.

Figure 2.17 shows an example of the application of rank, and voting-based rank fusion techniques on a set of three rank lists (as produced by SIFT, GIST and HoG features) (similar to Figure 2.15). The leftmost table contains the rank of each document for each feature type (the higher, the better). The rightmost table contains the score of the combination of the rank lists using multiple rank-based techniques.

Rank based tend to outperform score based approaches [30, 46] and, in some cases, learning to rank methods [30, 66]. They can also be deployed on use-cases where the score is not available.



	SIFT	GIST	HoG
d_1	5	4	2
d_2	4	5	4
d_3	3	3	5
d_4	2	2	**1**
d_5	**1**	**1**	3

	Borda	Condor	RRF
d_1	4	-4	0.950
d_2	3	-4	0.783
d_3	4	-4	0.866
d_4	**10**	**8**	2.000
d_5	9	6	**2.250**

Figure 2.17: Rank and voting based rank-fusion examples

2.5.3 Supervised techniques

The linear combination (LC) approach [13, 101], also called wCombSUM, assigns linear weights to rank lists,

$$\mathrm{LC(i)} = \sum\nolimits_{k=1}^{nrl(i)} \alpha_i S_k(i), \tag{2.17}$$

where α_i is a linear weight given to the rank list i. Linear weighting can be similarly applied to other score or rank based fusion algorithms. Other recent works have worked with other more data-driven schemes to find adequate weights and how to apply them [62, 69].

Anava et al. [5] describe a probabilistic fusion framework which generalizes existing rank and score-based fusion methods and enables the weighted combination of method components. For example, GeoCMNZ is a weighted combination of the frequency and score components of CombMNZ. Weights are estimated by "leave-one-out" cross-validation. Their experiments show that GeoCMNZ can outperform CombMNZ, depending on the metric and queries under optimization. Bhowmik and Ghosh [18] extend unsupervised methods, incorporating document features (common on LETOR techniques) into ranking using generalized linear models.

2.5.4 Learning to rank

LETOR techniques use machine learning to learn ranking models, using queries with curated rank lists (assessed by experts) as training data. They can be divided into three main families [24]:

- *Pointwise*: optimize each document-query point loss function at once: Coordinate Ascent [70], Random Forests [20],
- *Pairwise*: optimize document pairs loss function at once: RankNet [21], RankBoost [37], MART [38], LambdaMART [105], SVM-rank [53],
- *Listwise*: optimize full ranks/features loss function at once: AdaRank [107], ListNet [23], ListMLE [106].

Training effective LETOR models require vast amounts of training data (e.g., the MSLR-WEB30k dataset has over 30,000 queries with relevance judgments [85]). Vargas-Munoz et al. [100] select rank lists and fusion methods using genetic programming techniques, achieving good retrieval performance with lower training time than other supervised approaches. These techniques are based on machine learning techniques, which require vast amounts of training data to generate models with high degrees of complexity. For example, there are cases where decision tree models can generate trees seventeen levels deep [8]. Such model complexities are tied to the large amounts of training data. For example, the widely popular LETOR dataset [71] (v4) contains over 25 million web pages and two query sets from Million Query track of TREC 2007 (1700 queries with relevance assessments) and TREC 2008 (800 queries with relevance assessments). The MSLR-WEB30k dataset has over 30,000 queries with relevance assessments.

2.5.5 Discussion

Existing research on rank fusion and LETOR can outperform the best results from individual features [30, 85, 100]. However, both rank fusion and LETOR techniques impose no constraints on the number of rank lists used, resulting in more nodes queried on a vertically partitioned index. Using fewer rank lists can reduce the complexity, by reducing the number of features that are needed to compute the final search results.

2.6 Datasets

A multimodal search engine architecture is composed of multiple components, meaning that there is no single dataset that addresses all the processes we want to benchmark at the desired scales. Table 2.1 contains an overview of the datasets used throughout this thesis.

2.6.1 TREC CDS and PMC Open Access

The TREC CDS dataset[5] is the Open Access Subset of PubMed Central (PMC). PMC is a database of open access full-text biomedical articles. It contains over 1.25 million articles. The queries are medical case summaries created from actual medical records. They describe the patient medical history, symptoms, tests, diagnosis, and the care process. The queries are from one of 3 types:

- **Diagnosis:** What is the patient diagnosis?
- **Test:** What tests should the patient receive?
- **Treatment:** How should the patient be treated?

[5]https://trec-cds.appspot.com/

Table 2.1: Datasets for indexing, retrieval and fusion

Dataset	Type	Train & val	Test index	Queries
PMC Open Access	Articles (text & image)	—	5,000,000 articles 15,000,000 images	—
TREC CDS	Articles (text & image)	—	1,250,000 articles 5,000,000 images	90
ImageCLEF 2013 Medical Cases	Articles (text & image)	—	75,000 articles 300,000 images	36
ImageCLEF 2013 Medical Ad-hoc	Article Images (text captions & image)	—	300,000 images & captions	35
Tiny Image	Features (GIST)	—	79,302,017	—
ANN_1M SIFT	Features (SIFT)	500,000	1,000,000	1,000
ANN_1M GIST	Features (GIST)	100,000	1,000,000	10,000
ANN_1B SIFT	Features (SIFT)	100,000,000	1,000,000,000	10,000
TREC FedWeb	Ranked doc. lists	—	148 ranks 1,000 docs each	200

There is a set of 30 queries (10 for each type) for each edition of the track, for a total of 90 queries (2014 to 2016). Relevance judgments have 3-levels: "definitely relevant" for answering questions of the specified type about the given case report, "definitely not relevant", or "potentially relevant" if it may be relevant in from a literature review perspective.

The PMC Open Access is a superset of the TREC CDS dataset that is updated daily with new documents. As of January 2017, it contains over 4.7 million documents with over 15 million images.

2.6.2 ImageCLEF 2013 Medical

The ImageCLEF 2013 Medical dataset [43] is a subset of PMC with 75,000 journal articles and over 300,000 images. There are a total of 66 queries from 2011, 2012 and 2013 editions of the track. Relevance judgments are binary: Relevant (rel = 1) or Not relevant (rel = 0). The ImageCLEF 2013 Ad-hoc Medical image retrieval is a similar task, with the results are the images with captions from the articles.

2.6.3 Tiny Images

The Tiny Images dataset [98], contains 79,302,017 images and the corresponding GIST feature vectors [82] with 384 dimensions. This makes the dataset particularly adequate to simulate the conditions of a large-scale high-dimensional search.

2.6.4 ANN dataset

The ANN datasets [48, 49], were designed to evaluate the quality of nearest neighbors search algorithm with different feature vector and database sizes[6]. It is composed of two main vector sets:

- two sets of one million feature vector sets (GIST with 960 dimensions and SIFT with 128 dimensions), and corresponding training, validation and queries vector subsets [48];
- one set of one billion 128 dimensional SIFT feature vector sets and corresponding training, validation and queries vector subsets [49].

2.6.5 TREC FedWeb 2013

The TREC FedWeb 2013 dataset is an extension to the 2012 TREC FedWeb dataset [80]. It contains a collection of search results sampled from 148 search engines over 200 queries. Each search engine is related to one or more search categories, such as web, news, travel, and video. Relevance judgments span 5 categories: Navigational (rel = 1), Key (rel = 1), Highly relevant (rel = 0.5), relevant (rel = 0.25) and Not relevant (rel = 0).

2.7 Summary

The challenges for retrieval systems that deal with large amounts of heterogeneous data are far from over. This section described how existing literature on multimodal indexing, index distribution and result fusion, and areas where this thesis research can provide additional insights (e.g., distribute a multimodal index to multiple nodes by similarity). Sparse hashing techniques have the potential to create index partitions that can be the basis of a similarity-based distributed index. For rank fusion, designing a fusion algorithm that uses only a subset of features can improve the efficiency of a vertically partitioned index.

[6] http://corpus-texmex.irisa.fr/

Chapter 3

Learning to combine rank lists

Multimodal retrieval architectures are built on top of multiple features and ranking methods to produce the final ordered list of documents, rank list. However, using multiple features and method for retrieval produces multiple rank lists, with variable levels of retrieval performance. When working on new retrieval tasks and data with no annotated training or test data, it is not possible to know what rank lists offer the best retrieval performance.

Table 3.1 shows an example of the retrieval performance of such lists for the TREC CDS dataset: there is no single method that is the best for all metrics and datasets. The goal of rank fusion algorithms is to create a single, unified rank list that leverages on rank lists produced using multiple features and retrieval methods.

Section 2.5 described how rank fusion techniques follow the Skimming, Chorus and Dark Horse Effects Vogt and Cottrell [102]. For example, Reciprocal Rank (RR) [110] and Reciprocal Rank Fusion (RRF) [30] give scores to documents by summing the inverse of the document ranking in the individual rank lists, Skimming Effect (retrieve top-ranked documents for each retrieval approach). The Chorus effect is visible in CombMNZ [90]; the final sum of document score is multiplied by the number of rank lists where a document appears, *nrl*. However, existing works [5, 18, 30, 110] only evaluate the retrieval effectiveness of the final combined rank list. They do not provide a solid hypothesis on why they can model document relevance across rank lists (i.e., the correlation between the position of a document in the rank list and the document relevance). This chapter studies how existing rank fusion techniques fit real-world data and how existing techniques can be improved by better exploiting how these techniques model real-world datasets.

Unsupervised rank fusion algorithm's performance degrades when working on rank lists with large differences in retrieval performance. LETOR algorithms solve this problem by selecting which rank lists to use to generate the final rank list, given large amounts of queries with manually annotated results as training data. However, adding more rank

Table 3.1: Retrieval performance of a subset of the rank lists produced using various ranking functions (BM25L, BM25+, Language Models (LM) and TF_IDF, (D)) on multiple combinations of fields (**f**ull text, **a**bstract and **t**itle) query expansion (MeSH, SNOmed, Shingles and Pseudo-relevance feedback, PRF) for the TREC CDS dataset.

	TREC CDS 14		TREC CDS 15	
Rank lists	NDCG@20	P@10	NDCG@20	P@10
BM25+_f_MSH_NoPRF	0.2240	0.2700	0.2362	0.3333
BM25+_f_MSH_PRF	0.2771	**0.3767**	0.2573	0.3667
BM25+_f_NoEXP_NoPRF	0.2350	0.2900	0.2461	0.3367
BM25+_f_SHI_NoPRF	0.2273	0.2833	0.2376	0.3200
BM25+_f_SHI_PRF	0.2768	0.3767	0.2527	0.3400
BM25+_f_SNO_NoPRF	0.2265	0.2833	0.2270	0.3467
BM25+_f_SNO_PRF	0.2558	0.3300	0.2329	0.3400
BM25+_fat_MSH_NoPRF	0.1887	0.2300	0.2009	0.2700
BM25+_fat_MSH_PRF	0.2656	0.3433	0.2626	0.3567
BM25+_fat_NoEXP_NoPRF	0.1908	0.2367	0.2139	0.2900
BM25+_fat_NoEXP_PRF	0.2805	0.3400	**0.2741**	**0.3900**
BM25+_fat_SHI_NoPRF	0.1913	0.2300	0.2197	0.2967
BM25+_fat_SHI_PRF	0.2671	0.3467	0.2420	0.3433
BM25+_fat_SNO_NoPRF	0.1873	0.2333	0.2090	0.3033
BM25+_fat_SNO_PRF	0.2556	0.3167	0.2384	0.3367
BM25L_f_MSH_NoPRF	0.2200	0.2533	0.2420	0.3300
BM25L_f_MSH_PRF	**0.2857**	0.3667	0.2629	0.3567
BM25L_f_NoEXP_NoPRF	0.2318	0.2900	0.2533	0.3433
LM_fat_MSH_NoPRF	0.1941	0.2500	0.2075	0.2700
LM_fat_MSH_PRF	0.2635	0.3233	0.2623	0.3333
LM_fat_NoEXP_NoPRF	0.2113	0.2667	0.2268	0.3033
LM_fat_SHI_NoPRF	0.2064	0.2767	0.2277	0.3300
LM_fat_SHI_PRF	0.2101	0.2733	0.2697	0.3700
LM_fat_SNO_NoPRF	0.2088	0.2367	0.2189	0.3133
LM_fat_SNO_PRF	0.2469	0.3067	0.2397	0.3400

lists to fusion has some disadvantages: generating more rank lists increases the number of computational resources used (i.e., less efficient) and can lead to decreasing improvements in retrieval precision. This chapter proposes Learning to Fuse (L2F), a rank fusion method that greedily selects which rank lists to combine, based on incremental information gains. L2F goal is to improve retrieval effectiveness through the diversification of rank selection, improving efficiency by minimizing the number of rank lists (and thus features) to use.

The contributions of this chapter are three-fold:

- an *exploration* of the effects of document distribution across rank lists, described by Vogt and Cottrell [102];
- the *Inverse Square Ranking (ISR)* rank fusion algorithm family, based on the exploitation of these effects;

- the *learning to fuse* algorithm based rank fusion algorithms that selects a small set of rank lists that optimizes a given retrieval metric.

3.1 Modeling rank fusion scores according to document relevance

Rank fusion techniques compute a combined rank list according to document rankings over multiple rank lists. As rank lists provide different degrees of similarity, documents that are present on multiple rank lists are more likely to be relevant. The goal of this section is to study this effect be measuring *how relevant documents are distributed across rank lists.*

Relevance was studied using rank lists from two different domains: vertical web search (TREC Federated Web Search) and biomedical literature search (TREC Clinical Decision Support). The TREC CDS dataset [87, 91] is the Open Access Subset of PubMed Central (PMC). A total of *64 rank lists* were created as a combination of different retrieval models and query expansion methods. It is described in detail in section 2.6.1. The TREC Federated Web Greatest Hits dataset [34] is a collection of vertical search engines rank lists that cover a broad range of categories, including news, books, academia, travel, among others. It contains a collection of the top 10 documents sampled from *148 rank lists* from vertical search engines over 50 queries per year. It is described in detail in section 2.6.5.

Table 3.2: Number of rank lists per document *nrl* for the TREC FedWeb dataset.

nrl	1	2	3	4	5+
% of documents	97.3%	1.5%	0.5%	0.4%	0.3%

Table 3.3: Number of rank lists per document *nrl* for the TREC CDS dataset. Results are divided into ranges.

nrl	1-12	13-25	26-38	39-51	52-64
% of documents	71.5%	15.5%	6.8%	4.9%	1.3%

3.1.1 Exploring the distribution of documents across multiple rank lists

Multiple rank lists provide different and complementary views of the collection. The goal of this section is to measure whether the number of rank lists, *nrl*, where a document appears is correlated with the document relevance. This is related to the Chorus effect. Tables 3.2 and 3.3 show the percentage of documents in the rank lists which are present on multiple rank lists for the TREC FedWeb and TREC CDS datasets. On the TREC FedWeb data, over 97% of documents appear only in a single rank list out of the full set of 157 rank lists, meaning that there is a very low overlap of documents across rank lists. This is expected, as the search engines used to produce the rank lists were explicitly selected to produce a diverse set of results. As TREC CDS rank lists were produced from a more

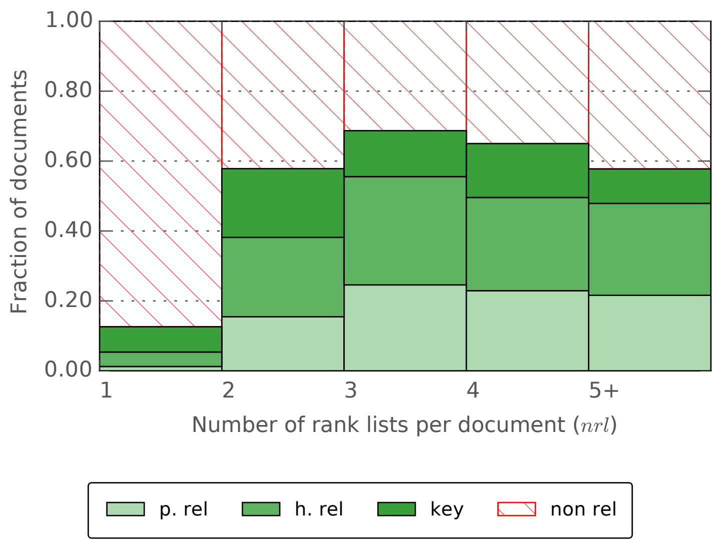


Figure 3.1: Document relevance vs. the number of rank lists for the FedWeb dataset. The X-axis represents the number of rank lists a document appears, nrl. The Y-axis represents the ratios of relevant documents from that rank list. Relevance labels are not relevant (non rel.), partly/potentially relevant (p. rel.), highly relevant (h. rel.) and navigational or key relevant (key)

homogeneous set of features, the overlap of documents across different rank lists is higher: 28.5% appear in 13 or more rank lists out of the 64 rank lists, although only 1.3% of the documents appear on over 52 rank lists.

The potential impact of nrl on retrieval performance becomes clear when measured together with relevance. Figures 3.1 and 3.2 shows how relevance varies with the nrl. For example, on Figure 3.1, over 80% of documents that appear on a single rank list are "not relevant", 10% have "key relevance", and the remaining 6% are "highly relevant".

For FedWeb data, Figure 3.1, the correlation between frequency and relevance is clear: over 50% of documents that show up on one rank list are relevant. The growth in relevance with nrl seems to plateau for documents that are present on more rank lists: the percentage of relevant documents remains between 55% and 70% as nrl increases. CDS data, Figure 3.2, also shows an increase of relevance with nrl is also clear: 0.7% of documents that appear only on a single rank list are relevant, while 25% of the documents that are present on 52 or more rank lists are relevant. On CDS data, relevance keeps growing with nrl (it does not plateau). Annex B provides an extended discussion on how different functions (e.g., linear, exponential, logarithmic) fit the growth with relevance with nrl. These experiments show that document frequency appears to be positively correlated with relevance. CombMNZ explicitly models this effect by giving a linear boost with nrl. However, the growth in relevance does not follow a linear pattern. The question becomes, *is there a function that can model this effect better?*

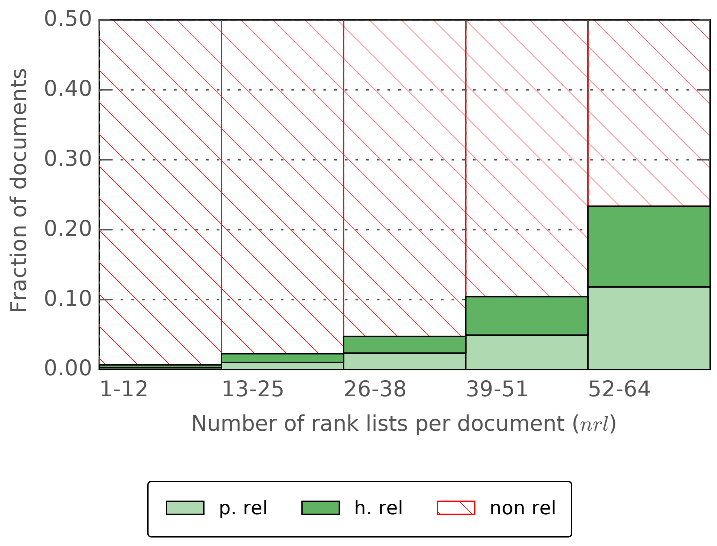


Figure 3.2: Document relevance vs. the number of rank lists for the CDS dataset. Axis are similar to Figure's 3.1. Relevance labels are not relevant (non rel.), partly/potentially relevant (p. rel.), highly relevant (h. rel.)

3.1.2 Inverse Square Ranking

Search engines are designed under the assumption that users will only inspect the top-ranked results. Differences in the ranking of documents towards the end of the rank list start to lose significance. By boosting top-ranked documents score and multiplying it by the nrl, the goal is to guarantee that documents that are highly ranked (higher probability of relevance to the query) and appear on multiple rank lists (to exclude documents from non-relevant engines) are ranked on top of the final rank list. ISR combines the inverse rank approach, of RR and RRF, Section 2.5.2, using the inverse of the rank as the score, with a linear nrl boost. The Inverse Square Rank fusion is defined as,

$$\text{ISR}(i) = nrl(i) \times \sum_{k=1}^{nrl(i)} \frac{1}{R_k(i)^2}, \tag{3.1}$$

where nrl is the number of times document i appears on a results list (document frequency), and $R_k(i)$ is the rank of document i on the kth rank list. Although the previous section showed that the exponential decay could over-penalize documents with low rankings on some approaches, the goal is to see if it can improve precision at the top positions of the rank list.

ISR boosts document score using the absolute nrl. The previous section shows that linear nrl boost over-emphasizes documents present on multiple rank lists. Logarithmic boosting was selected to better model nrl, LogNISR:

$$\text{LogNISR}(i) = log(nrl(i) + \sigma) \times \sum_{k=1}^{nrl(i)} \frac{1}{R_k(i)^2}, \tag{3.2}$$

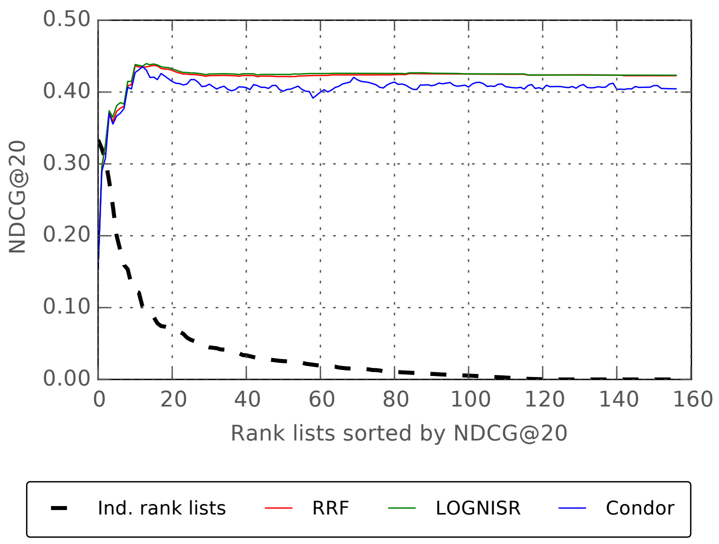


Figure 3.3: Rank list performance from best to worst by NDCG@20 for FedWeb data.

where σ is a regularization factor that sets the "base" boost for documents that appear on a single rank list.

3.2 Selecting which rank lists to fuse

The previous section showed how rank fusion techniques boost document scores according to *nrl*. However, rank fusion algorithms are also affected by the retrieval performance of the different rank lists. Figures 3.3 and 3.4 show how rank lists performance affects rank fusion retrieval performance. The X-axis is the represents the number of ranked lists combined for fusion techniques series; for the Ind. Rank Lists series the X-axis represents the relative ranking of rank lists: best-performing rank lists are on the left. The Y-axis represents the individual (Ind. Rank Lists) or combined rank list NDCG@20. For example, the best rank list on FedWeb data (the leftmost point on the dashed line on the X-axis) had an NDCG@20 of 0.33; the second best had an NDCG@20 of 0.32; the worse performing rank list (the rightmost point on the dashed line on the X-axis) obtained an NDCG@20 of 0.00. The full performance results for individual rank lists are available in Annex A. The remaining lines represent the combination of rank lists using multiple fusion algorithms (RRF and variants, LogNISR and Condorfuse). These charts illustrate one of the key differences between the CDS and FedWeb task: differences in NDCG@20 for CDS rank lists are small (low variability), while FedWeb rank list retrieval performance varies greatly (high variability).

On the FedWeb dataset, fusion performance is similar across the tested rank fusion algorithms. On CDS data, ISR and RRF have comparable performance, while Condor is more affected by adding lower performing rank lists.

This experiment shows that rank fusion algorithms can deliver better results than the

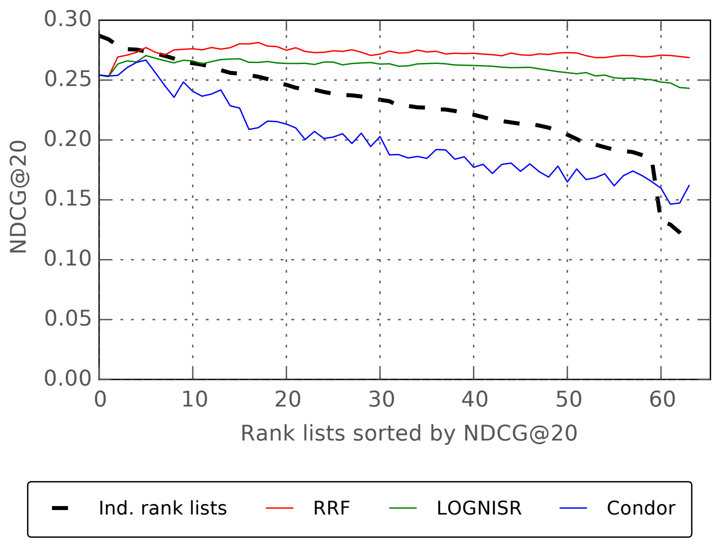


Figure 3.4: Rank list performance from best to worst by NDCG@20 for CDS data.

best individual rank list, but their performance degrades when adding low performing rank lists. Thus, the goal is to find when to stop adding rank lists for fusion for optimal retrieval performance. In other words, find the set of rank lists that is closer to the maximum of the retrieval curves of Figures 3.3 and 3.4.

Consider a corpus of documents D, a set of queries Q with the corresponding definitive (i.e., best possible) rank list r_d (i.e., lists of documents sorted by relevance) and a set of rank lists R, generated using multiple techniques and document features. The assumption is that the best possible rank list is the one that maximizes the desired retrieval metric, based on expert relevance judgments. E is an evaluation metric (e.g., Precision or Normalized Discounted Cumulative Gain, NDCG) that quantifies retrieval quality. The goal of fusion techniques is to find the rank list r_d that maximizes the following:

$$argmax_{r_d}(E(r_d,Q)) \tag{3.3}$$

Finding an adequate rank list r_d is hard when little training data is available. Machine learning based algorithms such as LETOR methods can overfit training data and generate models that are not adequate for new queries. Consider a rank fusion technique F (e.g., RRF) that combines a set of rank lists into a single rank list. The goal is to estimate it as a combination of rank lists: find the subset of rank lists $R_c = [r_1, r_2, ..., r_n]$, which maximizes the expression,

$$argmax_{R_c}(E(F(R_c),Q)), \tag{3.4}$$

where F is a fusion function such as RRF. Rank lists produced using different features will generate different sets of potentially relevant documents. Adding more rank lists for fusion can promote existing or add new relevant documents to the final rank list. The goal is to find which rank list $r_i \in R - R_c$ causes the largest increase in retrieval performance

Algorithm 1 Learning to fuse (L2F) algorithm

Input: R: list of rank lists, sorted in desc. order according to an evaluation metric E,
Input: E: evaluation metric that takes a rank list and a set of relevance judgments and returns the value of that evaluation metric for that rank list,
Input: F: rank fusion function (e.g., RRF, ISR) that takes a set of rank lists and combines them into a single rank list,
Output: R_c: final combined rank list.

```
1:  $R_c \leftarrow R_0$
2:  $\hat{R_c} \leftarrow R_c$
3:  $R \leftarrow R$ - $R_0$
4:  **while** $len(R) > 0$ **do**
5:      $bestR \leftarrow R_0$
6:      $i \leftarrow 1$
7:      **while** $i < len(R)$ **do**
8:          **if** $E(F(\hat{R_c}, R_i)) > E(F(\hat{R_c}, bestR))$ **then**
9:              $iterR \leftarrow R_i$
10:         $i \leftarrow i + 1$
11:     $\hat{R_c} \leftarrow F(\hat{R_c}, bestR)$
12:     **if** $E(\hat{R_c}) > E(R_c)$ **then**
13:         $R_c \leftarrow \hat{R_c}$
14:     $R \leftarrow R - bestR$
15: **return** $R_c$
```

on a metric E.

$$argmax_{r_i}(E(F([R_c, r_i]), Q) - E(R_c, Q)) \tag{3.5}$$

This approach was inspired by unsupervised rank fusion techniques that follow the effects described in [102]: promote documents that are highly ranked (skimming effect) and show up in more than one rank list (chorus effect). Rank fusion algorithms like RR and RRF give fusion scores to documents by summing the inverse of the document rank in the individual rank lists. The chorus effect is visible in CombMNZ and LogNISR: document score is multiplied by the number of rank lists where the document is present.

3.2.1 Learning to Fuse

Following the ideas described in the previous sections, unsupervised rank fusion methods were extended by adding a supervised step. It selects which rank lists to combine, based on their individual retrieval performance on few training query examples. Learning to Fuse's (L2F) training process sorts rank lists by performance and iteratively adds rank lists for fusion while the final result improves, Equation 3.5. Algorithm 1 details the L2F process. At each iteration, it finds the rank list $bestR \in R$ that maximizes the metric E, when combined with the tentative set of selected rank lists $\hat{R_c}$. After an iteration, if the tentative combined rank list $\hat{R_c}$ is better than the best current rank list R_c, $bestR$ is added to R_c. If not, R_c is not updated, and the $bestR$ is removed from the rank list pool R.

This process is repeated until all rank lists are tested. The iterative process is based on the previous section's findings: not adding rank lists that degrade performance increases efficiency and can improve effectiveness. It is important to note that the rank lists R are inspected according to their individual performance on the metric E. This allows L2F to return the best possible rank list on extreme scenarios such as selecting a single rank list when it achieves the best retrieval performance.

Algorithm 1 returns the set of rank lists R_c that maximizes metric E on training queries. To apply this model to new queries, one computes a set of rank lists for the new queries $\hat{R}_c$, using the same techniques as with R_c.

L2F shares some of the properties with LETOR algorithms, but also have crucial differences: it does not assign weights to rank lists and can work better than LETOR approaches on scenarios where less training data is available. Another L2F advantage is that it limits the rank lists to the ones that provide real improvements. Thus, L2F models can be more efficient than LETOR models, by working with a smaller subset of rank lists. Another important advantage is that L2F techniques do not require explicit scores, avoiding feature score normalization issues.

3.3 Evaluation

Experiments were divided into two sections that correspond to the techniques proposed in this chapter:

- *Unsupervised rank fusion*: evaluate the retrieval effectiveness without training data (i.e., manual relevance judgments);
- *Learning to Fuse*: find the balance between improving in retrieval effectiveness and adding rank lists for fusion (reduce efficiency).

3.3.1 Unsupervised rank fusion

When working on a novel retrieval task, one cannot measure which features generate the best rank list for that specific task. A possible solution is the application of unsupervised rank fusion approaches: combine multiple rank lists with unknown performance, and combine them into a single rank list. The goal is to generate a rank list which is close to the best possible rank list.

This section assesses the performance of unsupervised rank fusion approaches on a varied set of datasets, features and retrieval methods. Retrieval performance for the individual features for all datasets is available in Annex A.

The fusion process for all datasets is similar: for each query, a set of rank lists is produced using multiple features and retrieval methods. Documents present across rank lists are grouped by a unique document id, and their final scores are computed using multiple unsupervised rank fusion methods. To address score distribution issues as described by Montague and Aslam [75], rank list scores are normalized by subtracting the mean score and dividing by the standard deviation of the scores on the rank lists. Rank lists are limited to 1000 documents. When documents obtain the same final score, the ordering on the final list is stable, meaning that running the same fusion algorithm with the same data will return the same results.

Multimodal fusion: The goal of this experiment is to assess the performance of fusion techniques on the fusion of rank lists produced from multimodal queries and documents (text and images), Image CLEF Med dataset, Section 2.6.2 Queries are similar to the medical retrieval use-case presented in Figure 1.2, chapter 1. The dataset is divided into two tasks:

Ad-hoc image retrieval: Retrieve images that suit the query, for a total of 35 queries. Queries consist of 1 to 7 sample images and a short text description of the medical diagnosis.

Case-based retrieval: Retrieve relevant articles from a subset of the PubMed Central collection. Queries consist of a case description (with patient demographics, limited symptoms and test results including imaging exams).

The rank lists used on this experiment were produced using HSV histograms (216 dimensions) for the image queries and by searching a textual index based on Apache Lucene[1] with domain-specific features (query expansion and pseudo-relevance feedback).

Results: Table 3.4 contains the results for the multimodal fusion for the ImageCLEF image retrieval and case-based retrieval task task.

Rank-based methods such as ISR and RRF have the best retrieval performance on both tasks. In the image retrieval task, LOGNISR outperformed other rank-based approaches on all metrics. On the case-based retrieval task, ISR is the better method on all the tested most metrics. Score based approaches do not perform as well as rank-based approaches on both tasks, even after the normalization scores in the rank lists for fusion. CondorFuse's performance was also low due to the limited number of rank lists (two) for the voting scheme.

Federated web search: The goal of this experiment is to test rank fusion approaches on TREC FW dataset, Section 2.6.5. Table 3.5 contains the fusion results. The TREC Federated Web Greatest Hits[2] dataset [34] is a collection of vertical search engines rank lists, designed for the TREC FedWeb 2013 and 2014 tasks. As the TREC FedWeb is based on the rank lists provided by different commercial search engines, the document scores are not available. Thus, since score-based algorithms require scores, the RRF score corresponding to their inverse rank list was assigned to each document.

These results show a similar pattern to the ImageCLEF experiments: LOGNISR and RRF based techniques achieve the best performance. ISR linear *nrl* boost is penalized by the larger amount of rank lists. LOGNISR logarithmic *nrl* boost results in better retrieval performance. Score-based approaches performance is closer to rank-based approaches, but are still worse than rank-based approaches for most metrics.

[1]https://lucene.apache.org

[2]https://fedwebgh.intec.ugent.be/

Clinical decision support: The goal of this experiment is to test rank fusion approaches on TREC CDS dataset, Section 2.6.1. The TREC Clinical Decision Support (CDS) 2014 and 2015 datasets [87, 91] is the Open Access Subset of PubMed Central (PMC). PMC is a database of open access full-text biomedical articles. Rank lists were generated using the combination of various retrieval functions and query expansion techniques.

Table 3.6 contains the fusion results. On this experiment, score-based techniques achieved performance comparable to rank-based approaches. The reason is that rank lists are created using similar techniques, having more similar score distributions.

Table 3.4: ImageCLEF multimodal unsupervised fusion results.

	ImageCLEF image ret.				ImageCLEF case ret.			
Fusion	MAP	NDCG@20	P@10	P@30	MAP	NDCG@20	P@10	P@30
RRF	0.1508	0.2473	0.2171	0.1543	0.1505	0.2400	0.1600	0.1248
ISR	0.1458	0.2361	0.2057	0.1476	**0.1608**	**0.2468**	**0.1800**	**0.1257**
LogNISR	**0.1511**	**0.2558**	**0.2286**	**0.1590**	0.1443	0.2271	0.1429	0.1162
CombSUM	0.1394	0.2458	0.2000	0.1524	0.1232	0.2101	0.1571	0.0762
CombMNZ	0.1172	0.2125	0.2000	0.1533	0.1002	0.1737	0.1200	0.0790
Condor	0.0214	0.0636	0.0486	0.0371	0.0590	0.1302	0.0886	0.0448

Table 3.5: TREC FW unsupervised fusion results.

	TREC FW 13				TREC FW 14			
Fusion	MAP	NDCG@20	P@10	P@30	MAP	NDCG@20	P@10	P@30
RRF	**0.3304**	**0.4961**	**0.7300**	**0.4840**	0.2284	0.3486	**0.5460**	**0.3273**
ISR	0.2420	0.3245	0.5320	0.2707	0.1868	0.2271	0.3720	0.2073
LogNISR	0.3214	0.4871	0.6800	**0.4840**	0.2287	**0.3575**	0.5320	**0.3273**
CombSUM	0.2876	0.4166	0.6660	0.3713	0.2129	0.2799	0.4720	0.2587
CombMNZ	0.3164	0.4668	0.7160	0.4347	**0.2343**	0.3360	0.5280	0.3107
Condor	0.3174	0.4798	**0.7300**	0.4447	0.2234	0.3274	0.5320	0.2973

Table 3.6: TREC CDS unsupervised fusion results.

	TREC CDS 14				TREC CDS 15			
Fusion	MAP	NDCG@20	P@10	P@30	MAP	NDCG@20	P@10	P@30
RRF	**0.1392**	**0.2699**	0.3300	**0.2756**	**0.1331**	**0.2677**	0.3533	**0.3289**
ISR	0.1283	0.2522	0.3100	0.2611	0.1240	0.2413	0.3433	0.2889
LogNISR	0.1259	0.2490	0.3033	0.2589	0.1213	0.2373	0.3367	0.2856
CombSUM	0.1354	0.2694	**0.3367**	0.2678	0.1284	0.2596	**0.3767**	0.3167
CombMNZ	0.1358	0.2672	0.3300	0.2711	0.1280	0.2591	**0.3767**	0.3200
Condor	0.0656	0.1665	0.2033	0.1489	0.0566	0.1574	0.2533	0.1533

Conclusions: These experiments show that unsupervised approaches can offer comparable or better retrieval performance than the best individual rank list, without any prior knowledge about individual rank list performance.

The next section shows how analyzing the relevance judgments enable selecting only rank lists that help to create the best possible rank list.

3.3.2 Learning to Fuse

LETOR and L2F fusion performance was evaluated on two types of settings: (1) with rank lists showing *low-variability* in retrieval performance and *high-overlap* in documents (*TREC CDS*); and (2) rank lists showing *high-variability* in retrieval performance and *low-overlap* in documents across rank lists (*TREC FedWeb*).

Baselines: Baselines include the LETOR methods, AdaRank [107], Random Forests [20], LambdaMART [105] and Coordinate Ascent [70], as implemented in RankLib[3]. For L2F, the impact of the rank fusion techniques (e.g., LogNISR, RRF [30], and Condorfuse [76]) on L2F retrieval performance was also measured.

Protocol: Each dataset was divided into train and test splits. For the TREC CDS dataset, methods are trained and validated on 2014 queries and tested on 2015 queries (30 queries each), and vice-versa. Conversely, for the FedWeb dataset, methods are trained and validated on 2013 queries and tested on 2014 queries (50 queries each), and vice-versa. Final results are then averaged over both splits. Fused rank lists are limited to 1000 documents. Both L2F and LETOR methods used rank lists as the input features and were trained using the same train and test splits.

Rank fusion techniques were optimized for NDCG@20 and evaluated the produced rank lists using NDCG@20 and P@10. Valizadegan et al. [99] showed that NDCG provides the most stable fusion retrieval results. The evaluation metrics were computed using the *trec_eval* script.

3.3.2.1 Rank fusion experiment

Table 3.7 contains the results for the TREC FedWeb and CDS fusion experiments. Baseline results include the best single rank list on the training data (L2F-1 baseline), corresponding to the first iteration of the L2F algorithm and the L2F algorithm with no stopping criteria, thus, with the highest computational complexity (L2F-NoStop baseline). These two baselines correspond to the lower-bound and upper-bound of the L2F algorithm.

On the TREC FedWeb data, the best results were achieved by L2F-RRF, with the best NDCG@20 of 0.400 and second best P@10 (a statistically significant improvement over the L2F-1 baseline). The retrieval performance differences between L2F methods show that RRF is better at modeling the decay in relevance with rank list position. Learning to rank methods achieved good results, but still lower than L2F RRF, at a higher computational

[3]`https://sourceforge.net/p/lemur/wiki/RankLib/`

Table 3.7: Fusion results for TREC FedWeb and CDS. Metric under optimization was NDCG@20, "Fusion" is the fusion method used, "#rank lists" is the ratio of selected rank lists/total rank lists. The remaining columns are retrieval results.

	TREC FedWeb			TREC CDS		
Retrieval model	# rank lists	NDCG@20	P@10	# rank lists	NDCG@20	P@10
Baseline						
L2F-1	1/157	0.153	0.446	1/64	0.254	0.347
L2F-NoStop RRF	157/157	0.423	0.644	64/64	0.269	0.342
Learning to rank						
AdaRank	14/157	0.386	0.608	5/64	0.241	0.320
Coordinate Ascent	157/157	0.362	0.574	64/64	0.245	0.317
Random Forests	113/157	0.303	0.474	64/64	0.274	0.363
LambMART	25/157	0.301	0.482	50/64	0.233	0.315
Learning to fuse						
L2F-RRF	**6/157**	**0.400**	**0.601**	3/64	0.263	0.342
L2F-LogNISR	6/157	0.381	0.593	2/64	0.246	0.335
L2F-CondorFuse	6/157	0.342	0.506	**2/64**	**0.268**	**0.372**

cost. AdaRank achieved the second best NDCG@20 and best P@10, while Random Forests and LambdaMart were significantly better than the baseline on the NDCG@20 metric.

The results for TREC CDS data show a different pattern. The Random Forests model achieved the best NDCG@20 (but not statistically significant), while L2F-CondorFuse obtained the best P@10 corresponding to a statistically significant improvement of the L2F-1. Combined with L2F-RRF, these approaches got better performance than the best single run from training data. The smaller variance in retrieval performance in this task is due to the smaller differences in performance in the original rank lists.

These experiments show that finding the best rank list on datasets with few query examples and different document distribution across rank lists is a non-trivial task. For example, AdaRank achieved the second best results for the FedWeb dataset and poor results on the CDS dataset. On the other hand, L2F using RRF was the most consistent method across all metrics and datasets. Also, it is interesting to note that for both datasets, L2F achieved equal or better results with far fewer rank lists than the LETOR baselines. The next section discusses this issue in depth.

3.3.2.2 Influence of the number of rank lists

Figures 3.5 and 3.6 illustrates how the number of rank lists for fusion affects retrieval performance. Methods that combine fewer rank lists are on the left side of the graph, and thus, have a lower computational complexity (they require the computation of fewer rank lists per query). Using fewer rank lists reduces the complexity as fewer features are needed to compute the final search results. Thus, this is an important aspect of every search engine. L2F-RRF retrieval performance is based on the fusion of six rank lists on

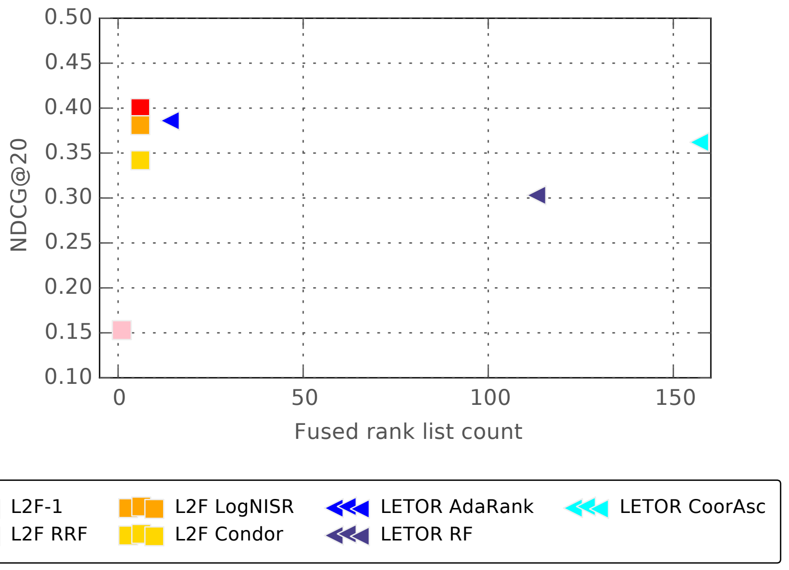


Figure 3.5: Retrieval performance versus number of rank lists fused for TREC FedWeb. The X-axis is the represents the number of ranked lists combined. The Y-axis represents the combined rank list NDCG@20.

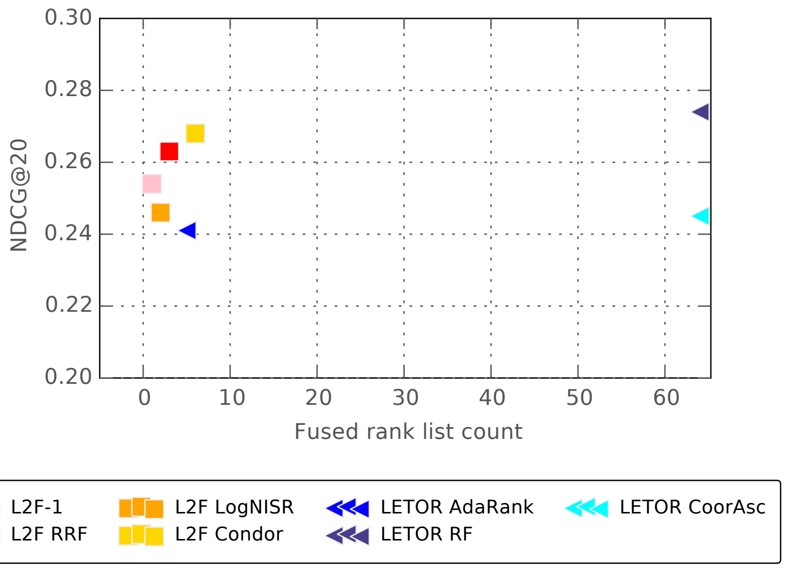


Figure 3.6: Retrieval performance versus number of rank lists fused for TREC CDS.

the TREC FedWeb and two rank lists on the TREC CDS (less than 5% and 3% of rank lists respectively). In contrast, LETOR approaches added more rank lists (the full set in the case of Coordinate Ascent and RandomForests on the FedWeb experiment), which not only increases the complexity without a corresponding increase in retrieval precision. On the TREC CDS experiment, Random Forest achieved an improvement in NDCG@20 over L2F-1, but it used all 64 rank lists, and the difference is not statistically significant. Other LETOR approaches achieved worse results while using more features for fusion. However, the surprising result is that L2F-Condorcet achieved the best P@10 with only two rank lists. L2F turned out to be the most economical model regarding complexity and the best model in retrieval precision.

3.4 Conclusion

This chapter described ISR rank fusion and the L2F rank selection method. ISR fusion is based on the quadratic decay of document scores and logarithmic document-frequency normalization, which is closer to the distribution of relevant documents, as seen in section 3.1.2. Highly ranked documents and document-frequencies across different lists are the most important qualities of documents in rank-based fusion. The ISR algorithm shares the simplicity of RRF with improved retrieval performance through logarithmic document frequency normalization with LogNISR. In the unsupervised retrieval experiments, ISR-based techniques were the most balanced solutions, in line the best existing algorithms (RRF) on most metrics.

L2F is a greedy supervised rank list selection and fusion algorithm. The proposed method works better than LETOR baselines in scenarios with limited training data, achieving good results using a small number of rank lists for fusion. The evaluation process showed that L2F achieves equal or better results than LETOR, using fewer rank lists than LETOR (3% and 5% of all rank lists for the TREC CDS and TREC FedWeb datasets respectively). It also shows how dataset properties such as overlap and performance variability affect both L2F and LETOR approaches.

Chapter 4

Multimedia vertical partitioning indexes

Multimedia search systems must deal with the challenges of managing multimodal documents and queries at increasingly larger scales. These challenges include how to combine the search results from a heterogeneous set of features while dealing with scales that go beyond the computational capabilities of individual nodes. Existing research the distribution of multimedia indexing is scarce; experiments are performed at either small scales [108] or are evaluated only on a single feature space [74]. This chapter addresses the key efficiency aspects in the deployment of a distributed search architecture, capable of handling several millions of multimedia documents. This chapter proposes the MVP architecture, based on feature-level vertical partitioning. It was designed to:

- simplify the distribution of documents and queries across nodes;
- work independently of the indexing algorithms used to index individual modalities;
- combine results from several heterogeneous sources.

To leverage the distributed nature of the search architecture, individual feature indexes are tested using an inverse index structure based on sparse hashes, Sparse Hash Index (SHI). A comprehensive evaluation of both general retrieval effectiveness metrics and efficiency metrics provides a unique assessment of the several efficiency bottlenecks faced by a search engine. The scalability of the search architecture was evaluated on multiple index sizes, i.e., up to 100 million documents per processing node. In summary, the contributions of this chapter are twofold:

- the formalization and deployment of a scalable vertically partitioned distributed architecture, MVP, capable of handling heterogeneous large-scale search, with minimal distribution overhead;

- a comparison of the scalability of SHI compared with state-of-the-art techniques on a large scale evaluation using datasets with hundreds of millions of feature vectors.

4.1 An architecture for vertical index distribution

The MVP architecture is based on two types of nodes: processor and aggregator nodes. *Processor nodes* which index documents and answer queries for individual feature spaces; *aggregator nodes* which serve as the architecture's entry points, routing queries to relevant processor nodes and combining results from searching those nodes.

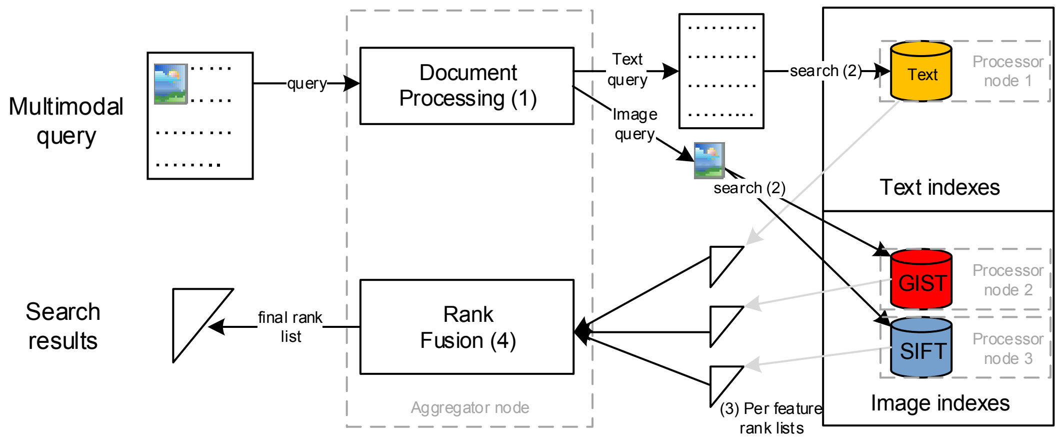


Figure 4.1: Multimodal Vertical Partitioning architecture

Figure 4.1 shows the full search process. Documents are parsed by the aggregator node and split by media type (e.g., text or images), step 1. For each media type, the aggregator node makes a search request to each relevant processor node, step 2. After the search, the aggregator collects and combines the rank lists, steps 3 and 4. Communication between nodes is made through REST requests using JSON to represent indexing and query requests.

4.1.1 Vertical distribution

An important challenge for large-scale search systems concerns the distribution of documents across nodes for indexing. On textual indexes, vertical partitioning is performed at term level: document words are distributed per nodes; at query time, nodes containing the query words will be searched. For multimedia feature vectors, the advantages of vertical partitioning for individual feature spaces are smaller, as vector similarity is measured across all dimensions. Thus, MVP vertical partitioning is performed at feature index level, as indexes are independent of each other and can be searched independently. This is also important as different types of media (e.g., text document only vs. text, images, and video) require different types of processing and indexing.

Algorithm 2 lists the full distributed indexing procedure: for each modality in the documents, line 1, the aggregator node sends indexing requests to each relevant processor node (e.g., send images to GIST and SIFT indexes). Each processor node extracts feature vectors and adds them to the index, line 4.

Algorithm 2 Distributed indexing algorithm

Input: document *Doc*, set of indexes *I* grouped by type

```
1: for each data type $t$ in $Doc$ do                    ▷ aggregator node
2:     for each feature index $s$ in $I$ with type $t$ do    ▷ concurrently on proc. nodes
3:         extract features $f$ for $Doc$ on $s$           ▷ optional
4:         add $f$ to index $s$
```

This algorithm works seamlessly for multi-feature image retrieval and multimodal retrieval systems (e.g., video retrieval system with audio, image and textual features). MVP is also flexible in the assignment of indexes to nodes: it works with horizontally partitioned indexes running on multiple nodes when a single feature index does not fit into a single machine. **mvp** is also adequate for indexes with few documents, as it enables the assignment multiple indexes to a single node.

Algorithm 3 lists the distributed retrieval procedure: for each media type in the query, line 2, the aggregator node simultaneously queries each relevant processor node and collects the results. Each processor node extracts feature vectors (if not previously extracted), queries the index, (line 5) and returns the sorted rank list. The aggregator node collects the results and aggregates individual rank lists into a single final rank list (line 7).

Algorithm 3 Distributed retrieval algorithm

Input: query *q*, number of neighbors to retrieve *k*, set of indexes *I* grouped by type
Output: a rank list *Rl* of *k* (nearest) neighbors.

```
1: $C \leftarrow \emptyset$                                  ▷ set of candidate rank lists
2: for each data type $t$ in $q$ do                      ▷ aggregator node
3:     for each feature index $s$ with type $t$ in $I$ do    ▷ concurrently on proc. nodes
4:         extract features $f$ for $q$                   ▷ optional
5:         $c_s \leftarrow$ search $f$ on $s$
6:         add $c_s$ to $C$
7: aggregate rank lists in $C$ into $Rl$                 ▷ aggregator node
8: return $Rl_{1,\dots,k}$
```

The aggregation algorithm can be generalized as a function that takes a set of rank lists and returns a combined rank list. These techniques can be either unsupervised or augmented by determining the relevance or weight of each index to a dataset using learning-to-rank approaches, as presented on Chapter 3.

An advantage of the proposed architecture is its resilience to nodes failure. In multimodal data each document is represented by many different feature vectors: image search engines can easily have more than ten feature vectors [1] per image corresponding to 10 independent indexes. Thus, the aggregator node waits for a set timeout (e.g., set to 100 ms) and, if a node fails to respond to requests, returns the result from the remaining nodes.

4.2 Sparse hashes for single-node indexing and retrieval

The MVP architecture temporal performance is limited by the time spent searching feature indexes. Sparse hashing techniques provide effective retrieval by the usage of a locality-aware sparse hash dictionary: at retrieval time, only the entries with common non-null values on the hash space are inspected. Thus, the search space is drastically reduced during retrieval time. This section recaps sparse hash techniques and details how they can be combined with an inverted indexing for retrieval.

4.2.1 Creating sparse hashes

Sparse hashing techniques transform dense d dimensional feature vectors into a sparse h dimensional vector, where $d << h$ and with s non-zero coefficients, $s << d << h$. In other words, compress the initial signal into a small number of coefficients (much smaller than the original space dimensionality) on a higher dimensional space. Sparse hashing is based on theoretically well-grounded dictionary learning and sparse reconstruction techniques.

Section 2.3 describes how to create sparse hashes (KSVD, [2] and OMP, [84]) and section 2.3.2 describes how inverted index structures can be used to index sparse text documents. Sparse hash indexing was initially proposed by Borges et al. [19], and is extended in this Chapter to vertically distributed indexes.

Consider the original feature vector $y \in \mathbb{R}^d$, a sparse vector $x \in \mathbb{R}^h$ and a sparsity coefficient s. From a set of n, d-dimensional vectors in the original space as $Y \in \mathbb{R}^{n,d}$, the goal is to find a function f that transforms them into h-dimensional sparse vectors as $X \in \mathbb{R}^{n,h}$.

4.2.2 Sparse hash indexing

Sparse hashes have a key property which can be explored for inverted index: vectors that are close in the original space have non-zero coefficients on similar positions in the sparse vector space than vectors that are further apart. This closely mimics an inverted index structure, where each inverted index entry (i.e., posting list) corresponds to a sparse hash position (i.e., h dimensional hashes are mapped to h posting lists): documents are stored in the posting lists that match the hash dimensions where they have non-null coefficients.

The construction of the sparse hash inverted index is illustrated by Figure 4.2. The indexing construction process is the following: given a set of feature vectors Y, recover their sparse solutions X This sparse representation set X determines the non-null coefficients to match hashes to posting lists of the index I. The final step is to sort each posting list by placing the hashes with higher coefficients at the beginning. This final sort guarantees

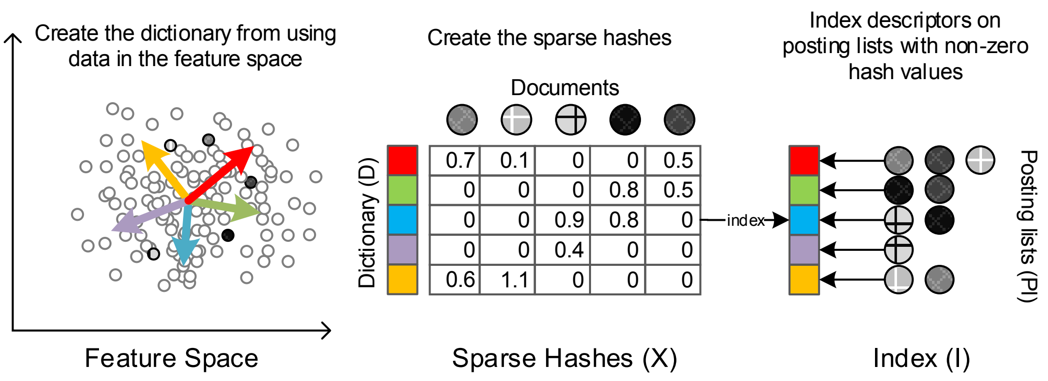


Figure 4.2: Sparse hash computation and indexing

Algorithm 4 Intra-node indexing algorithm (processing node)

Input: the query vector y and the set of posting lists Pl.

```
1: $x \leftarrow \text{sparseHashing}(y)$
2: for $Pl_b$ in $Pl$ do                                        ▷ in parallel
3:     $k \leftarrow binarySearch(Pl_b, x_b)$                  ▷ find closest coeff.
4:     $Pl_b \leftarrow ..., Pl_{b,k}, x_b, Pl_{b,k+1}, ...$        ▷ insert
```

Algorithm 5 Intra-node retrieval (processing node)

Input: the query vector y, the set of posting lists Pl, the number of neighbors to retrieve k and the number of candidates to examine per posting list nc.
Output: a list of k (nearest) neighbors C.

```
1: $C \leftarrow \emptyset$
2: $x \leftarrow \text{sparseHashing}(y)$
3: for $x_l$ in $x$ do                                          ▷ in parallel
4:     $j \leftarrow 0$
5:     while $|C| < nc \cap j < |Pl_l|$ do
6:         $C \leftarrow C \cup \{\|y, Pl_{l,j}\|_2, Pl_{l,j}\}$
7:         $j \leftarrow j + 1$
8: $C \leftarrow sortL2(C)$                                     ▷ Perform candidate sort
9: return $C_{1,...,k}$
```

that, at query time, the most representative document feature vectors of each posting list are examined first.

The intra node document indexing is described in Algorithm 4. After the partitioning process, Equation 5.3, documents are added to one or more posting lists in position, lines 3 and 4, to keep posting lists sorted in descending order by hash coefficient.

4.2.3 Sparse hash retrieval

Algorithm 5 details the processing node candidate selection process. For each posting list Pl that matches the non-null hash positions x_l, SHI collects and re-rank a set of candidates, lines 5 to 7. Candidates are sorted, line 8 and the k ones with the lowest distance on the original feature vector space are returned to the coordinator node. It exploits the fact that the tested sparse coding algorithm (OMP) is greedy and select the most correlated atom in the dictionary first.

4.3 Experiments

A set of experiments was performed to evaluate all the components of the architecture:

- *Multimodal image and text retrieval*: evaluate aggregation efficiency on a multimodal task, using multimodal (textual and image) documents (Section 4.3.2);
- *Single node retrieval*: evaluate the retrieval performance on an individual node (Sections 4.3.4 and 4.3.6);
- *Distributed multi-feature retrieval*: evaluate the retrieval performance using multiple feature types on a distributed setting (Section 4.3.5).

4.3.1 Experimental setup

Experiments were run on a set of five nodes, all running Ubuntu 14.04: four processor nodes with an i7-3930K processor and 64GB of RAM and one aggregator node with i7-920 with 12GB of RAM. All feature vectors, indexes, and other structures were placed in the main memory.

4.3.1.1 Datasets

The datasets tested were: The **ImageCLEF Medical** dataset, section 2.6.2, composed of 70,000 biomedical articles with the corresponding 300,000 images. The tested system was a part of our submission to ImageCLEF Medical 2013, described in detail in [77]. Textual retrieval is based on Apache Lucene with pseudo relevance feedback (PRF) query expansion; image retrieval is based on segmented histogram features based on the HSV color space and CEDD [25] features. The **Tiny Images** dataset, section 2.6.3 contains a set of 79,302,017 GIST feature vectors with 384 dimensions. The **ANN** dataset, section 2.6.4 is composed of a set of SIFT (128 dimensions) and GIST (960 dimensions) feature vectors. It is divided into three subsets: two one **million** vectors sets with both SIFT and GIST features (used in sections 4.3.2 to 4.3.5) and a one **billion** SIFT vector set (used in section 4.3.6).

4.3.1.2 Metrics

The efficiency and effectiveness of the proposed method were measured using metrics widely reported in the retrieval architecture [42, 96, 97]: the average of the percentage of the true 50 nearest neighbors that are present in the top 50 positions, **avgP@50** and average **query time**. Experiments were based on previously extracted feature vectors, as the objective is to test indexing and retrieval, and extracting feature vectors introduces additional temporal variability across nodes. For each query, the output of each method is compared with the output of the linear exhaustive search system. Hence, the top k nearest neighbors correspond to the relevant items.

4.3.1.3 SHI Parameters

KSVD dictionary learning iterations were limited 25 since from this point onward the average recovery error did not vary significantly ($< 1\%$), Borges et al. [19]. All results reported in this section used ten OMP iterations, which generates a sparse hash with both positive and negative hash coefficients. For the SHI algorithm, the number of candidates to examine was set to {1%, 2%, 3%, 4%, ..., 10%, 20%, 30%, 40%, ..., 100%} of the index size. Hash size was set to 1024.

4.3.2 Multimodal retrieval using the MVP architecture

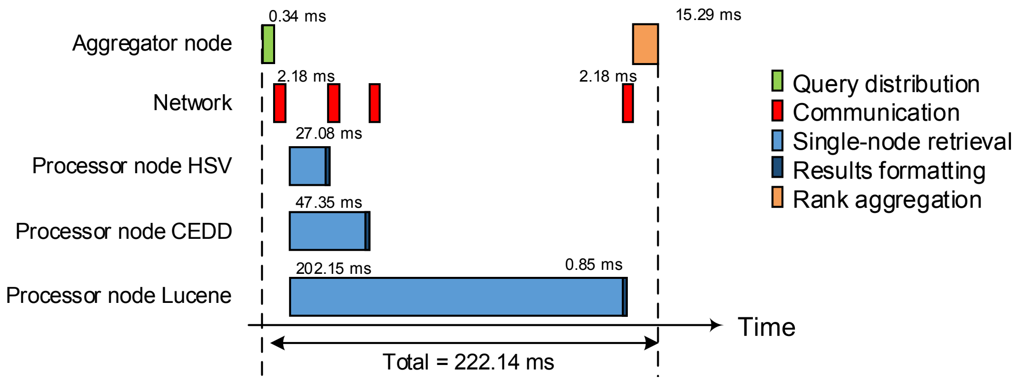


Figure 4.3: MVP overhead versus retrieval time on the ImageCLEFMed 2013 dataset

Figure 4.3 contains the detailed account of the time taken by the individual architecture components, for the SHI indexer using the ImageCLEF dataset. The times for query distribution, communication, formatting, and aggregation processes are average times, determined using a set of 5000 multimodal queries and three individual indexes (Lucene text index with PRF, CEDD and HSV image indexes) with $k = 1000$. On this experiment, the textual index takes the greatest share of retrieval time (over 200 ms) compared to the image index retrieval. This is due to time spend performing multiple index inspections for query expansion (pseudo-relevance feedback). Table 4.1 details how long each node spend on feature extraction and on retrieval. On small scales, the time spent extracting the CEDD feature vector was longer than the retrieval time.

The total overhead of SHI is $\tilde{2}0$ ms, meaning that the total query time will be approximately the retrieval time of the slowest index plus 20 ms. This overhead is small when compared to individual index retrieval time, lowering the cost of using multiple features.

Table 4.1: ImageCLEF Med 2013 efficiency results

Features	Feature extraction	Retrieval	Total time	Retrieval time (ms)
CEDD	27.12	20.23	47.35	
HSV	5.24	21.84	27.08	217.44
Text retrieval	–	202.15	202.15	

4.3.3 Single node comparative benchmark

This experiment shows the impact of changing the size of the hash codes and percentage of the index to inspect on temporal and retrieval performance on the ANN Vector GIST and SIFT one million vector sets. The following set of widely known indexing algorithms was selected as baselines:

- k-means tree (*kMeansTree*), a randomized kd-tree (*kdTree*), by [78], implemented in FLANN[1];
- LSH implementation[2] described in [42], with multiple code sizes {64, 128, 256, 512} bits, Spherical Hashing [42] (with multiple code sizes {64, 128, 256, 512 code sizes, and two vector distance metrics: Hamming distance (SH) and spherical Hamming distance (SH_shd))
- MSIDX method [96] [3], with the number of candidates to examine was set to {0.5%, 1%, 2.5%, 5%, 10%, 15%, 20%, 25%, 30%, 40%, 50%} of the index size.

Figure 4.4, shows two major trends. The first trend occurs in the initial part of the curve, where precision grows very rapidly as query time increases; in the second trend, precision grows linearly towards the maximum precision. These two trends are visible in the MSIDX method and the SHI method. The different slopes indicate that both methods are optimized for efficiently retrieving the top nearest neighbors. As a consequence, after a given point, the curve slope is not as steep, showing a loss in performance as a side-effect of OMP choosing the higher coefficients first.

SHI proposal achieved 49% precision by inspecting 10% of the index and in less than 20% of the linear search time (10% of inspected data corresponds to the tenth point in the SHI curve). Similar trends were observed on experiments with SIFT features, Figure 4.5.

4.3.4 Single-node retrieval efficiency

For the scalability experiments, parameters were set according to their performance on the previous experiment: SHI and MSIDX: number of candidates to examine: 10% of the index size, SH_hd, SH_shd, SH_lsh: 128-bit codes. Figure 4.6 shows the performance when

[1]http://www.cs.ubc.ca/research/flann/
[2]http://sglab.kaist.ac.kr/Spherical_Hashing/
[3]http://vcl.iti.gr/msidx/

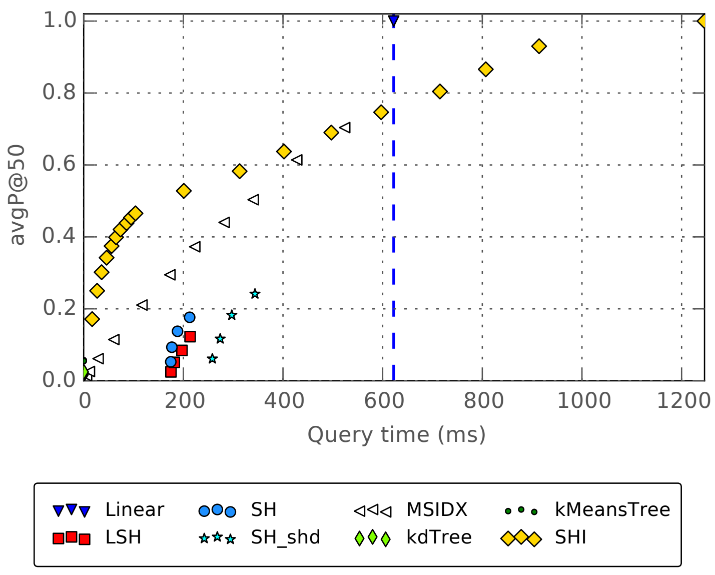


Figure 4.4: Single node retrieval benchmark on the ANN Vector GIST onde million vector dataset. avgP@50 vs. query time with $k = 1000$.

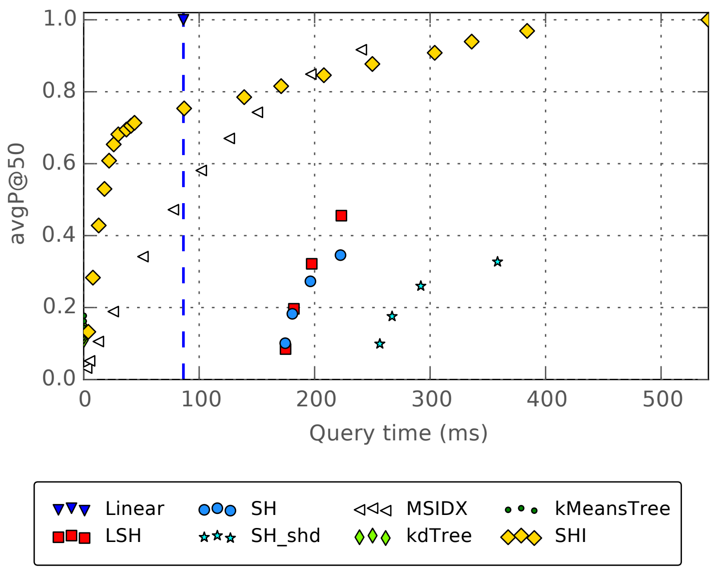


Figure 4.5: Single node retrieval benchmark on the ANN Vector SIFT one million vector dataset. avgP@50 vs. query time with $k = 1000$.

using indexes with variable size and with different dimensionalities. Retrieval performance is stable with index size, but, for SHI and to a smaller extent, MSIDX, avgP@50 increases until the index contains one million documents and then it stabilizes. This effect is opposite on hash-based algorithms, where precision drops with index size. The hypothesis is that the small hashes become less discriminative as the index size grows and that the sparse hashes remain discriminative for all tested index sizes.

The performance differences between features are also clear: the benefits are more clear

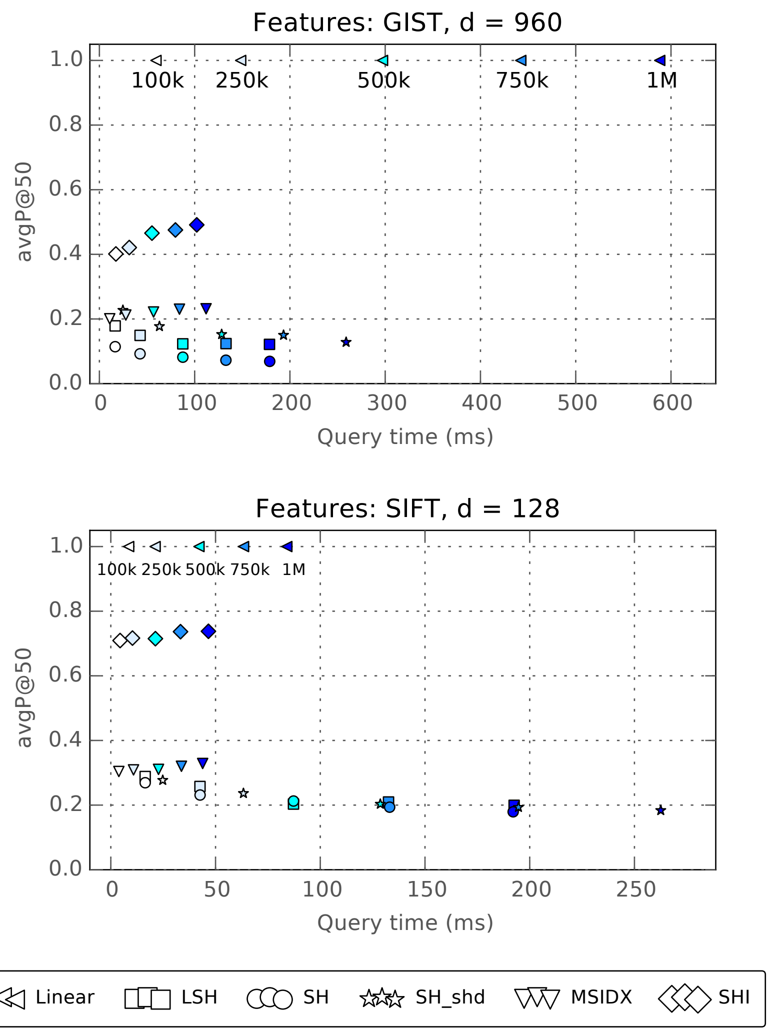


Figure 4.6: Avg. P@50 vs. query time for multiple index sizes. Each color represents a different index size: from lightest to darkest: 100k, 250k, 500k, 750k and 1M.

on the (high dimensional) GIST features: speedup of SHI and MSIDX vs. linear is about 5.2×. All indexers get better precision on the lower dimensional (128) SIFT features, but speedup vs. linear on the faster indexes (SHI and MSIDX) is not as high, about 2×. SHI stands out for both features: it is the faster or one of the fastest indexes on all experiments and achieved the highest precision (73% on SIFT and 43% on GIST).

Figure 4.7 shows the performance on the larger Tiny Images data with GIST features with 384 dimensions, using five million vectors. Index performance is close to ANN GIST dataset experiments: SHI is the fastest and achieves higher precision. Experiments on the larger 2M and 5M feature vector indexes follow the expected trends: SHI and MSIDX precision is stable while other hash-based approaches perform worse as the index size increases. In general, the SHI results compare favorably to the other tested methods. LSH and variants (SH) achieve the slowest results: the retrieval performance improves with the size of the binary codes, but they are still slower than the proposed method.

Table 4.2: Retrieval times on ANN SIFT and GIST one million vector dataset with variable index subset size

		SIFT		GIST		Total
Name	n	Time	avgP@50	Time	avgP@50	Time (ms)
Linear	100k	8.52	1.00	59.38	1.00	80.22
	250k	21.14	1.00	148.28	1.00	169.12
	500k	42.08	1.00	297.20	1.00	318.04
	750k	63.36	1.00	442.32	1.00	463.16
	1M	84.03	1.00	588.17	1.00	609.01
SHI	100k	4.43	0.71	17.32	0.40	38.16
	250k	10.34	0.72	31.35	0.42	52.19
	500k	21.29	0.72	55.18	0.47	76.02
	750k	33.27	0.74	79.80	0.48	100.64
	1M	46.61	0.74	102.20	0.49	123.04
MSIDX	100k	3.84	0.30	10.86	0.20	31.70
	250k	10.84	0.31	27.75	0.21	48.59
	500k	22.77	0.31	56.84	0.22	77.68
	750k	33.72	0.32	84.05	0.23	104.89
	1M	43.85	0.33	111.98	0.23	132.82
LSH	100k	16.41	0.29	16.54	0.18	37.38
	250k	42.57	0.26	42.73	0.15	63.57
	500k	87.22	0.20	87.53	0.12	108.37
	750k	132.59	0.21	132.91	0.12	153.75
	1M	192.56	0.20	178.54	0.12	199.38
SH_hd	100k	16.41	0.27	16.55	0.11	37.39
	250k	42.62	0.23	42.67	0.09	63.51
	500k	87.24	0.21	87.59	0.08	108.43
	750k	133.12	0.19	132.81	0.07	153.65
	1M	192.09	0.18	178.77	0.07	199.61
SH_shd	100k	24.79	0.28	24.70	0.23	45.54
	250k	63.32	0.24	63.02	0.18	83.86
	500k	128.66	0.20	128.33	0.15	149.17
	750k	194.62	0.19	193.29	0.15	214.13
	1M	262.56	0.18	258.98	0.13	279.82

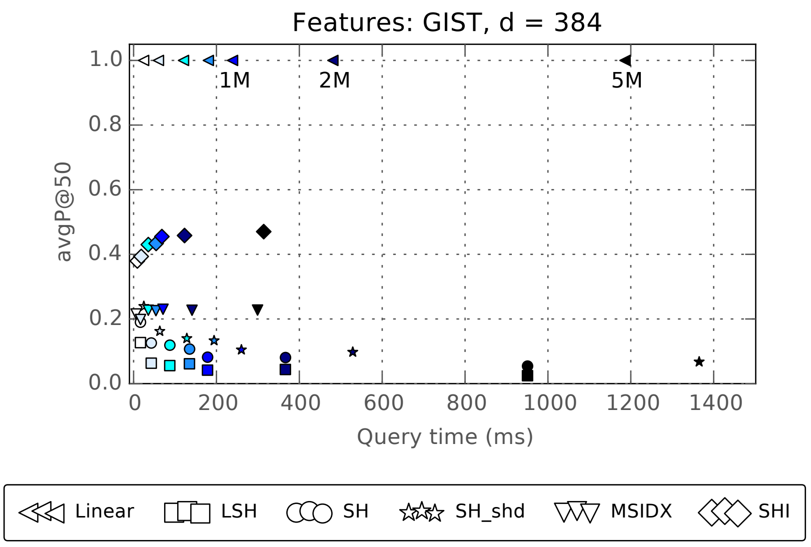


Figure 4.7: Avg. P@50 vs. query time for multiple index sizes. Each color represents a different index size:from lightest to darkest: 100k, 250k, 500k, 750k, 1M, 2M and 5M.

4.3.5 Distributed retrieval efficiency

This experiment studies the performance of MVP in a distributed setting across multiple index sizes and features (aggregator makes the query and aggregates the results from the processor nodes). In this experiment, the aggregation process refers to the combination of the results produced from the two modalities into a single rank list using rank fusion techniques. The datasets tested were ANN Vector SIFT and GIST one million sets.

Table 4.2 contains the precision and temporal values for the full architecture experiments. The n column contains index size, the Time SIFT, and the Time GIST column corresponds to processor node times on Figure 4.6 and the Total Time column correspond to the total query time at the aggregator for the multi-feature querying process. These results follow the same pattern as the scalability results on the previous section. The SHI index continues to offer the best balance between temporal and retrieval performance for all index sizes.

4.3.6 Testing the limits of vertical index partitioning

The final experiment was to test the scalability of the SHI algorithm for the larger ANN Vector SIFT one billion feature set (up to 100 million vector indexes). As the memory requirements of this experiment exceeded the resources available locally, this experiment was performed on a G5 Azure Virtual Machine (Intel Xeon E5 CPU, 2 GHz, 32 cores, 448 GB RAM). Due to the scale of the datasets, SHI retrieval was limited to 0.5%, 1% or 5% of the posting list sizes.

Figure 4.8 shows the performance of the SHI algorithm with multiple limits and the linear exhaustive search. On Figure 4.8 (top), shows the temporal performance vs. size. Retrieval grows sub-linearly for the 0.5% and 1% limit and approximately linearly for the

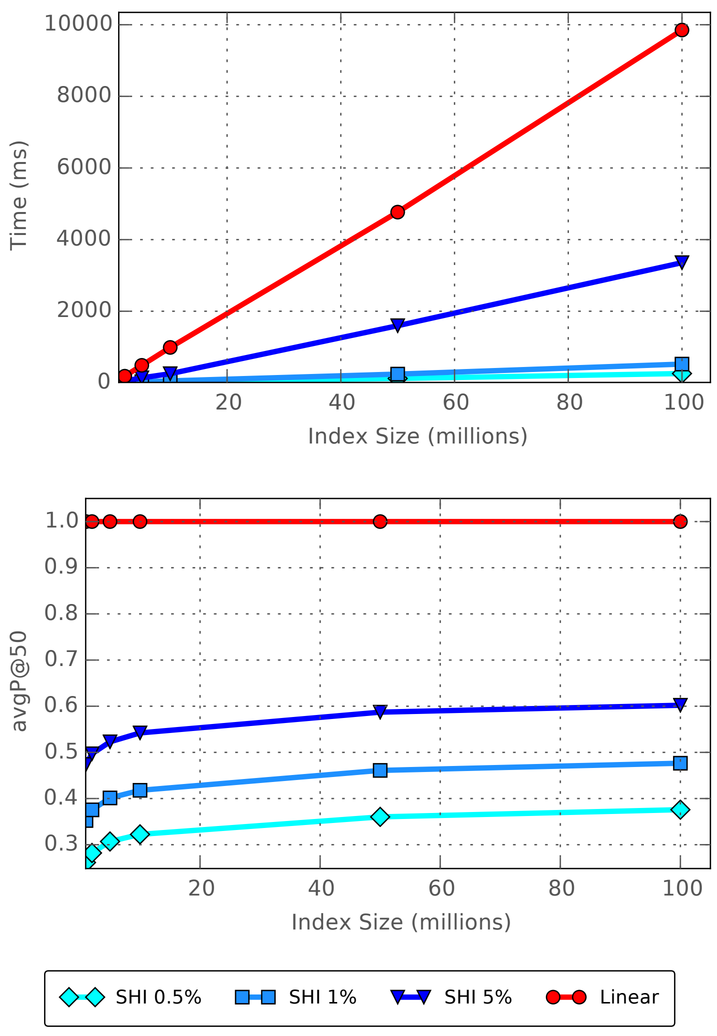


Figure 4.8: Large-scale retrieval comparison: (a) time vs. index size and (b) performance vs. avgP@50 size on subsets of the ANN Vector SIFT one billion vector dataset.

5% limit. As expected, linear search grows linearly with size. On Figure 4.8 (bottom) shows the retrieval performance vs. size. The plot shows a small increase in retrieval performance with index size, for the same inspection limit (about 10% increase from 1M to 100M). The hypothesis is that, as the increase in the number of indexed feature vectors leads to more documents per bucket, the retrieval algorithm will find less duplicated results across top buckets, meaning that the algorithm will add fewer candidates from less relevant buckets. At larger scales, inspecting smaller subsets of the index becomes even more effective; at 100M, examining 5% is over eight times slower than 1%, at the cost of 12% precision.

4.4 Conclusion

This chapter described MVP, an architecture for a large-scale heterogeneous search engine. The proposed architecture is generic, scalable and adds little overhead to the retrieval process. Rank fusion achieves good retrieval performance and is agnostic to the chosen features. The SHI algorithm is capable of sub 100 ms retrieval, scalable to millions of documents per index while offering the best precision results across the tested approaches.

Section 4.3.6's experiment showed that having hundreds of millions of feature vectors on a single node requires expensive nodes with very large amounts of memory (hundreds of GB). Partitioning large indexes to multiple nodes can decrease cloud deployment costs by using more, less expensive nodes. The following chapters describe the partitioning of large, single feature indexes to multiple nodes using sparse hashing techniques.

Chapter 5

Sharding very large-scale multimodal indexes

The previous chapter presented an architecture for the indexing of millions of multimedia documents, by partitioning indexes across nodes. However, the number of documents and the size of single-feature indexes in platforms such as Google Reverse Image Search or Pinterest recommendations [52] goes beyond the processing capabilities of a single processor node.

These platforms require efficient indexing algorithms that can deal with high-dimensional, dense feature vectors at very large scales. However, existing distributed indexes are either based on (i) distributing sub-trees [3, 79], meaning that the number of nodes to search is tied to data dimensionality or (ii) combining existing single-node indexing algorithms with partitioning based on large-scale processing frameworks (e.g., Map-Reduce [33]). Such frameworks are designed to treat all data and nodes equally (i.e., random partitioning, query all nodes for all queries); most do not take advantage of the similarities between the document collection data and queries for partitioning. In addition, they are optimized for batch processing, having high temporal overhead for non-batch query processing [74]. Existing effective single node similarity-based indexing techniques assign documents to only one partition [48] or cannot determine which partitions should be inspected per query without transferring the indexing structure [3, 79]. The question then becomes *can collection-specific knowledge be applied to the index distribution problem?*

A distributed index where partitions are based on the distribution of feature vectors in the original space can improve efficiency, as nearest neighbors are only present on a small number of partitions which reduces the number of nodes to inspect for each query. Chapter 4 showed how sparse hashes create similarity-based index partitions, SHI, by assigning documents and queries that share principal directions to the same posting lists. *Can sparse hashing provide effective horizontal index partitioning by distributing posting lists across nodes?*

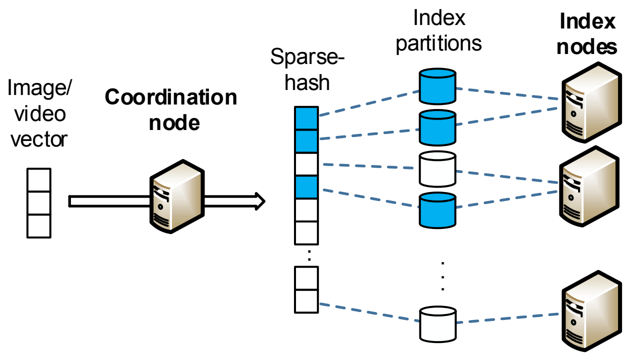


Figure 5.1: DISH architecture

This chapter describes DISH, a distributed k-NN index for cloud-based systems, based on sparse hashes. DISH shows that sparse hashes can balance the computational load between nodes using data-driven distribution policies for high-dimensional data. DISH distributes and balances documents and queries to a subset of the nodes, according to their orthogonal similarities. It explicitly addresses the efficiency and resource constraints of a distributed architecture, by enabling querying only nodes with potentially relevant documents and providing redundant by assigning documents to more than one nodes. The goal is to distribute documents across nodes so that each query is served by a *redundant, but limited* number of nodes. Indexing each document on more than one node enables distributing search over multiple nodes for each query and gives some redundancy guarantees. Not using the full set of nodes for each query is a fundamental property to prevent performance degradation when there are multiple streams of queries being answered simultaneously.

Figure 5.1 illustrates the DISH architecture: It is divided into two types of nodes: coordination nodes and index nodes. **Coordination nodes** receive search query requests, route them to index nodes using sparse hashing and aggregate the retrieved search results. **Index nodes** are responsible for storing index shards. They receive requests from the coordination node to retrieve the candidate documents that exist in their index shard.

The main contribution of this chapter DISH, and its application for distributed media search. DISH's main properties are:

- **Distributed image/video similarity index**: similarity-based document-to-node allocation policy based on theoretically well-grounded sparse hashing algorithms;
- **Robust and redundant index sharding**: each inspected shard brings new relevant candidate documents to the ranking. Also, the index redundancy also contributes towards a graceful performance degradation on node failure;
- **Flexible and balanced resources allocation**: the quantification of to document-partition membership value allows choosing which nodes to query (e.g., do not query nodes with low membership values if the load is too high). This allows supporting the increase in load when answering multiple concurrent query streams.

The second major contribution is the cloud deployment of DISH for large-scale distributed image search. DISH was evaluated on a multi-tenant cloud provider to measure its effectiveness in real-world scenarios. This real-world deployment of the distributed media index enabled a close examination of the behavior of the proposed solution regarding concurrent querying, scalability, and resilience to node failure.

5.1 Distributed indexing and retrieval

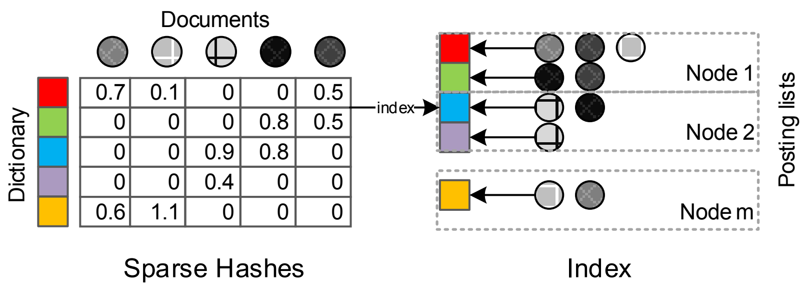


Figure 5.2: DISH index partitioning

DISH was designed to tackle the challenges arising from the distribution of an index of high-dimensional feature vectors to multiple nodes. DISH builds on top of SHI by partitioning posting lists across nodes in a balanced manner. Figure 5.2 provides an overview of how the partitioning is performed: the set of h posting lists generated from overcomplete hashes are partitioned across m nodes so that each node has h/m posting lists. At query time, only nodes containing posting lists with non-null values (i.e., at most, inspect s nodes, where s is the sparisty factor) are inspected. Not using the full set of nodes is a fundamental property to prevent performance degradation when there are multiple streams of queries being served simultaneously. The following sections detail the building blocks behind our distributed index.

5.1.1 Redundant sharding by sparse hashing

Deciding how to partition documents across nodes is a critical challenge for every distributed indexing system. On high-dimensional media retrieval, documents are represented as dense d-dimensional feature vectors. In their original form, dense feature vectors cannot be easily partitioned by similarity, as the similarity must be measured across all the feature vector dimensions. Partitioning these vectors randomly means that documents in a partition are not required to share any degree of similarity. This means that all nodes must be queried to get relevant results. The previous chapter showed that it is possible to transform dense descriptors d dimensional vectors Y into sparse h dimensional vectors X, with s non-zero coefficients, enabling:

Algorithm 6 Distributed indexing algorithm (coordination node)

```
Input: a sparsifying dictionary $D$, a vector to index $y$ and a set of proc. nodes $N$
1: $x \leftarrow sparseApproximation(D, y)$                ▷ Equation 2.10
2: $NS \leftarrow balancedPartitioning(x, N)$               ▷ Equation 5.3
3: for $N_g$ in $NS$ do                                    ▷ in parallel
4:     $intraNodeInd(N_g, x, y)$
```

- **similarity-based grouping**: documents that share non-null coefficients positions have higher degrees of similarity on the original space;
- **redundancy**: by selecting multiple, non-null coefficients per document.

The indexing is described on Algorithm 6. For each vector to index, coordinator node extracts its sparse representation x, line 1 and selects the relevant nodes NS (at most s nodes). The coordinator node sends then the indexing request to these nodes, line 4.

The following section details why the number of nodes to index NS is not always equal to s and how to estimate its true value.

5.1.2 Sparse indexing collisions vs redundancy

Sparse hashing assigns each document to at most s posting lists; as we are distributing posting lists across nodes, each document will be assigned to, at most s nodes. But, as the hash size h is designed to be two orders of magnitude larger than m, there is a non-zero probability of collisions, where a document will be indexed on more than one posting list for a node. The redundancy factor $\hat{r}$, i.e., the expected percentage of nodes where a document will be indexed, must take into account the number of collisions. Thus, the redundancy factor can be computed as a variation of the hash collision problem, where we want to compute the expected unique number of nodes, including collisions:

$$\hat{r} = 1 - \left(1 - \frac{1}{m}\right)^s \tag{5.1}$$

The proof for this expression is available in Appendix C. This equation is key to measure the expected redundancy properties for a distributed index (e.g., select a s value that will assign documents to 10% of total nodes). As it only depends on the s sparsity factor parameter, and the number of nodes m, it can provide a redundancy factor approximation $\hat{r}$, without the need for experimental hash distribution measurements. The experimental redundancy factor r measures the percentage of nodes where a document is indexed, and $rm = r \times m$ is the average number of nodes where documents are indexed. A document with an r of 0.25 on a set of $m = 32$ nodes will be distributed across $rm = 0.25 \times 32 = 8$ nodes. Figure 5.3 shows how this effect for multiple values of s. As the number of nodes m approaches the number of posting lists h, rm approaches s, Eq. 5.2.

$$rm \approx s, m \to h \tag{5.2}$$

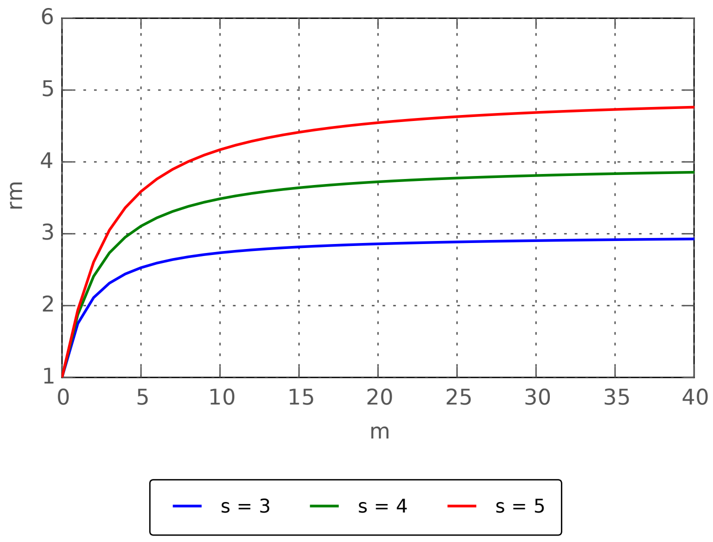


Figure 5.3: Changing the sparsity coefficient s limits the number of nodes rm where each document is indexed.

5.1.3 Balanced distributed indexes

A balanced distribution of documents across nodes ensures a more uniform usage of computational resources. The sparse hash values provide inherent partitioning through the concentration of document information into a small set of non-null hash coefficients. Sparse hashing by itself cannot guarantee uniform posting list distribution. As it is grouping documents by similarity, it can generate larger posting lists on principal directions with a higher density of feature vectors on the original space.

To create a more uniformly balanced document distribution in the sparse hash space, we balanced the total number of documents per node (i.e., the sum of the number of documents on each posting list). The distribution process is performed at posting list level. $Pl_{[0,h)}$ is the set of posting lists. h-dimensional hashes X are indexed in posting lists corresponding to the s non-null hash coefficients: $X_g \rightarrow Pl_g$, if $X_g \neq 0$ for $g \in [0,h)$. An effective partitioning scheme for balanced posting lists is to partition posting lists uniformly across m nodes, assigning h/m consecutive posting lists to each node N. For a node $N_{g\in[0,m)}$:

$$\begin{gathered} N_g \leftarrow Pl_{\left[\frac{h\cdot g}{m},\ldots,\frac{h\cdot(g+1)}{m}\right)} \\ X \rightarrow N_g, \text{ if } \exists X_b \neq 0, b \in \left[\frac{h\cdot g}{m},\ldots,\frac{h\cdot(g+1)}{m}\right). \end{gathered} \tag{5.3}$$

Formally, our goal is to minimize the absolute deviation of document distribution across nodes. Consider the set of index nodes $N_{[0,m)}$ and the set of posting lists $Pl_{[0,h)}$. The number of documents per node is obtained by the sum of the number of documents in each posting list, $pl \in N_g$. The exact number of documents per posting list Pl is unknown at the start of the distribution process. Thus, we use an estimate, $|\widehat{Pl}|$, computed from a

Algorithm 7 Balancing posting lists across nodes

```
Input: a set of $h$ posting lists $Pl$, a set of $m$ index nodes $N$
Output: set of nodes $\bar{N}$
1: $PlPerN \leftarrow h/m$
2: $P \leftarrow sortBySize(Pl)$
3: $\bar{N} \leftarrow \emptyset$
4: for $p$ in $Pl$ do
5:     $N \leftarrow sortByOcup(N)$
6:     $N_0 \leftarrow N_0 \cup p$
7:     if $|N_0| = PlPerN$ then
8:         $N \leftarrow N - N_0$
9:         $\bar{N} \leftarrow \bar{N} \cup N_0$
```

small set of validation data,

$$\begin{gathered}docs(N_g) = \sum_{Pl}^{N_g} |\widehat{Pl}| \\ \arg\min \left(\sum_{N_g \in N} |\overline{docs(N)} - docs(N_g)| \right), \\ \text{subject to } |Pl \in N_g| = h/m,\end{gathered} \tag{5.4}$$

where $\overline{docs(N)}$ is the average number of documents on each node.

Algorithm 7 solves this minimization problem, by distributing the larger posting lists to the nodes with the fewest number of documents assigned thus far, while respecting the restriction on having the same number of posting lists per node. It starts by computing how many posting lists will be assigned to each node, $PlPerN$, line 1. Then it sorts posting lists from largest to smallest, line 2. Then, for each posting list, it selects the node with the least assigned documents and assigns that posting list to that node, line 6. When a node reaches the limit in the number of assigned posting lists, line 7, it is removed from the assignment set, line 8.

This process can be performed on posting lists created from either training, validation or query log data, and will return the best possible distribution for the provided posting list distribution, considering a fixed number of posting lists per node. The index distribution can be adapted to fit seasonal patterns or large scale events that distort the search space by using query log data, as shown by Jin et al. [51].

Figure 5.4 provides a visualization of balancing process: Posting lists are assigned to nodes from largest to smallest. The top of the chart shows the first iteration: the largest partition is assigned to any node, as they are all empty. Iterations 2 and 3 will assign posting lists 2 and 3 to the remaining nodes. Iteration 4 shows the key balancing factor of the algorithm: Node 3 has the least number of documents, meaning posting list 4 will be assigned to that node. As the node already as the maximum number of allowed posting list ($h/m = 6/3$), it is removed from the assignment process. The remaining iterations assign the remaining partitions. The final partitioning is displayed at the bottom of the

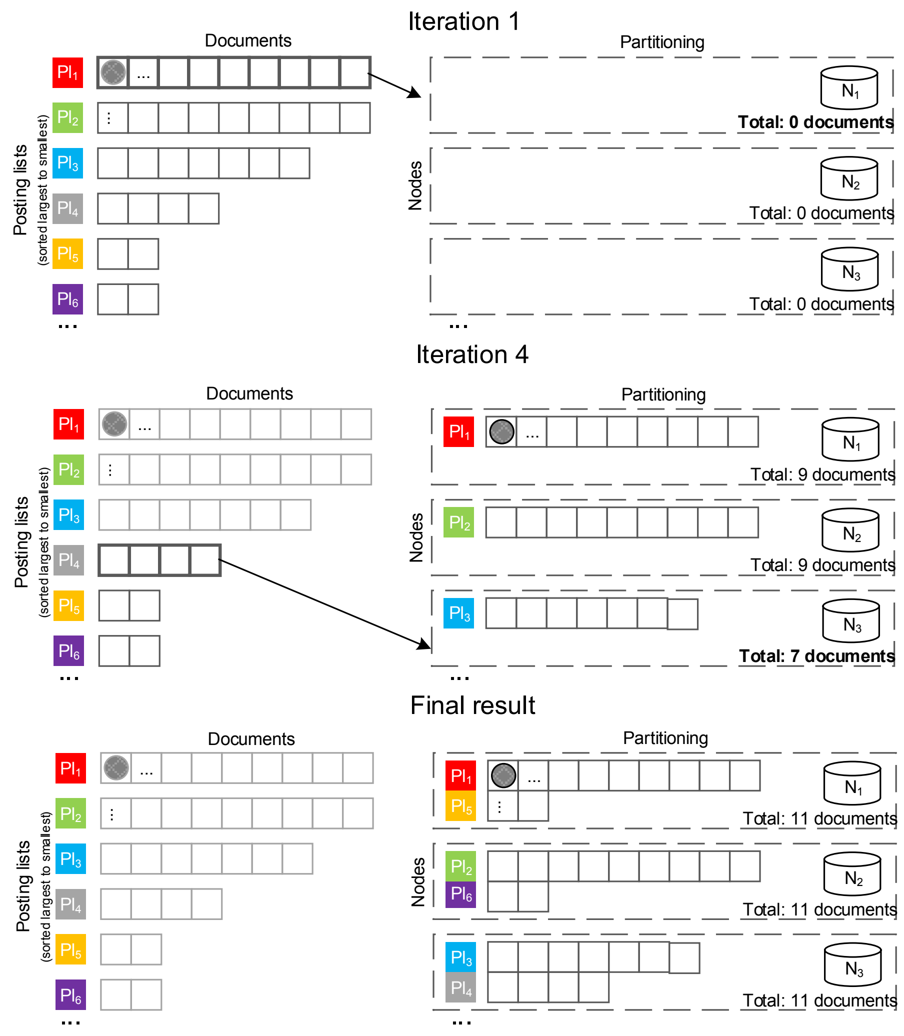


Figure 5.4: DISH posting list partitioning

figure: DISH can balance the total number of documents per node (11 documents in the example), even in scenarios where there is a large gap in the number of documents per posting list (e.g., 10 documents on the largest, 2 on the smallest).

Figure 5.5 shows how the DISH posting list distribution process compares with the distribution performed using the "Original" order returned by SHI, by a "Shuffled" posting list assignment where posting lists are shuffled and assigned to nodes sequentially. DISH posting list distribution can achieve a difference between the number of documents per node in the order of the thousands: the number of documents on nodes varied between 142,842,564 and 142,843,476. These results are very positive when compared to the "Shuffled" assignment (between 198,746,876 and 118,692,036 documents per node), where the differences in document load are in the order of the dozens of millions.

The DISH partitioning process gives our index the flexibility to support any m number of index nodes and sparsity factor s at run-time, without changing the hash size h or affecting the experimental redundancy factor r. Our index allows changing the number

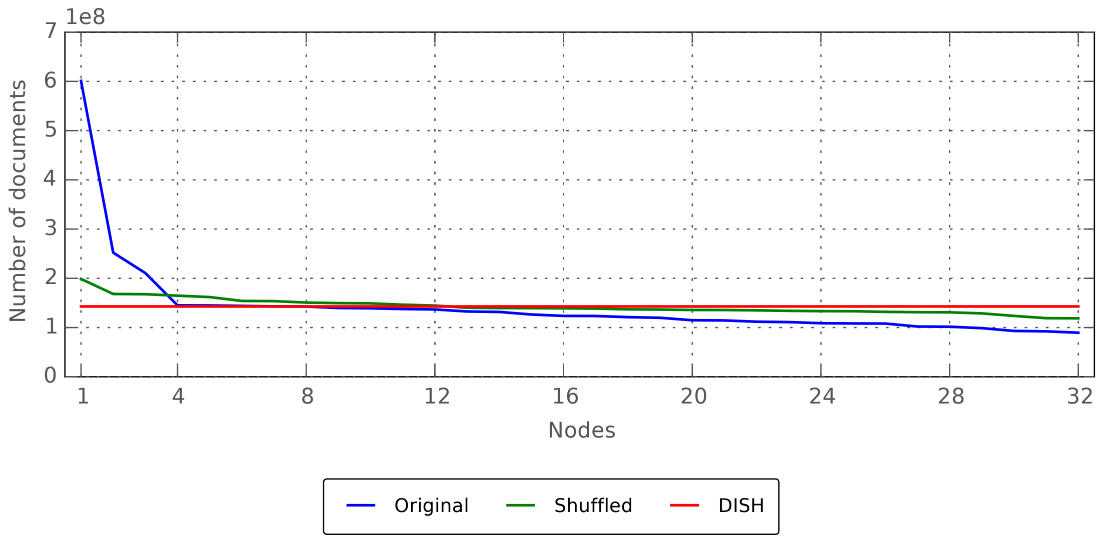


Figure 5.5: Comparison of posting list partitioning methods. "Original" represents partitioning nodes according to the order of returned by the KSVD dictionary, "Shuffled" represents an algorithm that shuffles the order of the posting lists before distribution and "DISH" represents the DISH's greedy partitioning process.

of index nodes, as it can re-balance posting lists across nodes by reordering the posting lists. This technique also works if the number of indexed documents unexpectedly varies across lists. Flexibility is also achieved by having different sparsity factors s at indexing and query time. For example, one can have a higher s at indexing time and have higher redundancy and a lower s at retrieval time, for faster search (inspecting fewer nodes).

5.1.4 Distributed retrieval

The distributed retrieval procedure addresses two constraints: i) minimize the number of nodes deployed to answer one single query and ii) explore DISH's partitioning redundancy properties to reduce the number of candidates to inspect.

5.1.4.1 Retrieval coordination and rank aggregation

By querying fewer nodes per query, one reduces network communication, can handle a higher amount of concurrent queries and improves the index vulnerability to hardware failures. Our goal is to be *efficient* by querying only nodes with potentially relevant documents. To meet our efficiency goal, we take advantage of the index inherent data partitioning and intra-node limited candidate selection.

Algorithm 8 details the querying process from the coordinator node point of view. First, the sparse hash is computed for the query. The coordinator node runs the sharding process, line 2 and queries the selected nodes, line 5. The final step of the retrieval process is to collect the partial candidate lists from the index nodes and combine them into the

Algorithm 8 Distributed retrieval (coordination node)

Input: a sparsifying dictionary D, a query vector y, the number of neighbors to retrieve k, a pruning factor pf and a set of index nodes N
Output: a list of k nearest neighbors C.

```
1: $x \leftarrow sparseApproximation(D, y)$          ▷ Eq. 2.10
2: $NS \leftarrow balancedPartitioning(x, N)$        ▷ Eq. 5.3
3: $C \leftarrow \emptyset$
4: for $N_g$ in $NS$ do                              ▷ in parallel
5:     $C_g \leftarrow intraNodeRet(N_g, x, D, pf, k)$
6:     $C \leftarrow C \cup C_g$                     ▷ Alg. 5
7: $C \leftarrow reduce(C)$          ▷ Perform final sort by $l_2$ dist to the query
8: return $C_{1,\dots,k}$
```

final rank, line 7. As the lists are already sorted and pruned in the index nodes, the coordinator node only needs to sort a small number of candidates (at most $k \cdot s$).

The retrieval operation on line 5 is blocking; the coordinator node waits for the results from all selected index nodes. To avoid blocking the querying process due to the lack of response from an index node (e.g., lost packages, nodes crashing), we set a timeout for that operation: if an index node does not answer in a set time (dependent on the expected query time, but usually in the order of $10 - 100$ ms), the fusion procedure continues with the partial results gathered from the remaining nodes. Due to the inbuilt redundancy of the distribution algorithm, the impact on retrieval performance is small (see Section 5.2.6 experiments for details).

5.1.4.2 Intra-node index pruning

Pruning the number of inspected candidates is an important process for fast indexes in general and even more critical for multimedia retrieval. The rationale is that by examining only documents with a high probability of similarity to the query, one can reduce the overall computational complexity. Inspired by text indexes, we implemented two types of pruning factors, $pf_{[0,\dots h]}$, to select how many candidates will be re-ranked for each posting list, on an index I with n documents. The first strategy uses a fixed number of documents to re-rank the same number of documents across all posting lists and similar candidate re-raking across nodes:

$$\textbf{Fixed: } pf \in [1, n] : pf_{[0,\dots h]} = pf. \tag{5.5}$$

The second strategy uses a variable number of documents corresponding to a percentage of the length of the posting list pf_i, based on the value of hash X_i for that posting list, divided by the sum of all the values in the hash:

$$\begin{gathered}\textbf{Hash-based: } \text{global } pf \in [0,1] : \\ \forall X_{i \in [0,\dots h]} \rightarrow pf_{i \in [0,\dots h]} = (pf \cdot n)\,(X_i / sum(X))\end{gathered} \tag{5.6}$$

The rational behind the hash-based pf is in-line with the greedy nature of the OMP sparse hash computation: the node with higher coefficients, contains more correct nearest neighbors. This removes the need for the coordinator node to know the index node occupation state. Note that the computed pruning factors are maximum values: if a posting list has fewer elements than the factors, only those elements are inspected (it does not "overflow" to other postings lists). This factor is key to explain the results of the posting list inspection experiments in Section 5.2.3.

The sparse hashes provide a sound, redundant partitioning of the index. The same vectors are indexed in multiple partitions and each partition stores similar vectors. Thus, pruning the index delivers important gains that can be further leveraged by state-of-the-art algorithms for single node search (e.g., [11, 49]), without any changes to the index redundant partitioning, balancing and aggregation procedures.

5.1.5 Computational complexity

This section describes the **complexity of the retrieval process**, in terms of the number of documents required in the index to inspect per query per node and the amount of main **memory** (RAM) necessary to store the inverted index and vectors.

5.1.5.1 Retrieval complexity

Considering a set of n, d-dimensional vectors, the original nearest neighbor problem has a theoretical computational complexity of $O(n \cdot d)$. Methods based on Hamming embeddings (e.g. [44, 47, 61]), greatly reduce the dimensionality resulting on the computational complexity $O(\frac{d}{f} \cdot n)$, where f is the dimensionality reducing factor (it is common to reduce 300 dimensional real-valued vectors into 64 bit dimensional vectors). Approaches based on space-partitioning techniques (e.g. [48, 49, 54]) tackle the dimensionality factor n.

Our approach is to distribute the index across m nodes and prune the search inside each node by a factor $pf \in [0,1]$. For a fixed pruning factor $fixed_{pf}$, it can be approximated by dividing by the index size and multiplying by the sparsity factor $pf = (fixed_{pf} \times s)/n$ Thus, the computational complexity can be unrolled into $O((pf \cdot n) \cdot (\frac{d}{m}))$. When we consider the redundancy factor that is embedded in our method as the sparsity factor, the computational complexity is:

$$O\left((pf \cdot n) \cdot \left(\frac{r}{m} \cdot d\right)\right) \tag{5.7}$$

with $\frac{r}{m} < 1$ and $pf < 1$. The proposed method can effectively distribute the search load across nodes with a known redundancy r. The complexity arising from the factor n is dependent on the intra-node retrieval method used on the index nodes.

5.1.5.2 Main memory complexity

The main memory requirements can be divided into two parts: the inverted index structure and the original space feature vectors. For the inverted index structure, one needs to store

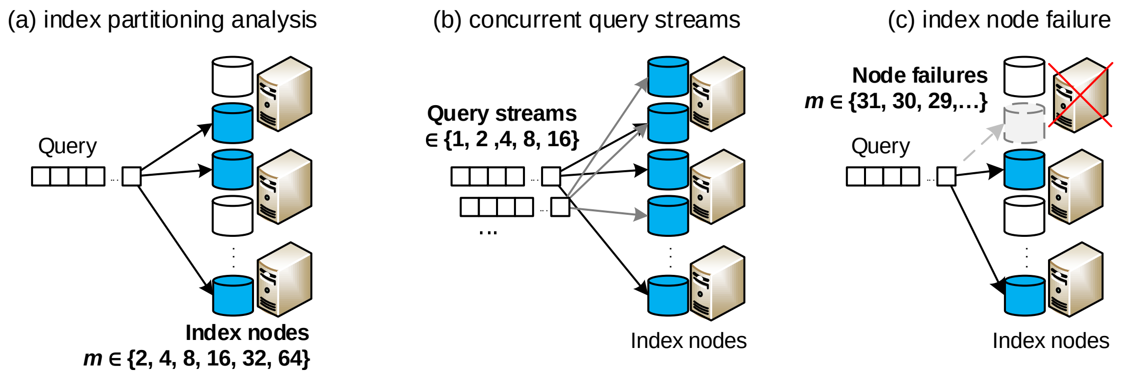


Figure 5.6: Overview for the DISH evaluation process: each diagram emphasizes which component of the architecture is being evaluated.

a four byte id and a four byte float coefficient for each index entry. Due to the sparsity factor, the number of inverted index entries per feature vector is s, and thus the final number of is $2 \cdot n \cdot s$. Regarding the indexed documents, each node must store the vectors that are stored with its inverted index partitions. As the redundancy factor provides the expected number of nodes a document a document will be index in, the complexity is $n \cdot r \cdot d$. Note that for the feature vectors, the exact memory requirements depend on whether the features can be represents as a short with one byte per entry (e.g., SIFT) or floats, four bytes per entry (e.g., GIST). Thus, considering that the index is distributed across m nodes, the final main memory requirements are:

$$\frac{n \cdot r \cdot d + n \cdot s \cdot 2}{m} \tag{5.8}$$

5.2 Experiments

This section evaluates the performance of DISH on a multi-tenant cloud environment, under multiple conditions. We start by analyzing DISH efficiency in section 5.2.4, Figure 5.6 (a), and examine how balanced our querying distribution is, and how query distribution changes with the number of nodes. Section 5.2.5 assesses the performance impact of answering multiple concurrent query streams, Figure 5.6 (b). Experiments concerning how node failures affect retrieval performance are reported on section 5.2.6, Figure 5.6 (c).

5.2.1 Baselines

Inspired by content-based distributed text indexes such as [56], we adapted k-means to compute sparse hashes. K-means is an adequate baseline, as it provides comparable retrieval performance to state-of-the-art indexing techniques based on clustering or quantization such as [50], which use k-means as a key part of the partitioning process.

The basic idea is to generate hashes based on the distances of original feature vectors to the set of s closest centroids (soft clustering). The process of using k-means to extract sparse hashes is as follows:

- compute a set of h centroids, based on the same training data as with sparse hashes using k-means++ [7];
- compute h-dimensional sparse hashes x with s non-null positions, corresponding to the s closest centroids in the Euclidean space.

The remaining hash positions have the value of zero. This process replaces the *sparseApproximation* step in Algorithms 6 and 8. Both k-means and KSVD can have the same hash size h: in k-means corresponds to the number of centroids and in KSVD to the number of dictionary codewords.

Consider a set of cluster centroids $C \in \mathbb{R}^{n \times d}$. Hash computation by clustering is based on finding the set of closest centroids $c \in \mathbb{R}^{s \times d} \subset C$, and assigns the Euclidean distances to those centroids as the sparse vector values:

$$\begin{gathered}\arg\min_c(\|c_i - y\|_2),\\ \text{for } c_{i\in[0,h]} \in C,\\ \text{with } |c| = s\end{gathered} \tag{5.9}$$

$$x_{i\in[0,h]} = \begin{cases} \|c_i - y\|_2, \text{ for } c_i \in c \\ 0, \text{ otherwise} \end{cases}$$

To find the set of centroids that best represent the feature space, three techniques were selected: random sampling, k-means and fuzzy c-means. Alternative clustering techniques such as DBSCAN do not allow setting the number of clusters and thus, do not meet the desired properties.

Fuzzy c-means clustering [17] techniques extend the assignment of documents to clusters and keep membership information to multiple clusters (e.g., ratio of the distance to the centroids). As with k-means, these techniques minimize the sum of the document to centroid distances, taking into account membership and cluster fuzziness information (degree of overlap between clusters). In preliminary experiments, fuzzy c-means produced very unbalanced clusters: all documents were assigned to only 20 clusters, regardless of the total generated centroids. Due to this extreme unbalance, fuzzy c-means was not explored further. k-means++[7] was used for initial seed centroid selection.

Using these centroids, h-dimensional (one for each centroid) hashes are computed so that each hash x has s non-null positions, corresponding to the s closest centroids in the Euclidean space. The remaining hash positions have the value of zero. Figure 5.7 illustrates the k-means hash creation process on a two-dimensional space using a set of $h = 5$ centroids and $s = 2$. Centroids are represented as colored squares, indexed points as colored dots, matching the color of the centroid with the lowest distance and a new point is represented as a grey dot. This process replaces the *sparseApproximation* step in Algorithms 6 and 8; The distributed indexing and retrieval are similar, with a small difference: as we are

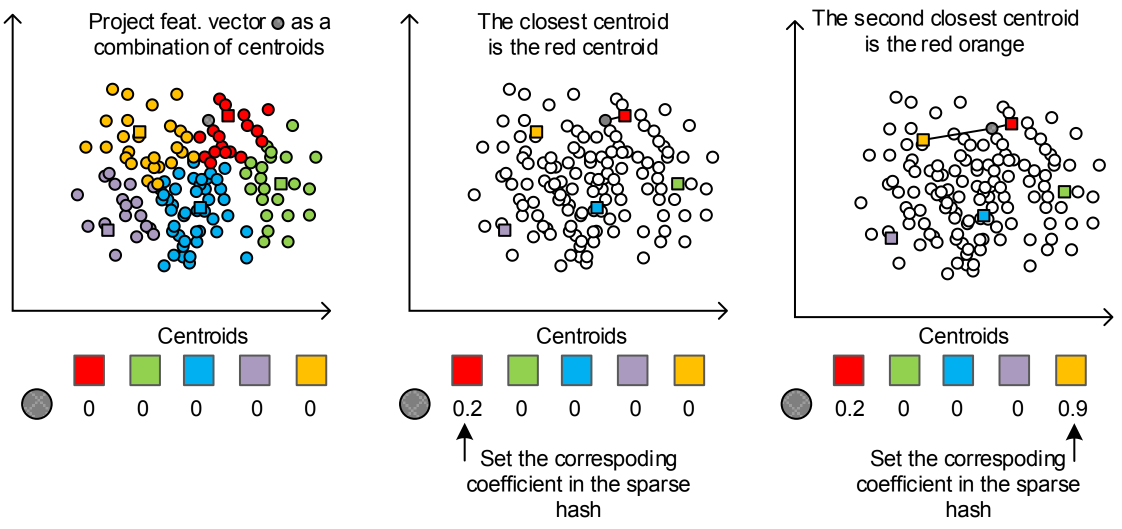

Project feat. vector ● as a combination of centroids
Centroids
0 0 0 0 0
The closest centroid is the red centroid
Centroids
0.2 0 0 0 0
Set the correspoding coefficient in the sparse hash
The second closest centroid is the red orange
Centroids
0.2 0 0 0 0.9
Set the correspoding coefficient in the sparse hash

Figure 5.7: Fuzzy k-means indexing

dealing with distances instead of reconstruction coefficients, the posting list sort order is reversed (i.e., documents with smaller distances to centroids are inspected first). On this chapter experiments, s was set to 5, which gives a slight advantage to k-means over KSVD that has a mean sparsity of 4.32.

5.2.2 Experimental setup

Machines: all nodes are virtual machine (VM) instances on Microsoft Azure. The VM host machine controls resource scheduling (e.g., CPU, network, storage), some of which may be shared with unrelated VMs. Both index and coordinator nodes are Microsoft Azure Standard D12 virtual machines[1], with four virtual cores (2.2 GHz Intel Xeon E5-2660), 28 GB memory and a 200 GB Local SSD (as of July 2017). All coordination and index nodes were allocated individual virtual machines; each machine uses four threads (one per core) for sending querying requests to index nodes (coordination nodes) and for index candidate inspection (index nodes).

Network: all nodes were in the same zone/datacenter (US East 2), connected to a 10 Gbps network, with sub 1 ms latency between nodes. Communication between nodes is made through UDP using a custom, low overhead binary protocol to send query requests between coordination and index nodes and send the sorted candidate nearest neighbours between the index and coordination nodes. During our experiments, we did not observe any package loss.

Datasets: the ANN dataset [48, 49], contains over one billion SIFT feature vectors, section 2.6.4. It was designed to evaluate the quality of nearest neighbors search algorithm on very large scales. We used the first 1000 vectors from the query set of queries on all non-concurrent experiments. In the concurrent query stream experiments, each query

[1]`https://azure.microsoft.com/en-us/pricing/details/virtual-machines/series/`

stream uses a set of 1000 queries randomly selected from the full 10,000 query set. We used a subset of 100,000 of the provided training vectors to compute the K-SVD dictionary and k-means centroids.

This setup follows the standard data splits for the ANN dataset but we measure avgP@50 (i.e., percentage of the true set of 50 nearest neighbours that are present in the top 50 positions of the rank list produced) instead of 1-Recall@1000 (percentage of times the true nearest neighbour is present on the top 1000 positions of the rank list produced). Our reasoning is P@50 can measure the distribution of the top true 50 nearest, instead of the top true nearest neighbour of 1-Recall@1000, which is in line with the goal of a search engine that aims at retrieving multiple relevant results for each query.

Metrics: the metrics used in the evaluation are rooted in both retrieval and distributed systems literature [42, 96, 97]. We measured average **query time** as the time it takes for the aggregator node receiving the query request, and returning the list of nearest neighbors, which corresponds to Algorithm 8. For retrieval effectiveness, we measured average **precision at** k:

$$\text{avgP@k} = \tfrac{1}{nq}\textstyle\sum_{n=1}^{nq}, \tfrac{|T_{n,[1,k]} \cap C_{n,[1,k]}|}{k}, \tag{5.10}$$

where T_i is the set of true nearest neighbors and C_i is the set of retrieved neighbors for a query i and nq is the number of queries. **Node load** is defined as the percentage of time a number of index nodes were being queried, over the time it took to answer all queries. For example, on a two-index node system, a node load of 30% on one node and 70% on two nodes, means that one of the nodes was answering a query for 30% of total time, and on the remaining 70% of the time, both nodes were answering a query.

Bulk indexing details: the indexing is divided into four stages: first, feature vectors are extracted from the documents; second, a dictionary is computed from training documents; third, sparse hashes are generated using the computed dictionary (2 ms per document) and placed at the corresponding posting list. Then, for each posting list, we sort them according to the hash coefficient value. The indexing process was performed on a 12-core Intel i7 machine with 64 GB of RAM. The dictionary training took approximately 8 hours for a 100,000 set of training feature vectors. Sparse hash extraction, posting list indexing, and sorting take about 48 hours for a 1 billion document index. Posting lists are stored into a centralized network data store on Azure. For each experiment, they are divided across nodes using the order returned by the posting list balancing algorithm, Alg. 7.

Parameters: Parameters were selected to measure the impact of the distribution overhead of DISH sharding process. We set a hash size h of 8192 for both methods, a sparsity factor s of 4.32 for DISH and 5 for k-means. The number of posting lists per node depends on the number of index nodes m: 8192 / m.

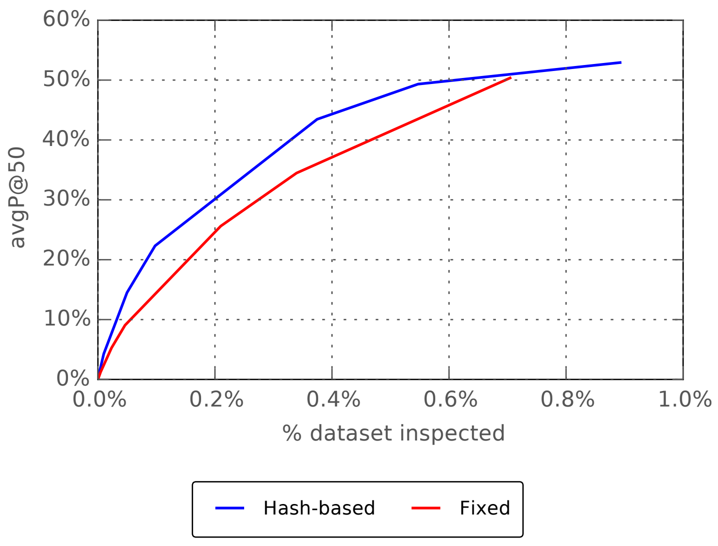


Figure 5.8: Posting list search effectiveness: Precision vs. percentage of index inspected

5.2.3 Precision vs. index inspection efficiency

The goal of the sparse hash index is to group similar documents into the same posting lists. At retrieval time, one must find the balance between precision and inspection efficiency. The goal of the following experiments is to establish the precision of the system and measure how it is affected by the % of the index inspected. Experiments were performed on an environment with 32 index nodes and one coordinator nodes and one billion vectors. Note that the percentage of dataset inspection from Eq. 5.6, refers to the real inspected percentage meaning that it will be lower than requested, as the posting lists may be smaller than the requested limit.

Figures 5.8 show the comparative results. These results show that, for both types of limits, we achieved an average P@50 of 50%, while inspecting less than 0.6% of the dataset (less than 6 million documents). For the tested limits, hash-based inspection is more efficient (inspects fewer documents for equivalent precision) that using a fixed limit. The effect is more clear at lower limits. This means that the magnitude of the hash coefficient is an important factor for candidate selection. As a result of this experiment, we used the hash-based criterion to limit the posting list inspection depth.

This experiment shows the retrieval effectiveness of the sparse hash index for multiple degrees of index inspection. The index node query time is tied to the percentage of the index inspected. The goal of DISH is to balance the size of the index of partitions across index nodes, so that inspection load is better distributed across nodes. The following experiments show how the search process is affected by the distribution of the index to multiple nodes.

Table 5.1: Impact of number of index nodes and index size on DISH overhead.

			Query dist. overhead (ms. per query)			
Index size	Dist. size	# of nodes	DISH part.	Commu-nication	Aggre-gation	Total time
50M	229M	2	3.766	0.109	0.005	3.880
50M	229M	4	3.308	0.108	0.007	3.423
50M	229M	8	3.219	0.103	0.010	3.332
50M	229M	16	3.342	0.104	0.014	3.460
50M	229M	32	3.064	0.104	0.011	3.179
100M	457M	2	4.202	0.113	0.004	4.319
100M	457M	4	4.131	0.124	0.007	4.262
100M	457M	8	4.143	0.126	0.009	4.278
100M	457M	16	4.144	0.131	0.010	4.285
100M	457M	32	4.209	0.136	0.011	4.356
1000M	4571M	32	4.414	0.197	0.012	4.623
1000M	4571M	64	4.744	0.201	0.016	4.961

5.2.4 Index partitioning analysis

DISH partitioning is the process of sparse hash computation and assignment to index nodes by the coordinator nodes, described on section 5.1.1. We first examined the impact of the number of documents and the number of nodes in the partitioning of the index, Figure 5.6 (a).

Balanced document-to-node allocation. The goal of this experiment is to measure DISH balancing properties for documents, Algorithm 7, for varying amounts of index nodes. The total number of feature vectors on the index is 1 billion. The index was distributed across 32 nodes. The document-to-node allocation follows a close to uniform distribution: differences between nodes' occupation are in the order of the few thousands: the number of documents on DISH varied between 142,842,564 and 142,843,476 (mean value: 142,843,085). For k-means, it varied between 156,249,993 and 156,250,005 (mean value: 156,250,000), Note that these values take the redundancy factor into account.

Index redundancy. Table 5.1 shows the impact of the number of index nodes (2 to 64) and index size (50M, 100M and 1000M) on the DISH process. The distributed index size (dist. size) column illustrates the index redundancy; i.e., documents are indexed on average on 4.5 index nodes, hence, increasing the redundancy and robustness of the index. Although having a larger index may seem like a potential disadvantage of our partitioning scheme, we argue that it has the potential to reduce the need for additional replication by node duplication. The node failure analysis in section 5.2.6 will expose this advantage of the proposed method.

Table 5.2 further explores the impact of the number of index nodes m on the index redundancy factor r, and compares it to the expected value $\hat{r}$. For the queries, the value

Table 5.2: Measured and expected redundancy factor for the queries. rm is computing by multiplying the measured redundancy percentage r with the (varying) number nodes m. The number of partitions per document is fixed.

$m = \#$ of nodes		2	4	8	16	32	64
	rm	1.90	2.92	3.68	4.16	4.48	4.48
DISH	r	0.95	0.73	0.46	0.26	0.14	0.07
	$\hat{r}$	0.96	0.74	0.46	0.26	0.14	0.07
	rm	1.94	3.12	3.92	4.48	4.80	4.93
k-means	r	0.97	0.78	0.49	0.28	0.15	0.08
	$\hat{r}$	0.97	0.76	0.49	0.28	0.15	0.08

r means the difference between the expected and measured number of nodes to query. The sparsity factor was $s_{query} = 4.65$ for DISH and $s_{query} = 5$ for k-means. For a small number of nodes, e.g., 2 or 4, the redundancy factor is limited by the total number of nodes, meaning that most documents will be placed on all nodes. For larger values, the redundancy factor approaches the value of s_{query}. The table shows that the r values are in line with the expected values computed from Equation 5.1. This shows that queries are being distributed across nodes according to our redundancy predictions.

Overhead. The communication time corresponds to the time sending queries and receiving results from index nodes, and the aggregation time corresponds to the time to combine and sort candidate nearest neighbors from the index nodes. The per-query coordination time is within a small range (3-5 ms) over all index node and index sizes. Section 5.2.5 will further analyze the DISH overhead in the presence of multiple query streams.

Balanced query allocation. Figure 5.9 shows the number of queries assigned to each node. The X-axis shows the node id, sorted from least to more queries answered and the Y-axis shows the percentage of queries that were assigned to the node. Query balancing is also good: each node answers 14.5% of all queries on average, with a maximum difference of less than a 4% in load between nodes (between 12.4% and 16.3% of queries). k-means achieves very similar results for single stream balancing.

The next step is to examine how changing the number of index nodes affects the balance of the querying process. In Figure 5.10, each chart corresponds to a fixed number of index nodes (8, 16, 32). The X-axis shows the number of nodes that were queried; the size of the axis is the total number of nodes in the experiment. The Y-axis shows the % of the experiment time where a number of nodes were busy answering questions. For example, the eight index node chart shows that, for the 1000 tested queries, four index nodes had to be queried for about 30 % of the time and six index nodes had to be queried for about 10% of the time. As expected, as one adds more index nodes, the number of nodes queried increases slightly. For 32 index nodes, the coordinator node queries, at most, seven nodes (about 20% of total nodes).

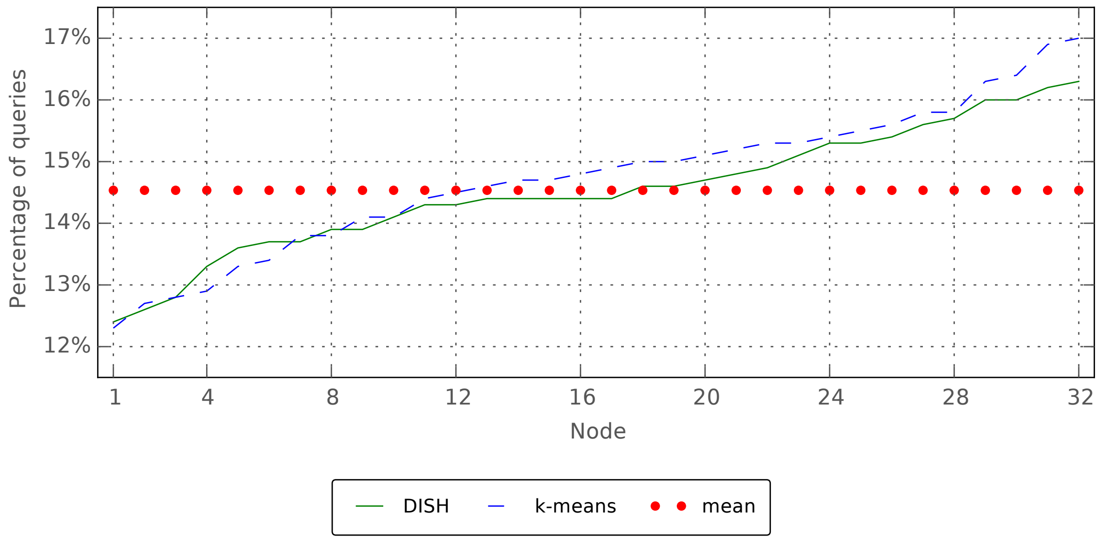


Figure 5.9: Percentage of total queries assigned to each index node (32 index nodes)

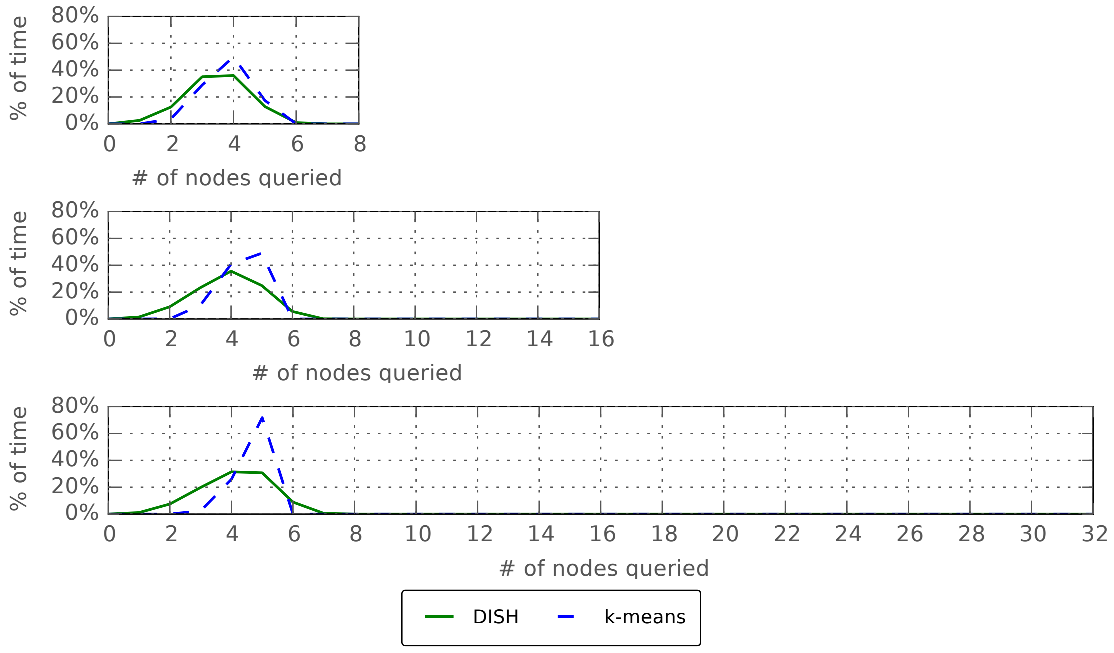


Figure 5.10: Distribution of the number of nodes queried as a percentage of total query time, for multiple numbers of index nodes.

On DISH, the average number of nodes being queried increases from 3 to 5 when the number of indexer nodes grows from 8 to 32. For k-means, the change in the number of queried nodes varies less (4 to 5).

The variance in number of nodes queries being queried at a time is lower on k-means (i.e., stays closer to the average) than on DISH. In addition, a higher variance in the percentage of total nodes queried per query (as with DISH) can be beneficial for concurrent querying (e.g., lower potential bottlenecks on more popular nodes). The following section details

Table 5.3: Concurrent query streaming results: time and performance relative to a single query stream.

# Query streams	Query distribution overhead (ms) DISH	k-means
1	4.62	1.55
2	5.15	1.59
4	5.17	1.59
8	5.12	1.58
16	5.32	1.56

how these factors impact an index answering up 16 concurrent query streams.

5.2.5 Concurrent query streams analysis

Indexes deployed on real-world situations must work in environments where queries are issued concurrently. An often overlooked, but key experiment, is *how media indexes deal with answering multiple streams of queries concurrently.* In this experiment, we tested how having multiple concurrent query streams (1, 2, 4, 8, 16) affects the performance of the DISH index, which corresponds to Figure 5.6 (b) scenario. The number of index nodes is fixed at 32. Note that retrieval accuracy (e.g., avgP@50) does not change with the number of concurrent queries, as the candidate documents examined during the querying process are the same.

Table 5.3 shows how DISH behaves on a situation where a set of index nodes being queried by multiple coordinator nodes concurrently. The *DISH* column shows the time the coordinator node takes to compute the hash (section 5.1.1), query the index nodes and aggregate the results (algorithm 8). The total query relative time depicts the relative slowdown over a single query stream. On the times for the *DISH* column, we see that multiple query streams have little effect on the time taken by the distribution and aggregation process. Increasing the number of concurrent query processes from 1 to 16 only increases the average DISH time from 4.62 to 5.32 ms. Regarding global retrieval time, it is only 0.99x slower answering 16 query streams vs. answering a single query stream. k-means sharding time is lower than DISH, but the global query time grows faster with more query streams (Total relative time). These results show that DISH distribution overhead does not increase significantly with the number of parallel query streams.

This is due to the system's efficiency, as described in section 5.2.4: only nodes where a non-zero coefficient matches their posting lists are queried. Combined with the balancing between hash size and the number of nodes (256 posting lists per node), the probability of querying a node is even lower (worst case scenario, only one node is queried) and thus, performing multiple queries in parallel will result in fewer collisions.

Figure 5.11 details the load-balancing across index nodes under concurrent query streams for DISH and k-means. The X and Y-axis are similar to those of Figure 5.10, but each chart corresponds to a different number of query streams (1, 2, 4, 8, 16). Each query

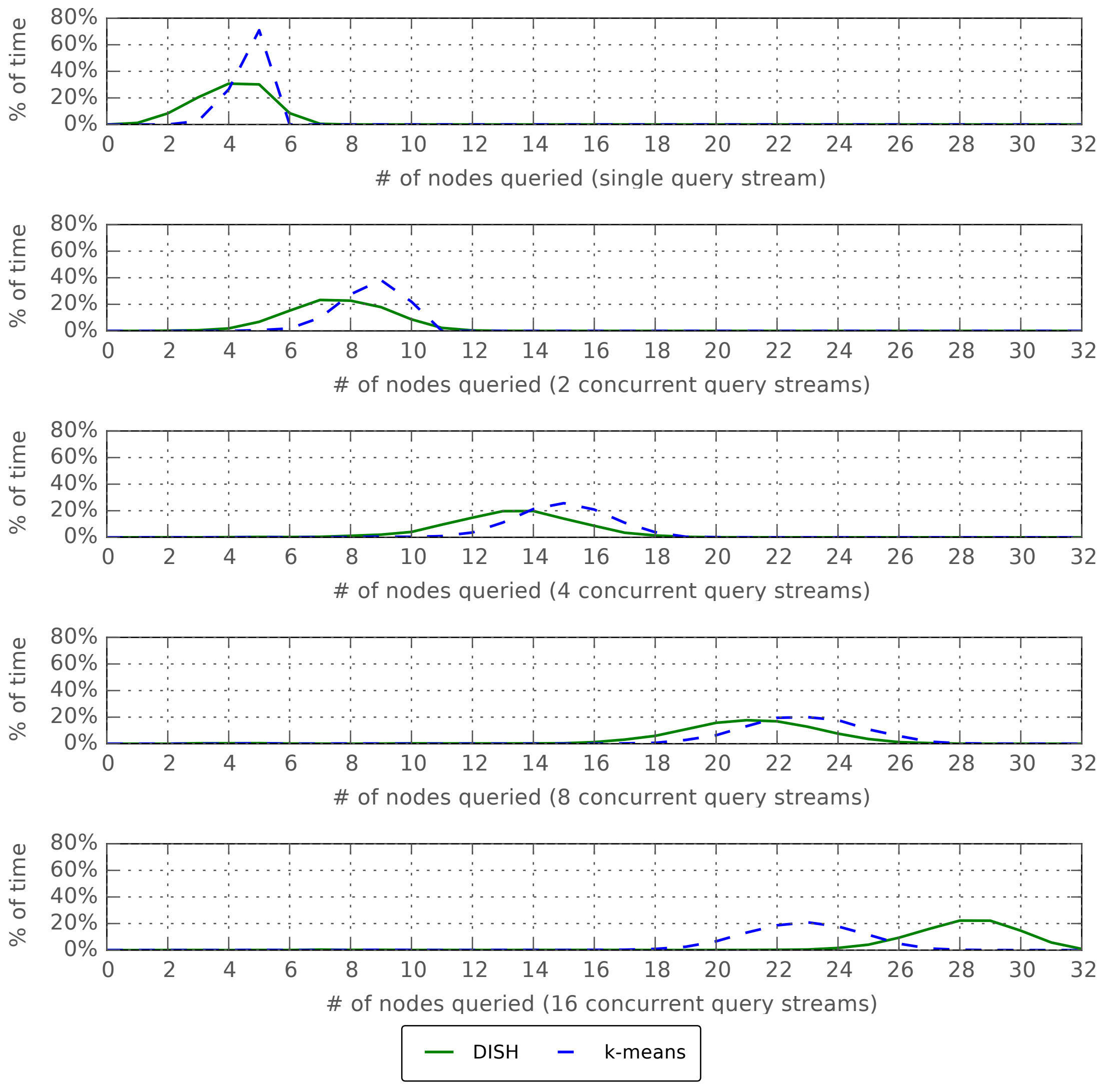


Figure 5.11: Distribution of the number of nodes queried as a percentage of total query time for multiple query streams. Each chart represents a different number of streams (1, 2, 4, 8, 16).

streams injected 1000 queries. Looking at the index nodes load with one query stream, Figure 5.11) top row, we can see that the most index nodes are largely unused. This is the result of the sparse hashing technique that only uses a few dimensions of the hash. When the number of concurrent streams increases, the probability of using more nodes increases, resulting in a higher occupation rate of the full set of index nodes. For 16 query streams, the mean number of occupied nodes is 28.5 nodes, reaching its maximum (32 nodes occupation) a non-negligible percentage of the time ($\approx 3\%$). When compared with k-means, DISH can spread the load more across nodes. The effect is more evident in the 16 stream experiment, where k-means queries the same amount of nodes as with the eight stream experiment, which leads to a larger concentration on load on nodes when compared

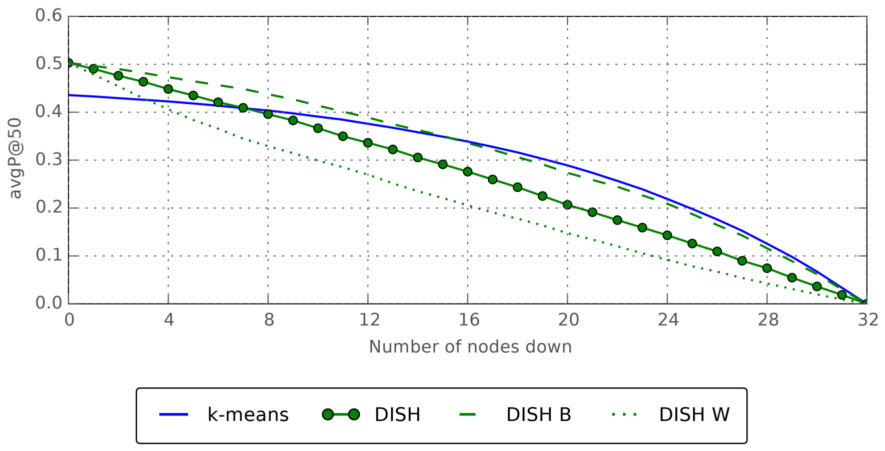


Figure 5.12: Avg. precision on index node failures.

to DISH.

5.2.6 Resilience to index node failure

When working in a distributed environment, node and network failure risk increase significantly with scale. Our sharding scheme implicitly offers redundancy, by computing sparse hashes with over-complete dictionaries. KSVD dictionaries offer an overcomplete, orthogonal view of the original feature space, meaning that document features will be distributed across multiple nodes, according to the sparsity coefficients.

The goal of this experiment is to measure the impact of those missing nodes on retrieval precision, Figure 5.6 (c). The importance of this measurement is two-fold: assess the resiliency of our redundant index partitioning scheme to node or network failures, and measure what would happen if one would voluntarily not query nodes (e.g., node load too high). Figure 5.12 shows the average precision loss in a situation where the coordinator node does not get a response from one or more nodes. The X-axis represents the number of nodes that did not return results, between 0 (all nodes returned results) and 32 (no nodes returned results). The Y-axis represents the avgP@50. The *DISH B(est)* baseline represents the best case scenario, where the worst global performing node (regarding average precision for all queries) goes down; the *DISH W(orse)* baseline represents a situation where the best global performing node goes down. *DISH* and *k-means* series represents what happens when a random node goes down for DISH and k-means respectively.

As we start to query fewer nodes, the precision of DISH is higher than k-means. This experiment also shows that losing a few nodes does not have a considerable impact on performance. The loss of one node only results in an average P@50 decrease of 2%. The figure also highlights that the differences between the DISH Best and DISH Worse case series are between 10% to 15%, which corresponds to the impact of the order in which individual nodes go down.

The current version of the intra-node index pruning is based on a linear search across index shards. A more efficient index node pruning would be able to inspect more candidates in a similar time frame, improving precision.

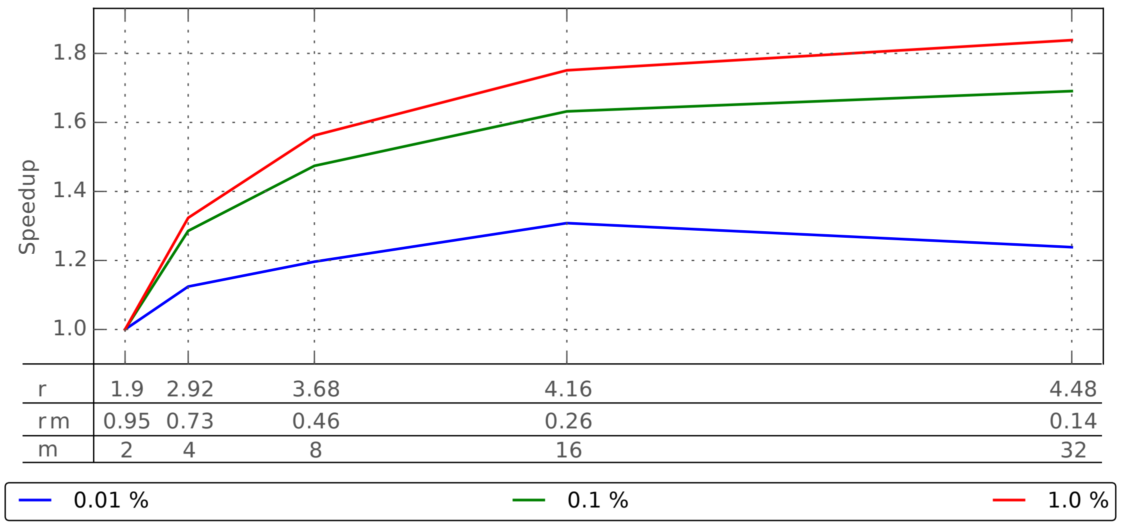


Figure 5.13: Query time speedup with the number of nodes, for multiple degrees of index inspection. rm is computing by multiplying the measured redundancy percentage r with the (varying) number nodes m. The number of partitions per document is fixed.

5.2.7 Performance limits

In this section we examine the limits beyond which adding more indexing nodes brings no further improvements. Namely, the fine trade-off between indexes *resiliency to node failure* and *retrieval speedup* when one changes the index *redundancy factor*. Redundancy also has an impact on the distributed index speedup. For too high *redundancy factor* values, the speedup is sacrificed, because index nodes have too many duplicate documents. However, lowering the index *redundancy factor* r will release more index nodes to quickly answer incoming queries, thus resulting in a speedup increase.

The following experiments analyze the break-even points for the *redundancy factor* and *inspection depth*.

Redundancy factor. In this experiment, we varied the redundancy factor by changing the number of nodes m and fixing the sparsity coefficient to 4. Figure 5.13 examines this behavior: it shows the impact of changing the redundancy factor, i.e., number of index nodes, in the retrieval speedup for a 100M document indexes. The Y-axis represents the speedup compared to searching on two index nodes m. The X-axis represents the total number of index nodes (m). Each series represent a different % of index inspection, 0.01%, 0.1%, 0.5% and 1.0%.

As expected, the effect of adding more nodes is more pronounced for higher % of index inspections, reaching a speedup of 1.83× for a redundancy factor of 0.14 (i.e., there are 32 nodes, but each query is served on average by 4.48 nodes). For lower inspection depths,

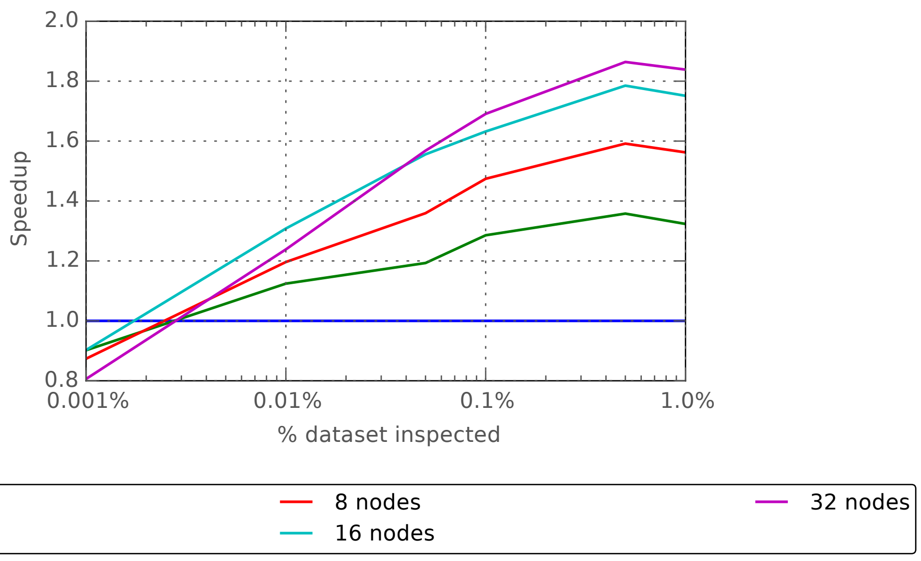


Figure 5.14: Query time speedup with multiple degrees of index inspection for multiple node counts.

e.g., 0.01%, distributing the index to 32 nodes resulted in a decrease in performance; we hypothesize that the increase in index distribution time was higher than the time gained from having more indexing nodes inspecting partitions.

The smaller gains in performance for higher numbers of nodes are linked to the rm factor: as the number of nodes per document rm approaches the sparsity factor s, (in this experiment, $m > 8$), adding more nodes to the index results in an increase in resilience to node failure (higher rm). Improvements on speedup are strongly related to the sparsity factor, which is fixed in this experiment. For a single query stream, the improvements in performance only happen while its posting lists that are placed on a single node are distributed to these new nodes. Adding more nodes can improve performance scenarios with multiple queries such as the ones described in Section 5.2.5.

Inspection depth break-even. The index inspection depth has a high impact on the speedup. Intuition tells us that for shallow inspection depths, the distribution overhead is too high, thus canceling the advantage of having a distributed index. Figure 5.14 gives a different perspective of the impact of changing the number of index nodes and % of index inspection. As with the previous experiment, the Y-axis represents the speedup compared to searching on two index nodes. The X-axis represents the % of index inspection. Each series represents a different number of index nodes.

This experiment shows that the % of index inspection has a significant impact on distributed retrieval performance: for search limits below 0.003%, distributing the index to more nodes results in a degradation in temporal performance ($speedup < 1$). For such low limits, the extra index partitioning overhead is greater than the gains in candidate inspections from having more nodes. Thus, the larger number of index nodes only brings benefits when inspecting larger % of the index.

5.3 Conclusion

This chapter described DISH, a large-scale index sharding algorithm by sparse-hashing, designed for the cloud. We showed how sparse hashes can be used to shard documents and queries in a balanced and redundant manner across nodes. We studied how to predict the impact of distributing hashes over multiple nodes on the expected redundancy, based on the hash collision problem, providing experimental results that show that DISH closely follows the expected distribution.

The evaluation section shows that DISH has low sharding overhead and distributes load effectively across nodes when under 16 concurrent query streams. Moreover, we also observed a graceful performance degradation on node failure (2% precision loss per node failure), which is a significant advantage on cloud environments where high-latency machines are a hindrance.

However, balancing documents across nodes, Figure 5.5 is not the same as balancing the search space distribution. DISH retrieval process is still limited by nodes that have larger partitions, as they inspect a larger number of candidate nearest neighbors, as described in Section 5.1.4.2. This problem is amplified by parallel query stream scenarios, Section 5.2.5: load balancing becomes less effective if multiple queries are assigned the same large posting list. The following chapter shows how we can reduce the differences between the posting list sizes and maintain similarity-based index by improving the sparse hash computation process.

CHAPTER 6

Balancing distributed index partitions

Effective multimedia index partitioning is key for efficient k-NN search. However, existing algorithms are based on document similarity, without partition size or redundancy constraints. As KSVD's goal is to find the principal directions of data in the original space, the posting lists that correspond to the first directions will have more documents. DISH balancing works on KSVD-based indexes by grouping larger and smaller posting lists together so that each node has the same number of documents (i.e., the sum of documents from all posting lists).

This unbalance becomes a problem when partitions are distributed across multiple nodes: at retrieval time, nodes with the larger posting lists will still spend more time on candidate selection, as the number of candidates to inspect is tied to the pruning factor. The question becomes *is it possible to balance the number of documents across posting lists without a negative impact on retrieval effectiveness?*

Figure 6.1 illustrates the difference between balancing at node level and balancing at posting list level. For an unbalanced system, the number of documents may vary greatly across nodes and partitions. In Chapter 5, DISH can achieve a balanced distribution of documents across nodes, for algorithms that generate posting lists with very large differences in size. This chapter proposes balancing the distribution of documents at both posting list and node level.

The main contribution of this paper is the B-KSVD algorithm which balances the allocation of documents across an adjustable number of dictionary atoms. It was designed to address the even distribution of data across atoms to achieve better load-balancing when allocating data to nodes while maintaining KSVD's key property of grouping similar documents the same partitions. B-KSVD works by reducing the magnitude of dictionary atoms that have more documents assigned.

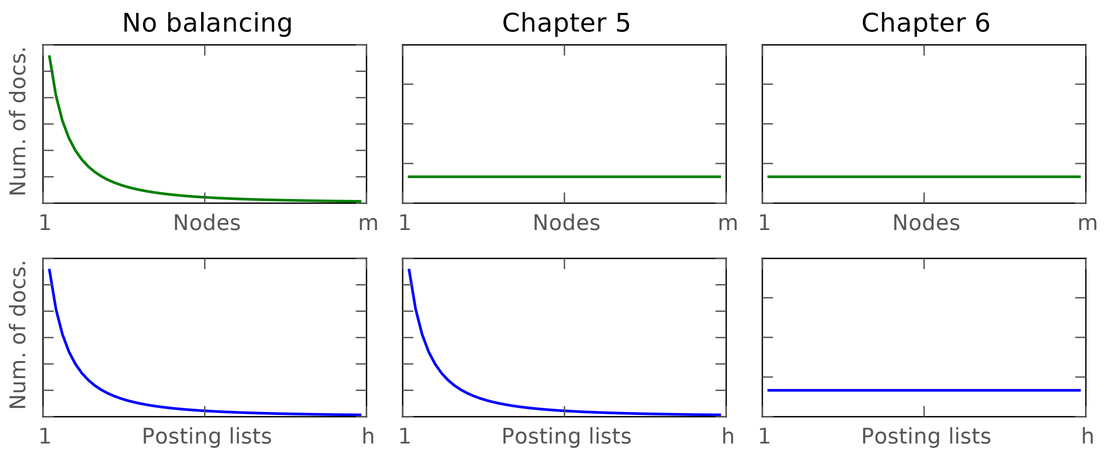


Figure 6.1: Expected distribution of the number of documents across nodes and partitions, for multiple document balancing techniques.

The remainder of this chapter is organized as follows: Section 6.1 formalizes overcomplete redundant partitioning and details the proposed solutions. Section 6.2 describes and discusses the experiments and section 6.4 contains the conclusions.

6.1 Space Partitioning Codebooks

Existing works are designed for similarity-based partitioning or on pure load balancing, with little overlap. Figure 6.2 illustrates how unbalanced partition sizes can become a problem during retrieval (top of the figure). The top of the figure shows an index created using DISH with a KSVD dictionary. The bottom of the figure shows an index created using DISH using a dictionary that generates posting lists with more uniform sizes, which leads to smaller differences in load across nodes. Balancing partition sizes while preserving similarity may appear contradictory: *if the data on a given space is not uniformly distributed, how can one guarantee a fair partitioning of space in both the densely and sparsely populated regions?*

This chapter proposes the incorporation of partition size balancing and redundancy at codebook level. An index composed of balanced, redundant partitions enables a flexible distributed retrieval process. Inspecting multiple partitions leads to incremental retrieval performance increases. Conversely, not inspecting a partition (e.g., node failure), should result in a small retrieval performance loss, instead of no results returned. Another important factor is how does a distributed retrieval system deal with parallel query streams. On use-cases with more query streams, having more uniformly sized partitions is beneficial: queries will be distributed across more partitions, and uniform partition sizes guarantee that the expected partition inspection time is more uniform. To create representations that fit this paradigm, the following properties were identified:

- generate **partitions** that group **documents** that are similar in the original space;

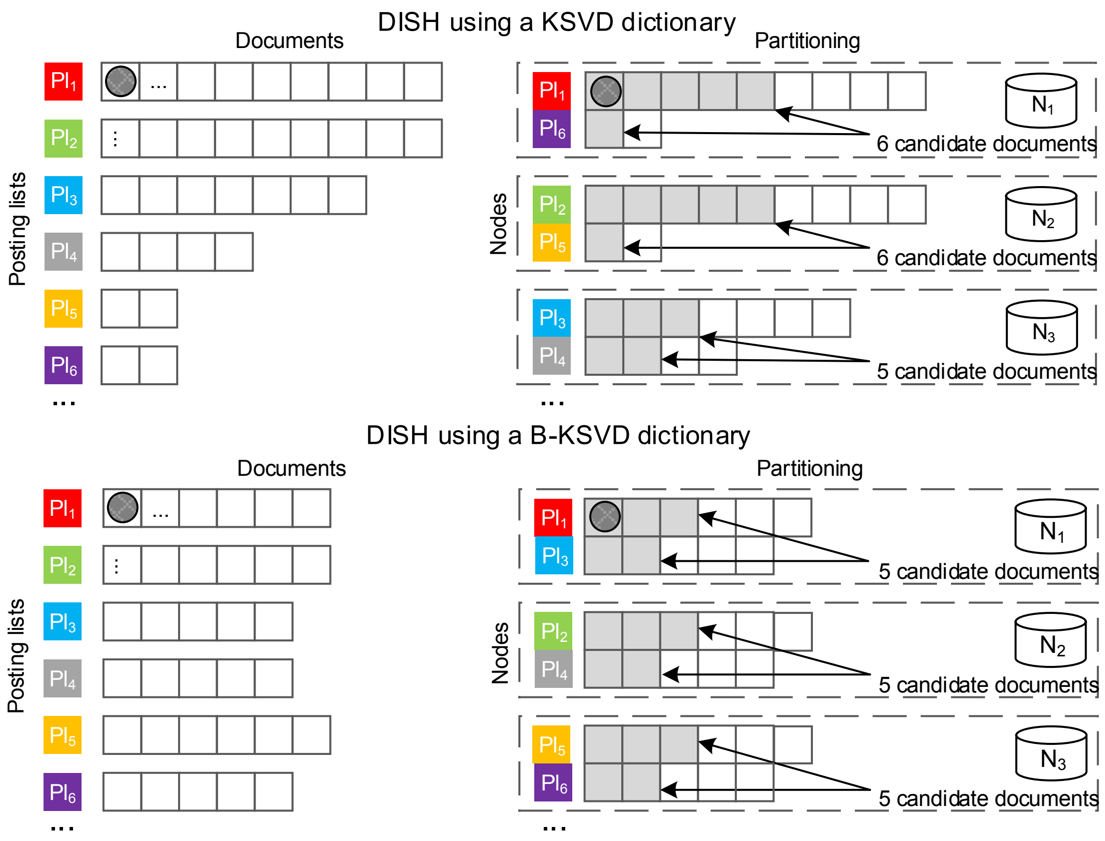


Figure 6.2: DISH index partitioning with KSVD (top) and the proposed method, B-KSVD (bottom). Gray squares represent candidate documents to inspected under a 55% per-partition pruning factor.

- generate **evenly sized** partitions.
- setting a fixed atom **over completeness**, i.e., sparsity factor;
- compute partition **membership magnitude** (e.g., distance to centroid, reconstruction weight) to allow candidate selection inside partitions;

Formally, consider the original vector $y \in \mathbb{R}^n$, a sparse vector $x \in \mathbb{R}^h$ and a sparsity coefficient s. From a set of m, n-dimensional vectors in the original space as $Y \in \mathbb{R}^{m,n}$, the goal is to find a function f that transforms them into h-dimensional sparse vectors as $X \in \mathbb{R}^{m,h}$:

$$\begin{gathered} f(y) = x, \\ \text{where } \|x\|_0 = s \text{ and } s \ll n \ll h. \end{gathered} \tag{6.1}$$

$\|...\|_0$ is the l_0 pseudo-norm: the sparse vector x must have exactly s non-zero values/-coefficients. Forcing sparsity to be equal to the sparsity factor s, instead of the general constraint of smaller or equal, ensures that each document will be placed exactly on s partitions. For a set of vectors $y_a, y_b, y_c \in \mathbb{R}^n$ and corresponding sparse vectors $f(y_a) = x_a, f(y_b) = x_b, f(y_c) = x_c \in \mathbb{R}^h$, the goal is to generate sparse vectors with the following property:

$$\begin{gathered}\text{if } \|y_a - y_b\|_2 < \|y_a - y_c\|_2 \text{ then} \\ \|x_a + x_b\|_0 < \|x_a + x_c\|_0\end{gathered} \tag{6.2}$$

In the above expression, Equation 6.2, vectors that are close in the original space have non-zero coefficients on similar positions in the sparse vector space than vectors that are further apart. Sparse vectors are the basis to generate a set of partitions P, $p \in P \subset Y$. Each sparse vector position corresponds to a partition, and the value at that vector position quantifies the "membership magnitude" to the partition. This chapter's balancing goal is to minimize the differences in partition sizes:

$$|p_a| - |p_b|, \forall p_a, p_b \in P. \tag{6.3}$$

The previous chapter identified and tested two families of methods that have the potential to meet the desired properties: sparse coding and clustering. Sparse coding techniques are designed to generate overcomplete representations of the search space: the reasoning is that codebook atoms can act as the basis of the partitions. For clustering techniques, centroids and distance to centroids act as codebook and atoms respectively, using soft clustering for redundant partitioning. The following sections detail how they were applied for balanced sparse hash computation.

6.1.1 Codebooks by sparse hashing

Sparse vectors can be high dimensional sparse hashes, generated using a codebook representative of the original space. Sparse hashes offer some advantages over binary hashes for search space partitioning: sparse coding techniques are designed to be overcomplete, real-valued membership (i.e., representative values on the non-null dimensions of the sparse hash) and control over the sparsity of the solution and thus, redundancy. Another advantage is that these techniques work on non-uniform or unbalanced feature spaces; codebooks are learned using feature-specific training data, meaning that they will always follow document distribution. The effectiveness of distribution is not limited to feature orthogonality: KSVD will index document on redundant non-orthogonal partitions, maintaining similarity-based indexing guarantees. The sparse hash generation steps are:

- compute the dictionary/codebook D from training data;
- use D to create an hash with s non-zero coefficients and assign them to the corresponding partitions;
- for search, inspect the s partitions corresponding to non-zero coefficients.

The process for the generation of sparse hashes that follow Equation 6.1 goals, is to solve the following optimization problem:

$$\begin{gathered}\arg\min_x \|Dx - y\|_2, \\ \text{subject to} \\ \|x\|_0 = s\end{gathered} \tag{6.4}$$

where $D \in \mathbb{R}^{h \times d}$ is a dictionary, learned from the data, $y \in \mathbb{R}^d$ is the the original vector, $x \in \mathbb{R}^h$ is the sparse hash and s is the sparsity coefficient. Note that x was set to be equal to s, instead of the usual smaller or equal requirement, which differs from the sparse hash processing described in Section 4.2. This was to ensure that generated hashes respect the redundancy goals (fixed number of partitions per document). Equation 6.4 generates a hash with the desired properties, using a previously computed dictionary. Techniques for dictionary computation include KSVD [2] and Stochastic Gradient Descent techniques.

6.1.1.1 KSVD and OMP

Equation 6.4 shows how to generate a sparse hash x for a vector y, based on an (existing) dictionary D. Thus, one must first generate the dictionary D that adequately represents documents in the original space. Computing the dictionary requires solving the following optimization problem: find the dictionary D and set of sparse hashes X that minimizes the reconstruction error against a set of vectors Y:

$$\begin{gathered}\arg\min_{D,X} \|DX - Y\|_2, \\ \text{subject to} \\ \|x\|_0 = s, \\ \text{for } x \in X\end{gathered} \tag{6.5}$$

where $Y \in \mathbb{R}^{n \times d}$ are the original document vectors, $D \in \mathbb{R}^{h \times d}$ is a dictionary, to be learned from the data, $X \in \mathbb{R}^{h \times n}$ are the sparse hashes (one per column), x is a sparse hash vector (column of X) and s is the sparsity coefficient.

As it was discused in section 4.2, solving for both D and X is NP-hard. KSVD alternatively optimizes the solution for D and X. KSVD updates each dictionary atom iteratively (represented by i), while fixing other atoms $j_{[0,h]} \neq i$. By decomposing Equation 6.5 into KSVD iterative process, one arrives at the following formulation, for the iteration where the atom i is fixed:

$$\begin{gathered}\arg\min_{D_i,(x_i)_I} \|D_i(x_i)_I + (E_i)_I - Y\|_F^2 \\ E_i = \sum_{j_{[0,k]} \neq i} \|D_j x_j - Y\|_F^2\end{gathered} \tag{6.6}$$

where $\|...\|_F$ is the Frobenius norm, and E_i is the reconstruction error for the (fixed) atoms, $j_{[0,h]} \neq i$. Sparsity is enforced by using only the dimensions with non-zero coefficients: I is the set of all index with non-zero coefficients that use atom i for reconstruction.

By fixing j atoms, the value for atom D_i can be computed by finding a rank-1 matrix approximation of E_i, $\hat{E}_i$, and factorizing the result into D_i and x_i.

$$\hat{E}_i = G\sum^{1} V^T \tag{6.7}$$

This decomposition will yield D_i as the first column of G and x_i as the first column of $V \times \sum^1$.

6.1.1.2 Balanced KSVD

KSVD enforces the creation of sparse representations that group similar vectors in the original space on atoms with non-zero coefficients. When generating multiple sparse hashes, KSVD will inherently create unbalanced representations, as the dictionary atoms are biased towards the principal directions of the data on the original space. Babenko and Lempitsky [11] already showed how relaxing orthogonality constraints reduces the number of non-empty partitions. The goal is to take it further, and minimize the standard deviation σ in the distribution of documents across partitions:

$$\begin{gathered} \text{arg min}_{D,X} \|DX - Y\|_2 + \sigma(\|X^T\|_0), \\ \text{subject to} \\ \|x\|_0 = s, \\ \text{for } x \in X \end{gathered} \tag{6.8}$$

The transposed matrix X^T combined with the l_0 pseudo-norm as $\|X^T\|_0$, represents a vector with h elements, containing the number of documents per atom. It contrasts with the sparsity constraint $\|x\|_0 \leq s,$for $x \in X$, which represents the number of atoms per document. Thus, $\sigma(\|X^T\|_0)$ is the standard deviation in the number of documents per partition.

To achieve this goal, B-KSVD reduces the magnitude of dictionary atoms that have more documents assigned. Its alternate optimization process is similar to Equation 6.6; balancing is applied to Equation 6.7's E decomposition; after the rank-1 approximation, the G matrix is multiplied by the penalty factor B:

$$\begin{gathered} \text{balanced } \hat{E}_i = B \times G\sum^{1} V^T \\ B = \frac{1}{\left(\|X_{-1}^T\|_0 + r\right)^e} \end{gathered} \tag{6.9}$$

where X_{-1} represent the hashes computed using the previous iteration of the dictionary. Therefore, $\|X_{-1}^T\|_0$ is a h-dimensional vector, containing the number of documents assigned to partitions on the previous iteration. This formulation matches the number of documents per atom formulation of $\|X^T\|_0$, stated on Equation 6.8. e is the parameter to control the magnitude of the penalty, and r is a regularization factor to avoid division by zero for partitions with zero documents. This penalty distorts the estimation of the dictionary atoms, creating non-orthogonal balanced representations. The regularization parameters r and s control the magnitude of this distortion, balancing between similarity-based indexing and balanced partitions. Figure 6.3 shows what happens when applied to a two-dimensional space. Dictionary atoms that match a large percentage of documents (e.g., red atom) have their magnitudes reduced, and atoms that match few documents have their magnitudes increased. This process distorts the hash space to create atoms that match the original feature space.

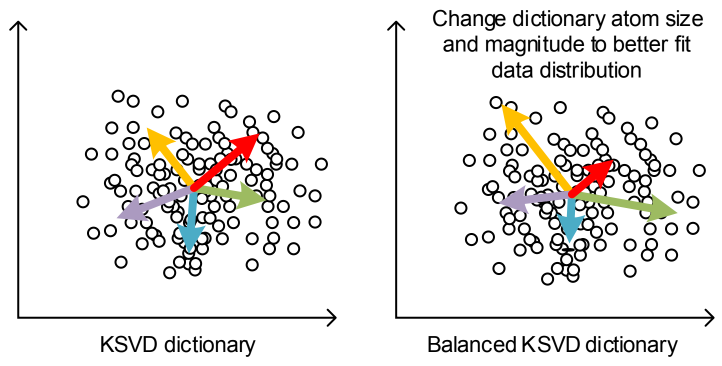

Change dictionary atom size and magnitude to better fit data distribution
KSVD dictionary
Balanced KSVD dictionary

Figure 6.3: B-KSVD dictionary distortion visualization

6.1.1.3 Random dictionary

The impact of dictionary learning on the computation of sparse hashes can also be measured by using a random dictionary. OMP will compute sparse hashes using random atoms, generated from the Gaussian distribution with zero mean and unit standard deviation.

$$D \in \mathbb{R}^{h \times d} \subset \mathcal{N}(0,1) \tag{6.10}$$

Random dictionaries show how OMP clusters data without prior search space information from dictionary computation.

6.1.2 Codebooks by soft quantization

An alternative interpretation of the sparse vector computation process follows soft clustering, where the cluster membership is controlled by a fixed s sparsity factor. Its focus is to measure how well these clusters can represent neighboring data in a balanced way, and how using multiple clusters affects the sparse vector computation process in a high dimensional feature space.

6.1.2.1 k-means centroids

k-means is one the most widely applied clustering functions in nearest neighbor search. It estimates a set of centroids $C \in \mathbb{R}^{h \times d}$ that minimizes the distances between the points to the centroids of their clusters. The k-means clustering process minimizes the following expression:

$$\arg\min_C \sum_{c_i \in C} \sum_{y_j \in P_i} \|c_i - y_j\|_2 \tag{6.11}$$

where $C \in \mathbb{R}^{h \times d}$ is the set of cluster centroids, $P_i, i \in [0, a]$ is the set of documents $y_j \in P$ that are assigned to centroid C_i. The k-means initialization requires the selection of a set of points as the initial centroids. k-means++ [7] was selected for centroid initialization, as it selects points that give a good representation of the search space and lead to faster convergence, on a large set of experiments and datasets.

6.1.2.2 Random centroids

A random sampling technique that selects a random set of points C from the training data Y was also tested for centroid selection:

$$C \in \mathbb{R}^{h \times d} \subset Y \tag{6.12}$$

The sampling process makes no assumptions regarding distribution. The expected behavior is that the algorithm will select more points in denser regions of the training data space. As with the random dictionary with OMP regression, this technique is a baseline to measure the impact of centroid selection for the creation of evenly balanced partitions.

6.2 Experiments

We have described how to create over-complete codebooks that generate sparse, high dimensional hashes. To measure how well the proposed methods meet the stated partitioning and retrieval goals, they were evaluated from three perspectives:

- **Balanced partitioning**: measure how the tested methods manage to balance the size of the partitions;
- **Inter-partition retrieval**: measure the cumulative impact of searching on more than one partition;
- **Intra-partition retrieval**: measure how well the partitions capture the original space nearest neighbours;

The index complexity and memory requirements for this index are similar to DISH, Section 5.1.5.1. The main advantage of this technique of the balanced index is that both computational and memory complexity will be more uniform across nodes.

Dataset: Index partitioning methods were tested on the ANN dataset [48] one million subsets, section 2.6.4. It contains one million descriptors from two feature types: GIST (960 dimensions) and SIFT features (128 dimensions). The datasets training, validation and test splits follow the standard protocol for the dataset[1].

[1] http://corpus-texmex.irisa.fr/

Metrics: In addition to load balancing quality metrics, which are the number of documents per partition p and standard deviation σ of partition size versus the mean, the following retrieval quality metrics were averaged over 1000 queries:

- 1-recall@r: average rate of queries for which the 1-nearest neighbor was returned. r changes with the number of candidates inspected.
- avgP@k: average percentage of true k nearest neighbors retrieved.

Parameters: Based on preliminary experiments, I found that setting the exponent of the penalty to $c = 2$ and regularization factor to $r = 0.001$ offered the best trade-off between similarity and even balancing. The sparsity coefficient was set to $s = 10$ for all algorithms and varied the codebook size h and thus, the number of partitions (512, 1024, 2048, 4096, 8192).

6.2.1 Balanced partitioning

The goal of this experiment is to measure how the selected techniques distribute documents across partitions, for multiple numbers of partitions and feature types. Documents were assigned to the partitions with corresponding non-zero atoms/centroids, for each partition method, feature type and the number of partitions.

Figures 6.4 and 6.5 shows the behavior of the partitioning algorithms for the GIST and SIFT features (left and right side charts respectively) and the number of partitions (different rows). For readability, each chart is divided into two parts: the smaller chart shows the occupation of the top 20 partitions, where the variation in scale of the number of documents is higher. The larger chart shows the variation for the remaining partitions (20 to h). The X-axis represents the partitions, sorted in descending order of the number of indexed documents (i.e., partitions with more documents are to the left). The Y-axis represents the number of documents on that partition. Note that, as the goal is to show the relative differences between partitioning methods, the Y-axis scale is different across charts. It is also important to note that the sum of the sizes of the partitions is the same for all partitioning methods (index size $n \times s$).

Table 6.1 shows the detailed standard deviation (σ), larger partition (Max), and median (Med) partition size ($h/2$). KSVD learns a dictionary with the principal directions of the data in the original space. Combined with OMP greedy atom selection, KSVD sparse representations are highly biased towards principal directions, which is clear on the top 20 charts. B-KSVD managed to counteract KSVD's greediness and generated the most balanced solutions (σ columns on Table 6.1). This effect is clearer at the partitions with the larger and smaller partitions: on the top 20 positions, B-KSVD is less affected than KSVD, by the most popular directions of the data; the occupation of the partition at median value is also consistently closer to the expected value (Mean) than other methods, meaning the decrease in the number of documents is much slower and gradual than the

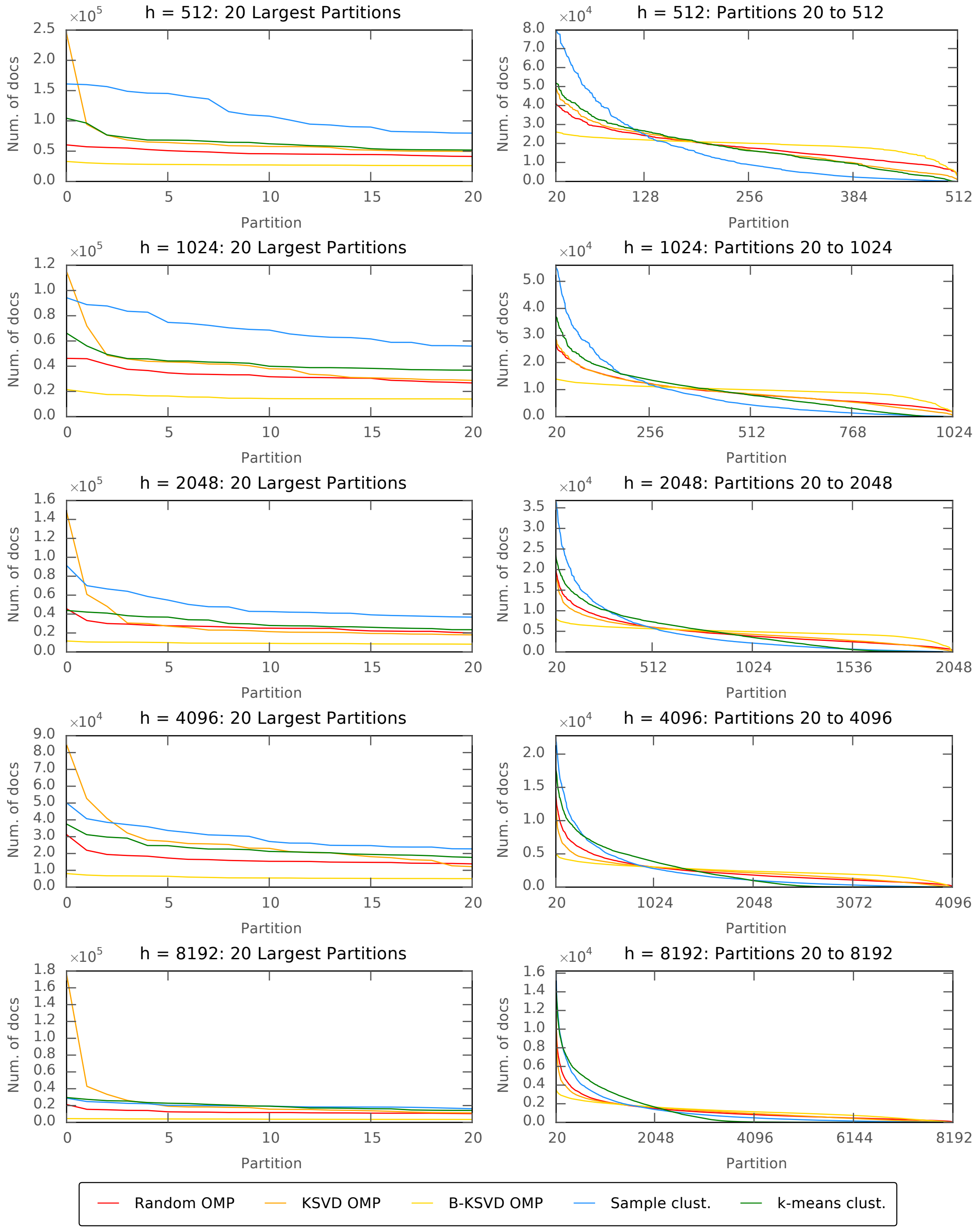


Figure 6.4: Sorted partition size distribution on GIST features for multiple numbers of partitions. The charts on the leftmost column show the top 20 partition sizes, and the charts on the rightmost column show the remaining partition sizes.

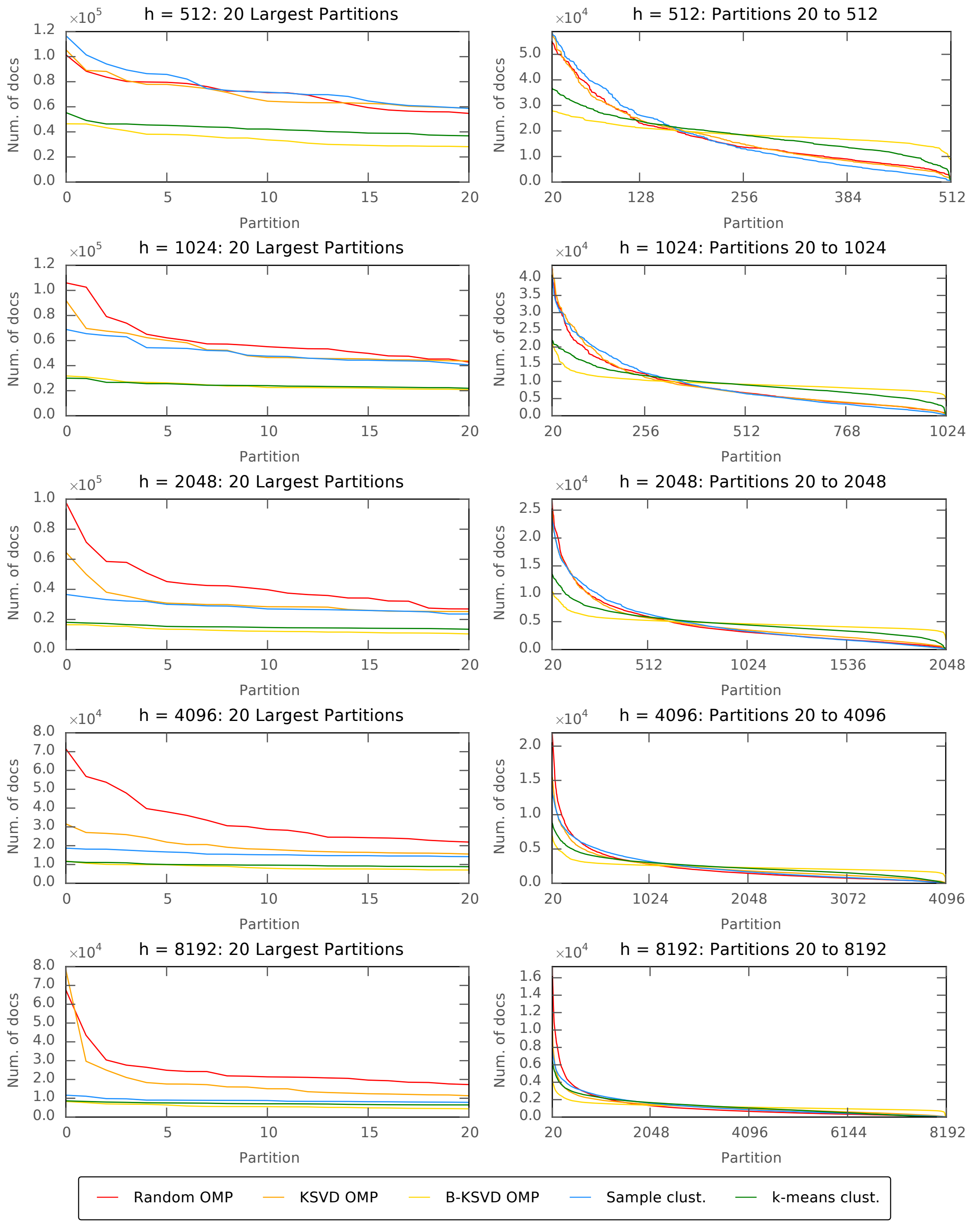


Figure 6.5: Sorted partition size distribution on SIFT features for multiple numbers of partitions. The charts on the leftmost column show the top 20 partition sizes, and the charts on the rightmost column show the remaining partition sizes.

Table 6.1: Partition balancing results: Max is the size of the largest partition (bold values: lowest is best), Med is the size of the median partitions, partition size / 2 (bold values: closest to mean is best), and σ is the standard deviation of the partition sizes (bold values: lowest is best). The Mean value is the same for all methods for a set partition size, as all methods produce solutions with similar sparsity s. Mean, Max, Med and σ values are on base 10^3.

Partitions:	512			1024			2048			4096			8192		
Mean:	**19.5**			**9.8**			**4.9**			**2.4**			**1.2**		
GIST															
Algorithm	Max	Med	σ	Max	Med	σ	Max	Med	σ	Max	Med	σ	Max	Med	σ
Random OMP	60.4	17.6	9.7	46.2	8.2	6.0	45.6	3.8	3.9	31.4	1.8	2.2	21.2	0.8	1.3
KSVD OMP	245.9	16.7	16.4	115.1	8.4	7.6	148.0	4.3	4.8	85.1	2.1	2.6	176.2	0.9	2.4
B-KSVD OMP	**33.0**	**20.1**	**4.2**	**21.5**	**9.9**	**2.4**	**11.5**	**4.9**	**1.3**	**8.1**	**2.4**	**0.9**	**4.5**	**1.1**	**0.7**
Sample clust.	160.9	8.9	27.0	94.3	4.3	13.7	91.4	2.1	7.6	50.2	1.0	3.9	28.8	0.5	2.1
k-means clust.	104.4	16.2	14.9	66.2	8.0	8.9	43.8	3.5	5.3	37.6	1.0	3.4	29.6	0.0	2.3
SIFT															
Algorithm	Max	Med	σ	Max	Med	σ	Max	Med	σ	Max	Med	σ	Max	Med	σ
Random OMP	101.4	13.7	15.9	106.0	6.7	10.3	97.6	3.1	6.0	71.5	1.4	3.6	67.6	0.7	2.1
KSVD OMP	105.4	14.9	16.0	91.9	6.5	9.9	64.5	3.5	4.8	31.6	1.8	2.4	78.2	1.0	1.5
B-KSVD OMP	**46.4**	**18.5**	**4.8**	**31.8**	**9.1**	**3.0**	**16.5**	**4.6**	**1.4**	11.8	**2.3**	**0.8**	**8.3**	**1.1**	**0.5**
Sample clust.	116.6	13.0	18.2	68.8	6.5	9.7	36.6	3.3	4.8	18.7	1.7	2.4	11.6	0.8	1.2
k-means clust.	55.3	18.4	8.4	30.0	9.0	4.4	18.1	4.4	2.4	**11.6**	2.2	1.5	8.6	1.0	1.0

other retrieval methods tested. B-KSVD is also the most stable solution, offering the best balancing properties for all partition sizes and feature types.

k-means performance is greatly affected by feature type. For SIFT features, k-means partition size balancing is in line with B-KSVD for the top 20 positions, with a faster decay in the number of documents on the smaller partitions. For GIST features, the unbalanced distribution is more clear and appears earlier (top 20).

The impact in balancing partition sizes of the data-dependent approaches (k-means, KSVD and B-KSVD) versus random and sampling techniques was also tested. OMP with the random dictionary balancing varied greatly for the type of features used: for GIST, it is in line with k-means; for SIFT it has the most unbalanced distribution of all tested methods (e.g., some partitions have over 1/8 of the total number of indexed documents). Sample clustering also shows large unbalances, where larger partitions clustered most of the documents. The large balancing variations for these methods shows that adjusting codebooks to the data has a large impact on balancing partitions.

The impact of the number of partitions is also clearly visible. The tested partitioning methods are not designed to handle a higher number of partitions, generating a large number of very small or empty partitions (visible on the right side of X-axis of Figures 6.4 and 6.5). The exception is B-KSVD, that managed to keep evenly sized partitions, regardless of the number of partitions.

These experiments showed how different partitioning methods distribute documents across partitions. B-KSVD countered the greedy nature of regular KSVD and offered the most uniform partitions. On the following sections, we will show how it affects the retrieval performance.

6.2.2 Searching redundant partitions

Balancing partition sizes is only desirable if it does not degrade retrieval performance. In this section, we will measure the retrieval impact of searching on over-complete partitions. An advantage of real-valued over binary hashes is that sparse hash values represent the document-partition membership likelihood. By having a measure of membership of the documents and queries to partitions, one can prioritize candidate selection at partition or global level.

Intra-partition search: This experiment shows the retrieval performance of individual partitions, Figure 6.6. For each query, 1000 candidates (i.e., 0.1% of total index size) were selected from each corresponding partition, for a combined limit of 1%. The order of partition inspection is defined by the coefficients (OMP-based approaches) or the inverse of the distance to the centroid (centroid-based approaches).

B-KSVD offers the best results on the first partition (i.e., higher membership) for GIST partitions (14% of 50 nearest neighbors, examining, 1000 documents, i.e., 0.1% of the index). The number of nearest neighbors decreases for lower membership partitions.

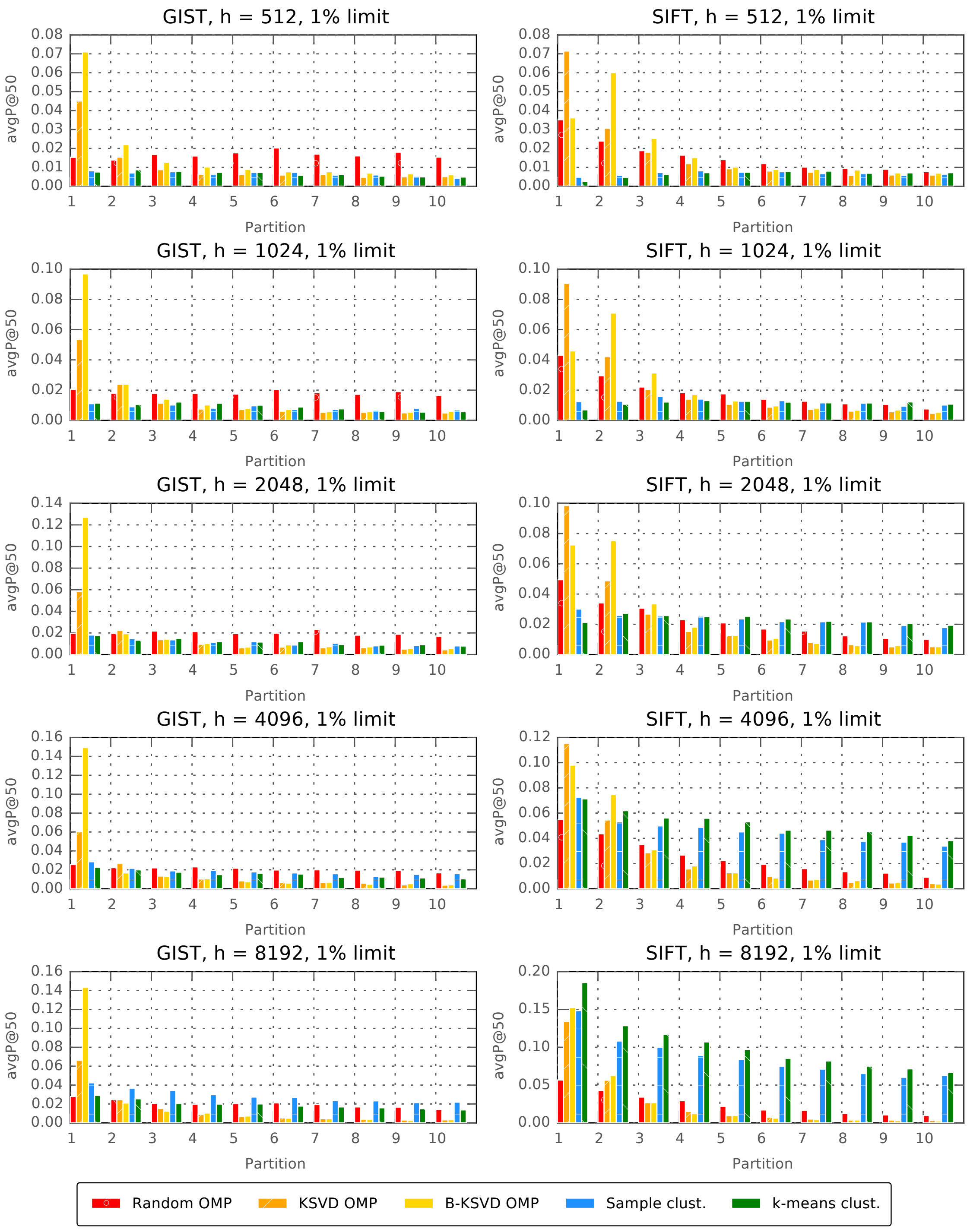


Figure 6.6: Inter-node partition: avgP@50 of individual partitions, for multiple feature types and number of partitions

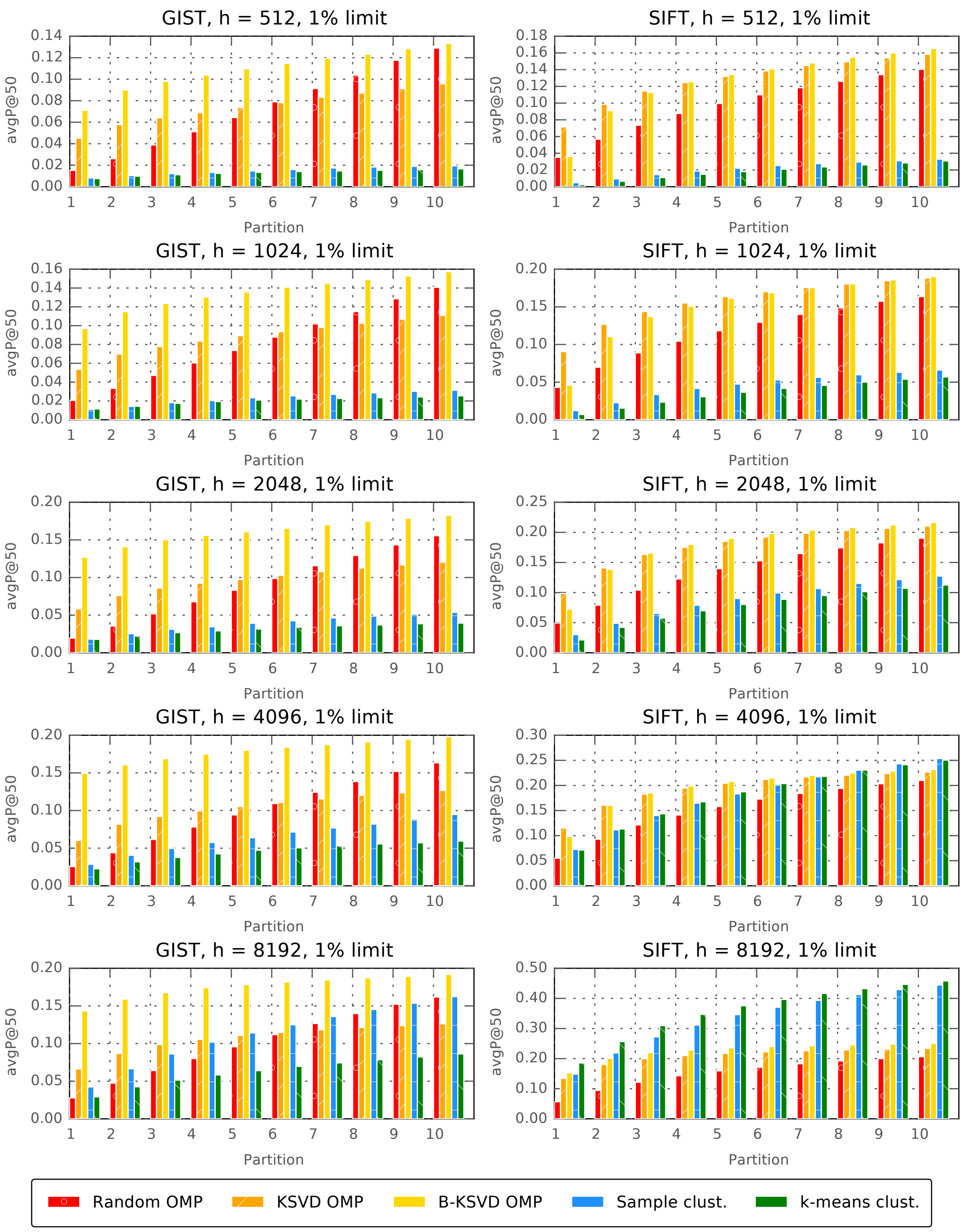


Figure 6.7: Cumulative inter-node partition: global avgP@50, for multiple feature types and number of partitions

Table 6.2: Intra-node, cumulative retrieval results for 1% and 10% global search limits ($\mathbf{1 \times 10^3}$ and $\mathbf{10 \times 10^3}$ candidates per partition, respectively)

Partitions:	512		1024		2048		4096		8192	
GIST: 1% limit										
Algorithm	avgP@50	1-recall	avgP@50	1-recall	avgP@50	1-recall	avgP@50	1-recall	avgP@50	1-recall
Random OMP	**0.13**	0.20	0.14	0.22	0.16	0.26	0.16	0.27	0.16	0.28
KSVD OMP	0.10	0.15	0.11	0.19	0.12	0.21	0.13	0.20	0.13	0.21
B-KSVD OMP	**0.13**	**0.23**	**0.16**	**0.26**	**0.18**	**0.27**	**0.20**	**0.29**	**0.19**	**0.30**
Sample clust.	0.02	0.03	0.03	0.05	0.05	0.09	0.09	0.15	0.16	0.25
k-means clust.	0.02	0.03	0.03	0.05	0.04	0.06	0.06	0.10	0.09	0.13
SIFT: 1% limit										
Algorithm	avgP@50	1-recall	avgP@50	1-recall	avgP@50	1-recall	avgP@50	1-recall	avgP@50	1-recall
Random OMP	0.14	0.20	0.16	0.23	0.19	0.27	0.21	0.36	0.21	0.30
KSVD OMP	0.16	**0.24**	**0.19**	**0.33**	0.21	0.33	0.23	0.37	0.23	0.37
B-KSVD OMP	**0.17**	**0.24**	**0.19**	0.27	**0.22**	**0.34**	0.23	**0.38**	0.25	0.40
Sample clust.	0.03	0.05	0.07	0.11	0.13	0.19	**0.25**	0.37	0.44	**0.59**
k-means clust.	0.03	0.05	0.06	0.10	0.11	0.16	**0.25**	0.35	**0.46**	**0.59**
GIST: 10% limit										
Algorithm	avgP@50	1-recall	avgP@50	1-recall	avgP@50	1-recall	avgP@50	1-recall	avgP@50	1-recall
Random OMP	0.53	0.62	**0.49**	**0.60**	0.42	0.56	0.34	0.44	0.28	0.41
KSVD OMP	0.41	0.50	0.38	0.51	0.33	0.47	0.29	0.39	0.26	0.35
B-KSVD OMP	**0.57**	**0.69**	0.47	**0.60**	0.36	0.50	0.28	0.39	0.22	0.33
Sample clust.	0.21	0.28	0.37	0.46	0.54	0.65	0.68	0.78	0.72	0.82
k-means clust.	0.21	0.27	0.44	0.53	**0.66**	**0.74**	**0.76**	**0.85**	**0.80**	**0.89**
SIFT: 10% limit										
Algorithm	avgP@50	1-recall	avgP@50	1-recall	avgP@50	1-recall	avgP@50	1-recall	avgP@50	1-recall
Random OMP	0.63	0.75	0.61	0.72	0.56	0.67	0.50	0.66	0.43	0.54
KSVD OMP	0.65	0.77	0.62	0.75	0.56	0.69	0.50	0.62	0.41	0.52
B-KSVD OMP	**0.67**	**0.78**	0.63	0.74	0.60	0.69	0.51	0.64	0.44	0.58
Sample clust.	0.43	0.51	0.69	0.79	0.89	0.95	**0.95**	0.98	0.92	0.98
k-means clust.	0.46	0.56	**0.84**	**0.90**	**0.96**	**0.99**	**0.95**	**0.99**	**0.93**	**0.99**

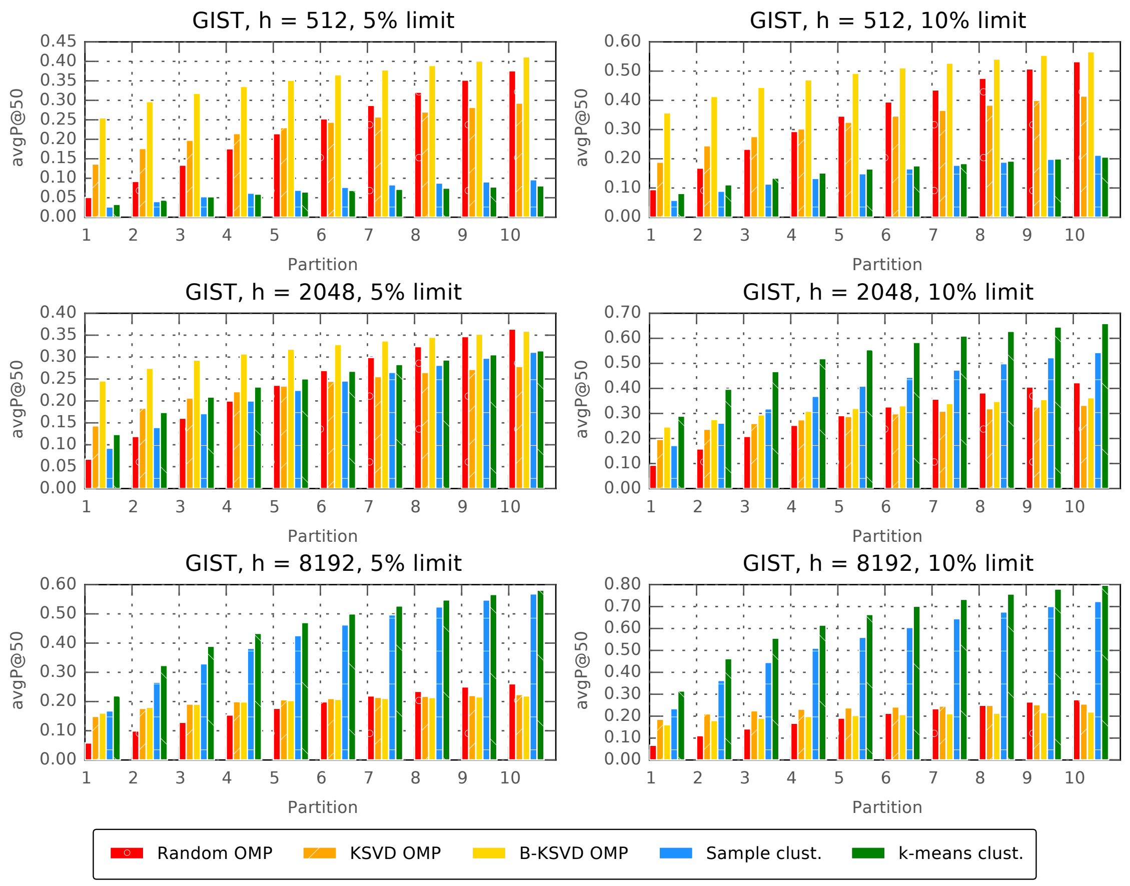


Figure 6.8: Cumulative inter-node partition: global avgP@50, for GIST features and multiple limits

The impact of the remaining partitioning methods is in the order of 2% of the 50 nearest neighbors, for a 0.1% search limit.

For SIFT, the partition results show a different pattern. KSVD and B-KSVD also retrieve the most results on the top membership positions, for all but the experiments with 8192 partitions. For larger numbers of partitions, clustering-based solutions offer better results.

Inter-partition (global) search: Table 6.2 shows the aggregated results for the search process. From each partition, 0.1% and 1% were selected from the total index size. With the (fixed) sparsity factor s set to 10, the combined limit is 1% and 10% of total index size, respectively.

The advantages of KSVD based methods are clear on the limited search conditions (inspecting 1% of the index). When using smaller search inspection limits, the reconstruction coefficient represents similarity in the original space better than the distance to cluster centroids. For larger search inspection limits (10%) and more partitions, clustering methods can retrieve a larger set of candidates. The same findings are also valid for 1-recall results. The main difference is that they are slightly higher that avgP@50 results. This means that both sparse coding and clustering methods can index the first nearest neighbor

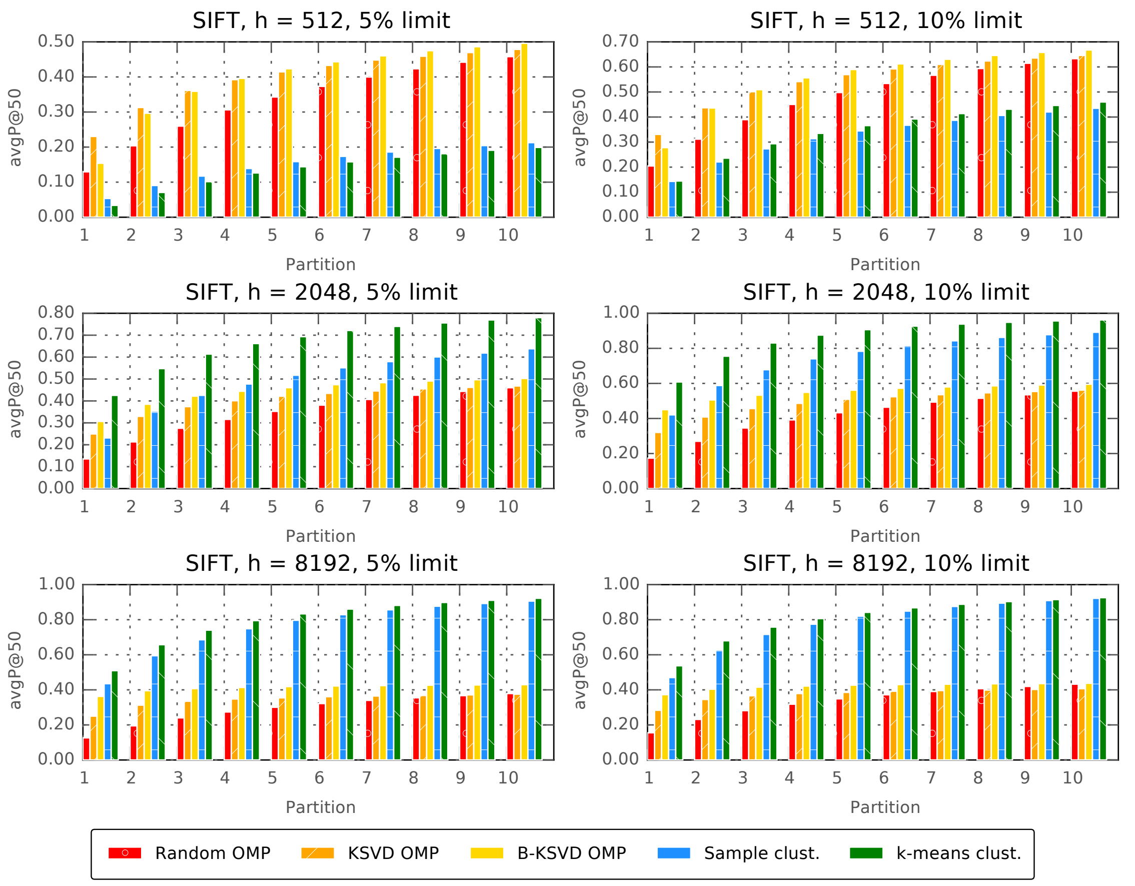


Figure 6.9: Cumulative inter-node partition: global avgP@50, for SIFT features and multiple limits

at higher rates than the remaining 49 nearest neighbors.

Incremental inter-partition search: Figures 6.7, 6.8 and 6.9 detail Table 6.2 results, by showing the incremental avgP@50 gains of inspecting all 10 partitions. Figure 6.7 shows the results for a 1% search limit; Figures 6.8 and 6.9 show what happens when increasing the search limit to 5% and 10% of index size. As with Figure 6.6 results, partitions are inspected by coefficient or inverse distance-to-centroid order.

These results show some interesting differences in retrieval effectiveness. For GIST features, B-KSVD achieves good avgP@50 when inspecting one partition with 1%-5% search limits. It is also clear that there is only an average difference in avgP@50 between 5 and 10% when inspecting one versus all (10) partitions, regardless of partition size. For SIFT features, the behavior is similar, with higher avgP@50 gains when inspecting more partitions. KSVD behaves similarly, but with lower initial avgP@50 on most parameter configuration. Random OMP follows a trend more similar to clustering based approaches: the initial avgP@50 is lower, and the gains from inspecting more partitions are higher.

On k-means and other clustering-based techniques, having more partitions increases retrieval performance. This property is more evident when dealing with more partitions and using higher search limits.

As both codebook and clustering methods are based on greedy atom selection, adding the document to more partitions does not affect the results for previous partitions. Thus, the expected retrieval performance of setting the value of the redundancy factor to between one and ten is visible by inspecting precision levels at those levels in Figures 6.7, 6.8 and 6.9.

6.3 Discussion

The previous section showed how search space partitioning algorithms affect the balanced distribution of documents across partitions, and the corresponding impact on retrieval performance. This fulfills the initial goal of this chapter: create balanced search space partitions for distributed indexing and retrieval. Experiments show that B-KSVD can work for the partitioning of a distributed search index better than the baseline methods.

A clear pattern across experiments is that sparse coding-based techniques have better precision with lower search limits and a smaller number of partitions, while centroid based approaches have better results when searching with higher limits and more centroids. This is due to centroid-based techniques being based on selecting points in the original space. Selecting more centroids increases the probability of getting better coverage of the search space. This property also results in having the nearest neighbors more spread out over more partitions, resulting in higher gains when inspecting more partitions. This is visible when dealing with SIFT features and a large number of partitions (e.g., 8192). It is also clear how centroid selection as a non-negligible impact on retrieval performance.

This contrasts with dictionary-based approaches, which transform the original feature space into a new space, based on the principal directions of the original space. Gains in performance decrease as one inspects the hash dimensions with a smaller reconstruction coefficients. B-KSVD achieves good distribution and works well for low search limits (1% of total index size), and a small number of partitions (512 to 1024). This is an interesting property to answer concurrent queries on distributed indexes: a more balanced distribution means that the probability of querying the same node for multiple queries decreases.

k-means works better for higher limits and number of partitions, due to being based on working in the original search sparse. It offers more predictable, linear performance increase, at the cost of by having larger partitions.

6.4 Conclusion

This chapter proposes B-KSVD, a codebook learning technique for the creation of balanced, over-complete search partitions. It formalized the requirements to create overcomplete representations with redundant document indexing, where partitions contain overlapping subsets of data. The proposed representations are based on sparse coding and clustering models and on an adaption to the KSVD algorithm, B-KSVD, that distributes hash values across positions, according to the global distribution.

Experiments showed that computing codebooks that penalize larger partitions creates more balanced partitions, and has a positive retrieval impact.

CHAPTER 7

Conclusions

7.1 Achievements

The goal of this thesis was to research multimedia search engine architectures that address the specific requirements of large-scale multimedia indexing in a distributed environment.

The proposed vertical partitioning technique, MVP, partitions indexes across nodes by their feature spaces, adding little overhead in the distribution and retrieval process.

As vertically partitioned indexes are limited to the resources of individual nodes, I proposed a set of techniques to effectively partition and distribute single feature indexes to multiple nodes. Thus, I proposed index distribution at two levels:

- **Node-level**: DISH is able to generate the best possible load distribution across nodes over a set of very unbalanced partitions;
- **Partition-level**: B-KSVD can generate evenly sized index partitions based on document similarity, with a positive impact on retrieval.

From an evaluation perspective, I performed extensive performance evaluations of all components, both from an effectiveness and an efficiency point of view:

- The deployment of a one billion feature vector DISH index on a commercial cloud provider, with exhaustive benchmarking over a varying number of nodes and query streams;
- B-KSVD results show that balancing can work at partition level and even bring retrieval performance improvements when compared to traditional, unbalanced approaches;
- Rank fusion experiments showed that L2F techniques achieved better results that state-of-the-art LETOR approaches when working with limited expert domain data.

One of the main findings of this thesis is the that balancing partition sizes is not diametrically opposed to content-based indexing. Moreover, incremental redundancy shows that it is possible to index documents in partitions with different sets of nearest neighbors, giving both retrieval performance and redundancy improvements.

Rank fusion approaches were applied to combine results from a vertically partitioned index. This thesis proposes a rank fusion approach, L2F, that can combine results across modalities, improving retrieval results from individual partitions. L2F selects the best features by measuring the incremental improvement on retrieval performance on datasets with limited training data. L2F can improve query time and efficiency, as it does not need to query all indexes to achieve top retrieval performance.

7.1.1 Clinical and federated IR evaluation campaigns

In addition to the evaluation performed on this thesis, the proposed techniques were tested on applied retrieval competitions such as ImageCLEF and TREC. The following list shows the corresponding Working Notes and noteworthy results:

- André Mourão, Flávio Martins and João Magalhães, *NovaSearch on Medical ImageCLEF 2013*, Working notes on ImageCLEF 2013. **Best multimodal case-based retrieval performance**;
- André Mourão, Flávio Martins and João Magalhães, *NovaSearch at TREC 2013 Federated Web Search Track: Experiments with rank fusion*, Working notes on TREC 2013. **Best NDCG@20** for the rank list fusion task;
- André Mourão, Flávio Martins and João Magalhães, *NovaSearch at TREC 2014 Clinical Decision Support Track*, Working notes on TREC 2014. **Best P@10** and Top 3 infNDCG;
- André Mourão, Flávio Martins and João Magalhães, *NovaSearch at TREC 2015 Clinical Decision Support Track*, Working notes on TREC 2015;
- Gonçalo Araújo, André Mourão and João Magalhães, *NovaSearch at TREC 2017 Clinical Decision Support Track*, Working notes on TREC 2017;

7.1.2 Impact

The long-term vision of this thesis is a multimodal federated search engine that can learn across documents multiple modalities and represented as a very diverse set of features. As application of such system is a clinical decision support system that can gather data from multiple sources (e.g., biomedical articles, medical imaging datasets focused on a family of diseases, blood work data, structured thesaurus, and coding systems, etc.) and provide relevant insights to current cases in the form of similar cases, possible diagnosis, suggested treatments and tests. This thesis takes the first steps towards this vision, by the creation of a generic indexing and retrieval architecture that can deal with these heterogeneous features and with the scale that comes from the support for such complex documents. The following list shows the current and potential applications of the proposed methods:

- Bio-medical image retrieval: a system that takes a short patient case report (including text, images, and demographic information) and returns a list of most relevant articles in the literature. (*NovaMedSeach*);
- Video search and summarization: find similar video scenes for video composition and summarization (*NovaVidSearch*);
- Duplicate image search: DISH and B-KSVD could be applied to improve the efficiency of near-duplicate search engines such as Google Reverse Image Search or TinEye;
- Federated web search: aggregate results from multiple search engines from multiple domains or using varied set of features from multiple modalities, to generate a more diverse set of results (as intended in the TREC Federated Web task);

7.2 Limitations

The proposed architecture is based on the assumption that feature vectors can adequately model documents and that load distribution follows the distribution of indexed documents. Thus, the following points show its potential limitations:

- Balancing the number of documents across nodes does not guarantee that the load will be perfectly balanced across nodes. Query distribution can be affected by factors which are external to document distribution such as seasonal patterns or large scale events that distort the search space Jin et al. [51];
- Sparse hashing approaches are dependent on a final l_2 candidate sort, which takes a non-negligible amount of processor node time;
- As with existing approximate nearest neighbor retrieval systems, SHI and DISH improve retrieval temporal performance at the cost of precision when compared to an exhaustive search process;
- Retrieval effectiveness is limited by the gap between human and computational perception of document similarity. In this thesis, this is mitigated by using multiple modalities and features for retrieval.

7.3 Future work

The following topics describe potential research ideas that arose during this thesis:

- Hierarchical distributed sparse hashing: replace posting lists and linear search at intra-node level with another inverted index structure better fitted to the documents assigned to each partition (e.g., a per-partition SHI);
- Rank fusion to combine results from a horizontally partitioned index: research on the possibility of replacing the costly l_2 candidate sort in the original feature space using rank fusion;
- Incorporate document age and other statistics into the codebook. For example, should the moment where documents are added to the index influence the indexing process? (e.g., documents that arrive at the same time are indexed together);
- Fine-grained control of the search process: choose which nodes to query, based on current node load and the document to partition membership;
- Test other types of numerical feature vector data: bio-medical data such as blood work, temperature, blood pressure, activity data; or mapping-based data: road and points coordinates, among others.

Bibliography

[1] M. Abedini, L. Cao, N. Codella, J. H. Connell, A. Geva, M. Merler, Q.-b. Nguyen, and M. Carmel. "IBM Research at ImageCLEF 2013 Medical Tasks". In: *Working Notes of CLEF 2013 (Cross Language Evaluation Forum).* 2013, p. 17.

[2] M. Aharon, M. Elad, and A. Bruckstein. "K-SVD: An algorithm for designing overcomplete dictionaries for sparse representation". In: *IEEE Transactions on Signal Processing* 54.11 (2006), pp. 4311–4322.

[3] M. Aly, M. Munich, and P. Perona. "Distributed kd-trees for retrieval from very large image collections". In: *Proceedings of the 2011 British Machine Vision Conference - BMVC'11.* 2011, pp. 1–11.

[4] R. Aly, D. Hiemstra, and T. Demeester. "Taily: shard selection using the tail of score distributions". In: *Proceedings of the 36th International ACM SIGIR Conference on Research and Development in Information Retrievall - SIGIR '13.* 2013, pp. 673–682.

[5] Y. Anava, A. Shtok, O. Kurland, and E. Rabinovich. "A Probabilistic Fusion Framework". In: *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management - CIKM '16.* 2016, pp. 1463–1472.

[6] A. Andoni and P. Indyk. "Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions". In: *Communications of the ACM* 51.1 (2006), pp. 117–122.

[7] D. Arthur and S. Vassilvitskii. "K-Means++: the Advantages of Careful Seeding". In: *Proceedings of the 18th Annual ACM-SIAM symposium on Discrete algorithms - SODA07.* 2007, pp. 1027–1025.

[8] N. Asadi, J. Lin, and A. P. De Vries. "Runtime Optimizations for Tree-Based Machine Learning Models". In: *IEEE Transactions on Knowledge and Data Engineering* 26.9 (2014), pp. 2281–2292.

[9] J. A. Aslam and M. Montague. "Models for metasearch". In: *Proceedings of the 24th International ACM SIGIR Conference on Research and Development in Information Retrieval - SIGIR '01.* 2001, pp. 276–284.

[10] A. Babenko and V. Lempitsky. "The inverted multi-index". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 37.6 (2015), pp. 1247–1260.

[11] A. Babenko and V. Lempitsky. “Efficient Indexing of Billion-Scale Datasets of Deep Descriptors”. In: *Proceedings of the 2016 IEEE Conference on Computer Vision and Pattern Recognition - CVPR’16.* 2016, pp. 2055–2063.

[12] R. Baeza-Yates, V. Murdock, and C. Hauff. “Efficiency trade-offs in two-tier web search systems”. In: *Proceedings of the 32nd International ACM SIGIR Conference on Research and Development in Information Retrieval - SIGIR’09.* 2009, pp. 163–170.

[13] B. Bartell, G. Cottrell, and R. Belew. “Automatic Combination of multiple ranked Retrieval Systems”. In: *Proceedings of the 17th International ACM SIGIR Conference on Research and Development in Information Retrieval - SIGIR’94.* 1994, pp. 173–181.

[14] M. Batko, F. Falchi, C. Lucchese, D. Novak, R. Perego, F. Rabitti, J. Sedmidubsky, and P. Zezula. “Building a web-scale image similarity search system”. In: *Multimedia Tools and Applications* 47.3 (2010), pp. 599–629.

[15] M. Bawa, T. Condie, and P. Ganesan. “LSH Forest: Self-tuning Indexes for Similarity Search”. In: *Proceedings of the 14th International Conference on World Wide Web - WWW ’05.* 2005, pp. 651–660.

[16] J. L. Bentley. “Multidimensional binary search trees used for associative searching”. In: *Communications of the ACM* 18.9 (1975), pp. 509–517.

[17] J. C. Bezdek. *Pattern Recognition with Fuzzy Objective Function Algorithms.* Vol. 25. 3. 1981, pp. 442–442.

[18] A. Bhowmik and J. Ghosh. “LETOR Methods for Unsupervised Rank Aggregation”. In: *Proceedings of the 26th International Conference on World Wide Web - WWW ’17.* 2017, pp. 1331–1340.

[19] P. Borges, A. Mourão, and J. Magalhães. “High-Dimensional Indexing by Sparse Approximation”. In: *Proceedings of the 5th ACM International Conference on Multimedia Retrieval - ICMR ’15.* 2015, pp. 163–170.

[20] L. Breiman. “Random forests”. In: *Machine Learning* 45.1 (2001), pp. 5–32.

[21] C. Burges, T. Shaked, E. Renshaw, A. Lazier, M. Deeds, N. Hamilton, and G. Hullender. “Learning to rank using gradient descent”. In: *Proceedings of the 22nd international conference on Machine learning - ICML ’05.* 2005, pp. 89–96.

[22] S. Büttcher, C. L. A. Clarke, and G. V. Cormack. *Information Retrieval - Implementing and Evaluating Search Engines.* 2010.

[23] Z. Cao, T. Qin, T.-Y. Liu, M.-F. Tsai, and H. Li. “Learning to rank: from pairwise approach to listwise approach”. In: *Proceedings of the 24th international conference on Machine learning - ICML ’07.* 2007, pp. 129–136.

[24] O. Chapelle and Y Chang. “Yahoo! Learning to Rank Challenge Overview”. In: *Proceedings of the Learning to Rank Challenge.* Vol. 14. 2011, pp. 1–24.

[25] S. A. Chatzichristofis and Y. S. Boutalis. “CEDD: Color and edge directivity descriptor: A compact descriptor for image indexing and retrieval”. In: *Lecture Notes in Computer Science* 5008 LNCS (2008), pp. 312–322.

[26] A. Cherian, S. Sra, V. Morellas, and N. Papanikolopoulos. “Efficient nearest neighbors via robust sparse hashing”. In: *IEEE Transactions on Image Processing* 23.8 (2014), pp. 3646–3655.

[27] O. Chum, J. Philbin, and A. Zisserman. “Near Duplicate Image Detection: minHash and tf-idf Weighting”. In: *Proceedings of the 2008 British Machine Vision Conference - BMVC’08.* 2008, pp. 50.1–50.10.

[28] P. Ciaccia, M. Patella, and P. Zezula. “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.* 1997, pp. 426–435.

[29] C. W. Cleverdon. “ASLIB Cranfield Research Project - Report on the first stage of an investigation into the ccomparative efficiency of indexing systems”. In: *Aslib Journal of Information Management.* Vol. 12. 12. 1960, pp. 421–431.

[30] G. V. Cormack, C. L. A. Clarke, and S. Buettcher. “Reciprocal rank fusion outperforms condorcet and individual rank learning methods”. In: *Proceedings of the 32nd International ACM SIGIR Conference on Research and Development in Information Retrieval - SIGIR’09.* 2009, pp. 758–759.

[31] Z. Dai, C. Xiong, and J. Callan. “Query-Biased Partitioning for Selective Search”. In: *Proceedings of the 25th ACM International Conference on Information and Knowledge Management - CIKM ’16.* 2016, pp. 1119–1128.

[32] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni. “Locality-sensitive hashing scheme based on p-stable distributions”. In: *Proceedings of the 20th Annual Symposium on Computational geometry - SCG ’04.* 2004, pp. 253–262.

[33] J. Dean and S. Ghemawat. “MapReduce: simplified data processing on large clusters”. In: *Communications of the ACM* 51.1 (2008), pp. 107–113.

[34] T. Demeester, D. Trieschnigg, D. Nguyen, D. Hiemstra, and K. Zhou. “FedWeb Greatest Hits”. In: *Proceedings of the 24th International Conference on World Wide Web - WWW ’15.* 2015, pp. 27–28.

[35] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu. “A Density-based Algorithm for Discovering Clusters a Density-based Algorithm for Discovering Clusters in Large Spatial Databases with Noise”. In: *Proceedings of the 2nd International Conference on Knowledge Discovery and Data Mining - KDD’96.* 1996, pp. 226–231.

[36] S. E. Fienberg and D. C. Hoaglin. *Selected Papers of Frederick Mosteller.* 2007.

[37] Y. Freund, R. Iyer, R. E. Schapire, and Y. Singer. “An efficient boosting algorithm for combining preferences”. In: *Journal of Machine Learning Research* 4 (2003), pp. 933–969.

[38] B. J. H. Friedman. “Greedy function aproximation: A gradient boosting machine”. In: *The Annals of Statistics* 29.5 (2001), pp. 1189–1232.

[39] P. Gould. “The rise and rise of medical imaging”. In: *Medical Physics* 16.8 (2003), pp. 1–29.

[40] K. Grauman and R. Fergus. “Learning binary hash codes for large-scale image search”. In: *Studies in Computational Intelligence.* Vol. 411. 2013. Chap. Learning B, pp. 49–87.

[41] L. Havasi, D. Varga, and T. Szirányi. “LHI-tree: An efficient disk-based image search application”. In: *International Workshop on Computational Intelligence for Multimedia Understanding - IWCIM 2014.* 2014, pp. 1–5.

[42] J. P. Heo, Y. Lee, J. He, S. F. Chang, and S. E. Yoon. “Spherical hashing”. In: *Proceedings of the 2012 IEEE Conference on Computer Vision and Pattern Recognition - CVPR’12.* 2012, pp. 2957–2964.

[43] A. G. D. Herrera, J. Kalpathy-Cramer, D. D. Fushman, S. Antani, and H. Müller. “Overview of the ImageCLEF 2013 medical tasks”. In: *Working Notes of CLEF 2013 (Cross Language Evaluation Forum).* Vol. 1179. 2013, p. 15.

[44] G. E. Hinton. “Reducing the Dimensionality of Data with Neural Networks”. In: *Science* 313.5786 (2006), pp. 504–507.

[45] A. E. Hoerl and R. W. Kennard. *Ridge Regression.* Vol. 1. 8. 2011, pp. 1–10.

[46] D. F. Hsu and I. Taksa. “Comparing rank and score combination methods for data fusion in information retrieval”. In: *Information Retrieval* 8.3 (2005), pp. 449–480.

[47] H. Jegou, M. Douze, and C. Schmid. “Hamming Embedding and Weak Geometric Consistency for Large Scale Image Search”. In: *Proceedings of the 10th European Conference on Computer Vision - ECCV’10.* Vol. 5302. October. 2008, pp. 304–317.

[48] H. Jégou, M. Douze, and C. Schmid. “Product quantization for nearest neighbor search”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 33.1 (2011), pp. 117–128.

[49] H. Jegou, R. Tavenard, M. Douze, and L. Amsaleg. “Searching in one billion vectors: Re-rank with source coding”. In: *Proceedings of the 41st IEEE International Conference on Acoustics, Speech and Signal Processing - ICASSP’11* 54.2 (2011), pp. 861–864.

[50] R. Ji, L. Y. Duan, J. Chen, L. Xie, H. Yao, and W. Gao. “Learning to distribute vocabulary indexing for scalable visual search”. In: *IEEE Transactions on Multimedia* 15.1 (2013), pp. 153–166.

[51] X. Jin, A. Gallagher, L. Cao, J. Luo, and J. Han. “The wisdom of social multimedia: using flickr for prediction and forecast”. In: *Proceedings of the 18th ACM international conference on Multimedia - MM ’10.* 2010, pp. 1235–1244.

[52] Y. Jing, D. Liu, D. Kislyuk, A. Zhai, J. Xu, J. Donahue, and S. Tavel. “Visual Search at Pinterest”. In: *Proceedings of the 21st ACM SIGKDD International Conference on Knowledge Discovery and Data Mining - KDD’15.* 2015, pp. 1889–1898.

[53] T. Joachims. “Training linear SVMs in linear time”. In: *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining - KDD’06.* 2006, pp. 217–226.

[54] Y. Kalantidis and Y. Avrithis. “Locally optimized product quantization for approximate nearest neighbor search”. In: *Proceedings of the 2014 IEEE Conference on Computer Vision and Pattern Recognition - CVPR’14.* 2014, pp. 2329–2336.

[55] D. Karger, T. Leightonl, D. Lewinl, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. “Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web”. In: *Proceedings of the 29h Annual ACM symposium on Theory of computing . SOTC’97.* 1997, pp. 654–663.

[56] A. Kulkarni and J. Callan. “Document allocation policies for selective searching of distributed indexes”. In: *Proceedings of the 16th ACM International Conference on Information and Knowledge Management - CIKM’10.* 2010, pp. 449–458.

[57] I. Lee. “Relevance Feedback for Distributed Content Based Image Retrieval”. In: *Proceedings of the 2009 International Symposium on Computer Network and Multimedia Technolog - CNMT’09.* 2009, pp. 1–4.

[58] J. H. Lee. “Analyses of multiple evidence combination”. In: *Proceedings of the 20th International ACM SIGIR Conference on Research and Development in Information Retrieval - SIGIR’97.* Vol. 31. SI. 1997, pp. 267–276.

[59] H. Lejsek, F. H. Ásmundsson, B. P. Jónsson, and L. Amsaleg. “NV-tree: An efficient disk-based index for approximate search in very large high-dimensional collections”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 31.5 (2009), pp. 869–883.

[60] D. M. Levy. “Fixed or Fluid?: Document Stability and New Media”. In: *Proceedings of the 1994 ACM European conference on Hypermedia technology - ECHT’94.* 1994, pp. 24–31.

[61] Z. Li, H. Ning, L. Cao, T. Zhang, Y. Gong, and T. Huang. “Learning to Search Efficiently in High Dimensions”. In: *Advances in Neural Information Processing Systems 24 - NIPS’11.* 2011, pp. 1–9.

[62] S. Liang, M. de Rijke, and M. Tsagkias. “Late Data Fusion for Microblog Search”. In: *Proceedings of the 35th European conference on Advances in Information Retrieval - ECIR’13.* 2013, pp. 743–746.

[63] S. P. Lloyd. “Least Squares Quantitization in PCM”. In: *IEEE Transactions on Information Theory* 28.2 (1982), pp. 129–137.

[64] M. Lux and S. A. Chatzichristofis. “Lire: lucene image retrieval”. In: *Proceedings of the 16th ACM international conference on Multimedia - MM’08*. 2008, pp. 1085–1088.

[65] Y. Lv and C. Zhai. “When documents are very long, BM25 fails!” In: *Proceedings of the 34th international ACM SIGIR conference on Research and development in Information - SIGIR ’11*. 2011, pp. 1103–1104.

[66] C. Macdonald and I. Ounis. “Voting for candidates”. In: *Proceedings of the 15th ACM international conference on Information and knowledge management - CIKM ’06*. 2006, pp. 387–396.

[67] J. Magalhaes and S. Rueger. “High-dimensional visual vocabularies for image retrieval”. In: *Proceedings of the 30th International ACM SIGIR Conference on Research and Development in Information Retrieval - SIGIR’07*. 2007, p. 815.

[68] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval*. Vol. 1. 4. 2008. Chap. Scoring, t, p. 496.

[69] K. McDonald and A. F. Smeaton. “A Comparison of Score, Rank and Probability-based Fusion Methods for Video Shot Retrieval”. In: *Proceedings of the 4th ACM International Conference on Image and Video Retrieval - CIVR’05*. 2005, pp. 61–70.

[70] D. Metzler and W. Bruce Croft. “Linear feature-based models for information retrieval”. In: *Information Retrieval* 10.3 (2007), pp. 257–274.

[71] Microsoft Research Asia. “LETOR: A Benchmark Collection for Learning to Rank for Information Retrieval”. In: *Information Retrieval* 13.4 (2009), pp. 1–19.

[72] V. Mirrokni, M. Thorup, and M. Zadimoghaddam. “Consistent Hashing with Bounded Loads”. In: *Computing Research Repository* abs/1608.0 (2016), pp. 1–36.

[73] H. Mohamed and S. Marchand-Maillet. “Distributed media indexing based on MPI and MapReduce”. In: *Multimedia Tools and Applications* 69.2 (2014), pp. 513–537.

[74] D. Moise, D. Shestakov, G. Gudmundsson, and L. Amsaleg. “Indexing and searching 100M images with map-reduce”. In: *Proceedings of the 3rd ACM International conference on multimedia retrieval - ICMR ’13*. 2013, pp. 17–24.

[75] M. Montague and J. A. Aslam. “Relevance score normalization for metasearch”. In: *Proceedings of the 10th international conference on Information and knowledge management - CIKM’01*. 2001, pp. 427–433.

[76] M. Montague and J. a. Aslam. “Condorcet Fusion for Improved Retrieval”. In: *Proceedings of the 11th international conference on Information and knowledge management - CIKM’02*. 2002, pp. 538–548.

[77] A. Mourão, F. Martins, and J. Magalhães. “Multimodal medical information retrieval with unsupervised rank fusion”. In: *Computerized Medical Imaging and Graphics* 39 (2015), pp. 35–45.

[78] M. Muja and D. G. Lowe. “Fast Approximate Nearest Neighbors with Automatic Algorithm Configuration”. In: *Proceedings of the 4th International Conference on Computer Vision Theory and Applications - VISSAPP’09.* 2009, pp. 331–340.

[79] M. Muja and D. G. Lowe. “Scalable Nearest Neighbour Algorithms for High Dimensional Data”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 36.11 (2014), pp. 2227–2240.

[80] D. Nguyen, T. Demeester, D. Trieschnigg, and D. Hiemstra. “Federated search in the wild”. In: *Proceedings of the 21st international conference on Information and knowledge management - CIKM’12.* 2012, pp. 1874–1878.

[81] D. Novak and P. Zezula. “M-Chord: a scalable distributed similarity search structure”. In: *Proceedings of the 1st international conference on Scalable information systems - InfoScale ’06.* 2006, p. 19.

[82] A. Oliva and A. Torralba. “Modeling the shape of the scene: A holistic representation of the spatial envelope”. In: *International Journal of Computer Vision* 42.3 (2001), pp. 145–175.

[83] B. A. Olshausen and D. J. Field. “Emeregnce of simple-cell receptive field properties by learning a sparse code for natural images”. In: *Letters to Nature* 381.6583 (1996), pp. 607–609.

[84] Y. Pati, R. Rezaiifar, and P. Krishnaprasad. “Orthogonal matching pursuit: recursive function approximation with applications to wavelet decomposition”. In: *Proceedings of 27th Asilomar Conference on Signals, Systems and Computers - ACSSC’93.* 1993, pp. 40–44.

[85] T. Qin and T.-Y. Liu. “Introducing LETOR 4.0 Datasets”. In: *CoRR* abs/1306.2 (2013).

[86] M. Raginsky and S. Lazebnik. “Locality-sensitive Binary Codes from Shift-invariant Kernels”. In: *Advances in Neural Information Processing Systems 22 - NIPS’09.* 2009, pp. 1509–1519.

[87] K. Roberts, E. M. Voorhees, and W. Hersh. “Overview of the TREC 2015 Clinical Decision Support Track”. In: *Proceedings of The 24th Text REtrieval Conference - TREC’15* (2015), p. 12.

[88] M. Samwald, M. Kritz, M. Gschwandtner, V. Stefanov, and A. Hanbury. “Physicians searching the web for medical question answering: A European survey and local user studies”. In: *Studies in Health Technology and Informatics.* Vol. 192. 1-2. 2013, p. 1103.

[89] R. E. Schapire. “The Strength of Weak Learnability”. In: *Machine Learning* 5.2 (1990), pp. 197–227.

[90] J. A. Shaw and E. A. Fox. “Combination of Multiple Searches”. In: *Proceedings of the 2nd Text REtrieval Conference - TREC'94*. Vol. 500-215. 1994, pp. 243–252.

[91] M. S. Simpson, E. M. Voorhees, and W. Hersh. “Overview of the TREC 2014 Clinical Decision Support Track”. In: *Proceedings of The 23rd Text REtrieval Conference - TREC'14* (2014), p. 9.

[92] T. Skopal, J. Pokorný, and V. Snasel. “PM-tree: Pivoting Metric Tree for Similarity Search in Multimedia Databases.” In: *Computer and Automation Research Institute Hungarian Academy of Science*. 2004, pp. 99–114.

[93] S. Society and S. B. Methodological. “Regression Shrinkage and Selection via the Lasso Robert Tibshirani”. In: *Journal of the Royal Statistical Society. Series B: Statistical Methodology* 58.1 (2007), pp. 267–288.

[94] R. Tavenard, H. Jégou, and L. Amsaleg. “Balancing clusters to reduce response time variability in large scale image search”. In: *Proceedings of the 9th International Workshop on Content-Based Multimedia Indexing - CBMI'11*. 2011, pp. 19–24.

[95] D. G. Thaler and C. V. Ravishankar. “Using name-based mappings to increase hit rates”. In: *IEEE/ACM Transactions on Networking* 6.1 (1998), pp. 1–14.

[96] E. Tiakas, D. Rafailidis, A. Dimou, and P. Daras. “MSIDX: Multi-sort indexing for efficient content-based image search and retrieval”. In: *IEEE Transactions on Multimedia* 15.6 (2013), pp. 1415–1430.

[97] A. Torralba, R. Fergus, and W. T. Freeman. “80 million tiny images: A large data set for nonparametric object and scene recognition”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 30.11 (2008), pp. 1958–1970.

[98] A. Torralba, R. Fergus, and Y. Weiss. “Small codes and large image databases for recognition”. In: *Proceedings of the 2008 IEEE Conference on Computer Vision and Pattern Recognition - CVPR'08*. 2008, pp. 1–8.

[99] H. Valizadegan, R. Jin, R. Zhang, and J. Mao. “Learning to Rank by Optimizing NDCG Measure”. In: *Advances in Neural Information Processing Systems 22 - NIPS'09*. 2009, pp. 1883–1891.

[100] J. a. Vargas, R. S. Torres, and M. A. Gonçalves. “A Soft Computing Approach for Learning to Aggregate Rankings”. In: *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management - CIKM '15*. 1. 2015, pp. 83–92.

[101] C. C. Vogt. “Adaptive Combination of Evidence for Information Retrieval”. PhD thesis. University of California, San Diego, California, 1999.

[102] C. C. Vogt and G. W. Cottrell. “Fusion Via a Linear Combination of Scores”. In: *Information Retrieval* 1.3 (1999), pp. 151–173.

[103] R Weber, H. J. Schek, and S Blott. “A Quantitative Analysis and Performance Study for Similarity-Search Methods in High-Dimensional Spaces”. In: *Proceedings of the 24rd International Conference on Very Large Data Bases - VLDB’98*. Vol. New York C. 1998, pp. 194–205.

[104] Y. Weiss, A. Torralba, and R. Fergus. “Spectral Hashing”. In: *Advances in Neural Information Processing Systems 21 - NIPS’08* 9.1 (2008), pp. 1–8.

[105] Q. Wu, C. J. C. Burges, K. M. Svore, and J. Gao. “Adapting boosting for information retrieval measures”. In: *Information Retrieval* 13.3 (2010), pp. 254–270.

[106] F. Xia, T.-Y. Liu, J. Wang, W. Zhang, and H. Li. “Listwise approach to learning to rank”. In: *Proceedings of the 25th international conference on Machine learning - ICML ’08*. 2008, pp. 1192–1199.

[107] J. Xu and H. Li. “AdaRank: a boosting algorithm for information retrieval”. In: *Proceedings of the 30th International ACM SIGIR Conference on Research and Development in Information Retrieval - SIGIR ’07*. 2007, pp. 391–398.

[108] Z. Yang, S Kamata, and A Ahrary. “NIR: Content based image retrieval on cloud computing”. In: *Proceedings of the IEEE International Conference on Intelligent Computing and Intelligent Systems - ICIS’09*. Vol. 3. 2009, pp. 556–559.

[109] P. N. Yianilos. “Data structures and algorithms for nearest neighbor search in general metric spaces”. In: *Proceedings of the 4th Annual ACM-SIAM Symposium on Discrete Algorithms - SODA’93*. 1993, pp. 311–321.

[110] M. Zhang, R. Song, C. Lin, S. Ma, Z. Jiang, Y. Jin, Y. Liu, L. Zhao, and S Ma. “Expansion-based technologies in finding relevant and new information: Thu trec 2002: Novelty track experiments”. In: *NIST Special Publication*. 251. 2003, pp. 586–590.

[111] H. Zou and T. Hastie. “Regularization and variable selection via the Elastic Net”. In: *Journal of the Royal Statistical Society, Series B* 67 (2005), pp. 301–320.

Appendix A

Retrieval performance for individual features and methods

Table A.1: Retrieval performance of the rank lists produced using a textual index (BM25L retrieval function with MeSH query expansion and Pseudo-Relevance feedback query expansion) and image search based on a set of HSV histogram

	Image-based				Case-based			
Textual index	MAP	NDCG@20	P@10	P@30	MAP	NDCG@20	P@10	P@30
HSV histogram	0.0072	0.0297	0.0343	0.0267	0.0281	0.0637	0.0429	0.0238
BM25L_MSH_PRF (text)	0.2305	0.3359	0.2971	0.2181	0.2233	0.3216	0.2600	0.1800

Table A.2: Retrieval performance of the full set of 64 rank lists produced using various ranking functions (BM25L, BM25+, Language Models (LM) and TF_IDF, (D)) on multiple combinations of fields (**f**ull text, **a**bstract and **t**itle) query expansion (MeSH, SNOmed, Shingles and Pseudo-relevance feedback, PRF) for the TREC CDS dataset.

	TREC CDS 14				TREC CDS 15			
	MAP	NDCG@20	P@10	P@30	MAP	NDCG@20	P@10	P@30
BM25+_f_MSH_NoPRF	0.0922	0.2240	0.2700	0.2222	0.0828	0.2362	0.3333	0.2700
BM25+_f_MSH_PRF	0.1378	0.2771	0.3767	0.2778	0.1299	0.2573	0.3667	0.3000
BM25+_f_NoEXP_NoPRF	0.0953	0.2350	0.2900	0.2267	0.0878	0.2461	0.3367	0.2800
BM25+_f_NoEXP_PRF	0.1354	0.2881	0.3833	0.2900	0.1368	0.2724	0.4033	0.3156
BM25+_f_SHI_NoPRF	0.0813	0.2273	0.2833	0.1989	0.0837	0.2376	0.3200	0.2667
BM25+_f_SHI_PRF	0.1302	0.2768	0.3767	0.2722	0.1195	0.2527	0.3400	0.2867
BM25+_f_SNO_NoPRF	0.0932	0.2265	0.2833	0.2178	0.0853	0.2270	0.3467	0.2711
BM25+_f_SNO_PRF	0.1282	0.2558	0.3300	0.2633	0.1261	0.2329	0.3400	0.2911
BM25+_fat_MSH_NoPRF	0.0812	0.1887	0.2300	0.1856	0.0672	0.2009	0.2700	0.2311
BM25+_fat_MSH_PRF	0.1428	0.2656	0.3433	0.2756	0.1272	0.2626	0.3567	0.3056
BM25+_fat_NoEXP_NoPRF	0.0813	0.1908	0.2367	0.1956	0.0701	0.2139	0.2900	0.2356
BM25+_fat_NoEXP_PRF	0.1420	0.2805	0.3400	0.2944	0.1347	0.2741	0.3900	0.3278
BM25+_fat_SHI_NoPRF	0.0789	0.1913	0.2300	0.1889	0.0732	0.2197	0.2967	0.2467
BM25+_fat_SHI_PRF	0.1386	0.2671	0.3467	0.2789	0.1206	0.2420	0.3433	0.2956
BM25+_fat_SNO_NoPRF	0.0802	0.1873	0.2333	0.1922	0.0691	0.2090	0.3033	0.2433
BM25+_fat_SNO_PRF	0.1335	0.2556	0.3167	0.2522	0.1285	0.2384	0.3367	0.2922
BM25L_f_MSH_NoPRF	0.0932	0.2200	0.2533	0.2200	0.0836	0.2420	0.3300	0.2700
BM25L_f_MSH_PRF	0.1465	0.2857	0.3667	0.2989	0.1305	0.2629	0.3567	0.3000
BM25L_f_NoEXP_NoPRF	0.0958	0.2318	0.2900	0.2300	0.0887	0.2533	0.3433	0.2856
BM25L_f_NoEXP_PRF	0.1339	0.2754	0.3533	0.2967	0.1416	0.2746	0.3700	0.3144
BM25L_f_SHI_NoPRF	0.0830	0.2333	0.2833	0.2011	0.0848	0.2452	0.3100	0.2744
BM25L_f_SHI_PRF	0.1283	0.2703	0.3767	0.2667	0.1255	0.2478	0.3433	0.2844
BM25L_f_SNO_NoPRF	0.0946	0.2262	0.2767	0.2167	0.0870	0.2265	0.3367	0.2767
BM25L_f_SNO_PRF	0.1302	0.2588	0.3233	0.2656	0.1288	0.2384	0.3367	0.2944
BM25L_fat_MSH_NoPRF	0.0819	0.1845	0.2400	0.1811	0.0671	0.1943	0.2800	0.2256
BM25L_fat_MSH_PRF	0.1508	0.2725	0.3333	0.2822	0.1271	0.2656	0.3667	0.3022
BM25L_fat_NoEXP_NoPRF	0.0813	0.1923	0.2433	0.1956	0.0702	0.2117	0.2967	0.2289
BM25L_fat_NoEXP_PRF	0.1412	0.2709	0.3433	0.2844	0.1364	0.2768	0.3833	0.3178
BM25L_fat_SHI_NoPRF	0.0789	0.1863	0.2267	0.1967	0.0726	0.2157	0.3000	0.2289
BM25L_fat_SHI_PRF	0.1336	0.2593	0.3267	0.2744	0.1255	0.2500	0.3500	0.2933
BM25L_fat_SNO_NoPRF	0.0807	0.1894	0.2300	0.1878	0.0689	0.2078	0.3000	0.2389
BM25L_fat_SNO_PRF	0.1341	0.2493	0.3400	0.2622	0.1297	0.2445	0.3500	0.2956
D_f_MSH_NoPRF	0.1015	0.2368	0.2867	0.2189	0.0779	0.2012	0.2833	0.2433
D_f_MSH_PRF	0.1445	0.2539	0.3233	0.2656	0.1113	0.1971	0.2700	0.2356
D_f_NoEXP_NoPRF	0.0999	0.2270	0.2867	0.2289	0.0833	0.2189	0.3100	0.2544
D_f_NoEXP_PRF	0.1426	0.2581	0.3300	0.2700	0.1308	0.2280	0.3333	0.2722
D_f_SHI_NoPRF	0.0888	0.2219	0.2433	0.2122	0.0831	0.2171	0.2967	0.2556
D_f_SHI_PRF	0.1304	0.2382	0.3233	0.2456	0.1219	0.2298	0.3533	0.2711
D_f_SNO_NoPRF	0.0984	0.2194	0.2800	0.2200	0.0765	0.1929	0.2700	0.2378
D_f_SNO_PRF	0.1332	0.2375	0.2733	0.2500	0.1167	0.1846	0.2533	0.2567
D_fat_MSH_NoPRF	0.0393	0.1046	0.1600	0.1211	0.0406	0.1408	0.2067	0.1589
D_fat_MSH_PRF	0.1171	0.2343	0.3200	0.2322	0.0891	0.1924	0.2900	0.2278
D_fat_NoEXP_NoPRF	0.0371	0.1040	0.1567	0.1167	0.0386	0.1508	0.2167	0.1633
D_fat_NoEXP_PRF	0.1048	0.2242	0.3100	0.2233	0.0889	0.2157	0.3400	0.2389
D_fat_SHI_NoPRF	0.0378	0.1079	0.1533	0.1167	0.0393	0.1518	0.2200	0.1633
D_fat_SHI_PRF	0.0979	0.2122	0.2867	0.2078	0.0883	0.2250	0.3300	0.2500
D_fat_SNO_NoPRF	0.0388	0.1126	0.1500	0.1189	0.0363	0.1231	0.1733	0.1389
D_fat_SNO_PRF	0.1315	0.2420	0.3167	0.2533	0.1109	0.1884	0.2800	0.2389
LM_f_MSH_NoPRF	0.0916	0.2127	0.2633	0.2167	0.0808	0.2299	0.3167	0.2589
LM_f_MSH_PRF	0.1296	0.2610	0.3267	0.2611	0.1258	0.2609	0.3400	0.3000
LM_f_NoEXP_NoPRF	0.0926	0.2185	0.2533	0.2156	0.0858	0.2526	0.3400	0.2789
LM_f_NoEXP_PRF	0.1132	0.2361	0.2900	0.2356	0.1388	0.2861	0.3600	0.3189
LM_f_SHI_NoPRF	0.0799	0.2011	0.2433	0.1911	0.0816	0.2423	0.3133	0.2644
LM_f_SHI_PRF	0.1007	0.2149	0.2700	0.2089	0.1259	0.2722	0.3600	0.3056
LM_f_SNO_NoPRF	0.0926	0.2113	0.2633	0.2044	0.0839	0.2227	0.3133	0.2678
LM_f_SNO_PRF	0.1253	0.2427	0.3300	0.2467	0.1280	0.2338	0.3133	0.2944
LM_fat_MSH_NoPRF	0.0927	0.1941	0.2500	0.2000	0.0758	0.2075	0.2700	0.2478
LM_fat_MSH_PRF	0.1359	0.2635	0.3233	0.2667	0.1283	0.2623	0.3333	0.3044
LM_fat_NoEXP_NoPRF	0.0969	0.2113	0.2667	0.2111	0.0805	0.2268	0.3033	0.2633
LM_fat_NoEXP_PRF	0.1224	0.2423	0.2933	0.2589	0.1422	0.2825	0.3833	0.3356
LM_fat_SHI_NoPRF	0.0914	0.2064	0.2767	0.2022	0.0812	0.2277	0.3300	0.2644
LM_fat_SHI_PRF	0.1091	0.2101	0.2733	0.2178	0.1284	0.2697	0.3700	0.3156
LM_fat_SNO_NoPRF	0.0959	0.2088	0.2367	0.2000	0.0788	0.2189	0.3133	0.2544
LM_fat_SNO_PRF	0.1309	0.2469	0.3067	0.2511	0.1297	0.2397	0.3400	0.3000

Table A.3: Retrieval performance of the full set of 148 rank list produced using multiple search engines for the TREC FW dataset.

	TREC FW 13		TREC FW 14	
	NDCG@20	P@10	NDCG@20	P@10
arXiv.org	0.0232	0.0917	0.0050	0.0273
CCSB	0.0008	0.0061	0.0097	0.0429
CERN_Documents	0.0000	0.0000	0.0046	0.0286
CiteSeerX	0.0097	0.0622	0.0008	0.0042
CiteULike	0.0037	0.0333	0.0146	0.0651
eScholarship	0.0224	0.0824	0.0275	0.1269
KFUPM_ePrints	0.0000	0.0000	0.0035	0.0083
MPRA	0.0009	0.0040	0.0021	0.0135
MS_Academic	0.0190	0.0960	0.0321	0.1391
Nature	0.0116	0.0500	0.0014	0.0125
Organic_Eprints	0.0000	0.0000	0.0000	0.0000
SpringerLink	0.0155	0.0773	0.0157	0.0918
U._Twente	0.0003	0.0036	0.0025	0.0140
UAB_Digital	0.0088	0.0200	0.0084	0.0500
UQ_eSpace	0.0043	0.0233	0.0084	0.0356
PubMed	0.0218	0.0905	0.0125	0.0676
LastFM	0.0046	0.0125	0.0031	0.0267
LYRICSnMUSIC	0.0018	0.0105	0.0004	0.0034
Comedy_Central	0.0065	0.0556	0.0000	0.0000
Dailymotion	0.0692	0.2468	0.0799	0.2551
YouTube	0.1648	0.4580	0.2062	0.5420
Google_Blogs	0.1567	0.4440	0.2896	0.5420
LinkedIn_Blog	0.0000	0.0000	0.0000	0.0000
Tumblr	0.0459	0.2500	0.0054	0.0154
WordPress	0.0882	0.2170	0.1076	0.3540
Goodreads	0.0272	0.1357	0.0760	0.2187
Google_Books	0.0242	0.1240	0.0462	0.1860
NCSU_Library_	0.0066	0.0333	0.0266	0.1140
IMDb	0.0102	0.0737	0.0127	0.0143
Wikibooks	0.0075	0.0187	0.0196	0.0854
Wikipedia	0.0725	0.1898	0.0951	0.2354
Wikispecies	0.0000	0.0000	0.0000	0.0000
Wiktionary	0.0328	0.0533	0.0089	0.0500
E!_Online	0.0458	0.0727	0.0029	0.0176
Entertainment_Weekly	0.0079	0.0385	0.0055	0.0314
TMZ	0.0038	0.0152	0.0000	0.0000
Addicting_games	0.0000	0.0000	0.0000	0.0000
Amorgames	0.0000	0.0000	0.0000	0.0000
Crazy_monkey_games	0.0000	0.0000	0.0000	0.0000
GameNode	0.0000	0.0000	0.0003	0.0029
Games.com	0.0000	0.0000	0.0111	0.0250
Miniclip	0.0000	0.0000	0.0000	0.0000
About.com	0.0942	0.2909	0.1539	0.4021
Ask	0.2058	0.4780	0.2692	0.4940
CMU_ClueWeb	0.0842	0.2261	0.1085	0.2796
Gigablast	0.1332	0.3040	0.1558	0.1000
Baidu	0.1345	0.2760	0.1130	0.2939
Centers_for_Disease_Control_and_Prevention	0.0084	0.0273	0.0082	0.0273
Family_Practice_notebook	0.0230	0.0333	0.0360	0.1200
Health_Finder	0.0098	0.0400	0.0037	0.0250
HealthCentral	0.0077	0.0208	0.0231	0.0714
HealthLine	0.0109	0.0458	0.0294	0.0667
Healthlinks.net	0.0238	0.1000	0.0150	0.0500
Mayo_Clinic	0.0193	0.0500	0.0326	0.0735
MedicineNet	0.0174	0.0519	0.0234	0.0938
MedlinePlus	0.0000	0.0000	0.0087	0.0618
University_of_Iowa_hospitals_and_clinics	0.0009	0.0091	0.0140	0.0727
WebMD	0.0231	0.0720	0.0296	0.0974
Glassdoor	0.0000	0.0000	0.0000	0.0000
Jobsite	0.0000	0.0000	0.0000	0.0000
LinkedIn_Jobs	0.0005	0.0063	0.0000	0.0000
Simply_Hired	0.0000	0.0000	0.0029	0.0067
USAJobs	0.0000	0.0000	0.0000	0.0000
Comedy_Central_Jokes.com	0.0127	0.0500	0.0000	0.0000
Kickass_jokes	0.0000	0.0000	0.0000	0.0000
Cartoon_Network	0.0000	0.0000	0.0000	0.0000
Disney_Family	0.0027	0.0089	0.0142	0.0354
Factmonster	0.0072	0.0188	0.0324	0.0848
Kidrex	0.3107	0.5617	0.2402	0.4275
KidsClicks!	0.0000	0.0000	0.0148	0.0625
Nick_jr	0.0000	0.0000	0.0000	0.0000
OER_Commons	0.0000	0.0000	0.0291	0.1182
Quintura_Kids	0.0172	0.1000	0.0536	0.0667
Foursquare	0.0000	0.0000	0.0008	0.0022

	NDCG@20	P@10	NDCG@20	P@10
BBC	0.0488	0.1103	0.0271	0.0585
Chronicling_America	0.0000	0.0000	0.0008	0.0069
CNN	0.0454	0.1351	0.0424	0.1316
Forbes	0.0143	0.0829	0.0063	0.0452
JSOnline	0.0045	0.0368	0.0562	0.1077
Slate	0.0089	0.0522	0.0196	0.0571
The_Street	0.0060	0.0429	0.0022	0.0143
Washington_post	0.0058	0.0311	0.0090	0.0489
HNSearch	0.0136	0.0385	0.0344	0.0455
Slashdot	0.0014	0.0083	0.0080	0.0467
The_Register	0.0062	0.0167	0.0000	0.0000
DeviantArt	0.0161	0.0900	0.0285	0.1070
Flickr	0.0260	0.1476	0.0088	0.0619
Fotolia	0.0152	0.0750	0.0060	0.0348
Getty_Images	0.0074	0.0357	0.0054	0.0412
IconFinder	0.0000	0.0000	0.0003	0.0022
NYPL_Gallery	0.0165	0.0333	0.0142	0.1125
OpenClipArt	0.0000	0.0000	0.0000	0.0000
Photobucket	0.0490	0.1955	0.0350	0.1091
Picasa	0.0452	0.1816	0.0106	0.0900
Picsearch	0.0705	0.2500	0.0738	0.2100
Wikimedia	0.0542	0.2476	0.0355	0.1486
Funny_or_Die	0.0108	0.0692	0.0078	0.0538
4Shared	0.0336	0.1133	0.0501	0.2214
AllExperts	0.0166	0.0795	0.0191	0.0976
Answers.com	0.0218	0.0653	0.0718	0.1840
Chacha	0.0227	0.1412	0.0300	0.1625
StackOverflow	0.0031	0.0042	0.0004	0.0031
Yahoo_Answers	0.3703	0.6816	0.1106	0.3813
MetaOptimize	0.0000	0.0000	0.0000	0.0000
HowStuffWorks	0.0109	0.0435	0.0281	0.0580
AllRecipes	0.1510	0.4500	0.0912	0.1900
Cooking.com	0.0464	0.0727	0.0166	0.0474
Food_Network	0.0115	0.0167	0.0343	0.0650
Food.com	0.0112	0.0200	0.0700	0.1357
Meals.com	0.0187	0.0333	0.0676	0.1889
Amazon	0.0420	0.1480	0.0379	0.1552
ASOS	0.0000	0.0000	0.0066	0.0182
Craigslist	0.0000	0.0000	0.0189	0.0667
eBay	0.0382	0.1204	0.0508	0.2318
Overstock	0.0275	0.1000	0.0208	0.0889
Powell's	0.0187	0.1143	0.0609	0.1846
Pronto	0.0142	0.0460	0.0073	0.0347
Target	0.0136	0.0828	0.0386	0.1621
Yahoo!_Shopping	0.0152	0.0580	0.0272	0.0980
Myspace	0.0084	0.0286	0.0000	0.0000
Reddit	0.0679	0.2049	0.0415	0.1617
Tweepz	0.0144	0.0500	0.0063	0.0421
Cnet	0.0156	0.0878	0.0097	0.0333
GitHub	0.0000	0.0000	0.0222	0.0545
SourceForge	0.0000	0.0000	0.0000	0.0000
bleacher_report	0.0555	0.1889	0.0039	0.0444
ESPN	0.0000	0.0000	0.0000	0.0000
Fox_Sports	0.0408	0.1571	0.0011	0.0133
NHL	0.0000	0.0000	0.0000	0.0000
SB_nation	0.0006	0.0056	0.0000	0.0000
Sporting_news	0.0597	0.1538	0.0014	0.0148
WWE	0.0000	0.0000	0.0000	0.0000
Ars_Technica	0.0060	0.0318	0.0123	0.0396
CNET	0.0027	0.0143	0.0060	0.0245
Technet	0.0046	0.0139	0.0005	0.0063
Technorati	0.0711	0.2857	0.0251	0.0650
TechRepublic	0.0192	0.0833	0.0043	0.0294
TripAdvisor	0.0204	0.0361	0.0194	0.0448
Wiki_Travel	0.0531	0.1000	0.0000	0.0000
5min.com	0.0450	0.1826	0.0761	0.1758
AOL_Video	0.3427	0.6740	0.2722	0.4649
Google_Videos	0.2029	0.5040	0.1942	0.5060
MeFeedia	0.0645	0.1667	0.0434	0.1433
Metacafe	0.0441	0.1020	0.0308	0.0800
National_geographic	0.0063	0.0419	0.0153	0.0395
Veoh	0.0828	0.2460	0.0674	0.1900
Vimeo	0.0276	0.1211	0.0380	0.1174
Yahoo_Screen	0.3773	0.6837	0.1499	0.4480
BigWeb	0.2755	0.6128	0.1058	0.1780

APPENDIX B

Detailed analysis of the impact of the number of rank lists and rank positions on relevance

B.1 Modeling relevance as a function of the number of rank lists per document

Figure B.1 shows a set of functions for boosting the score according to the document nrl. Our hypothesis is that a linear nrl, as applied by CombMNZ, Section 2.5.1, over-emphasis documents that have a high nrl. This effect is similar to what was observed in BM25L ranking function [65]: where the logarithm was introduced to counteract increased score on long documents given by regular BM25. The function that were tested to replace a linear nrl boost represent logarithmic, exponential and sigmoid growth boosts. The chosen functions are normalized so that documents on one rank list have a boost of one: the percentage of relevant documents is divided by the percentage of relevant documents when $nrl = 1$.

To show how well the nrl boost functions model real-world data, the dashed lines represent the relative number of relevant documents per nrl values (a normalized version of Figures 3.1 and 3.2 data). These functions are normalized so that relevant documents on one rank list have a boost of one: the percentage of relevant document in a set nrl, $perRel_nrl$ is divided by the percentage of relevant documents when $nrl = 1$, $perRel_1$, yielding $b_nrl = perRel_nrl/perRel_1$. For example, if 25% of document with a $nrl = 1$ are relevant ($perRel_1 = 0.25$) and 75% of document with a $nrl = 2$ are relevant ($perRel_3 = 0.75$), the $b_{nrl=3} = 0.75/0.25 = 3$.

Regularization factors that model the growth in score with frequency are represented as f,g, and h. For the logarithmic function, $b = log(l + f)$, f represents the regularization to add to the number of rank lists, to avoid a logarithm value of 0 for $l = 1$. The sigmoid

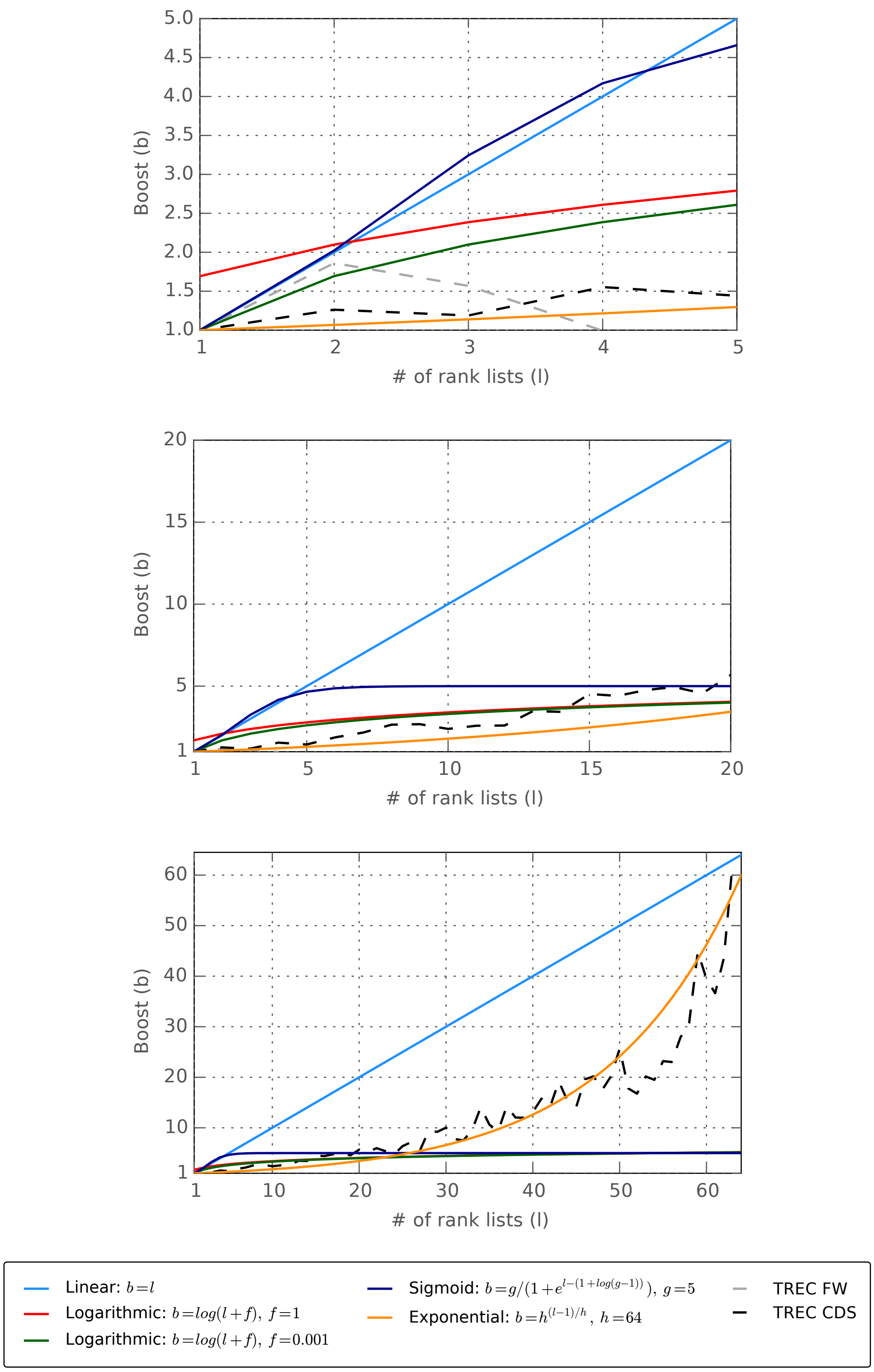


Figure B.1: Multiple document frequency weighting functions. The X-axis represents the number of rank lists a document appears), *nrl*. The Y-axis represents the boost given by the function. Each chart varies the scale of the X-axis (1-5, 1-20 and 1-60).

function, $b = g/(1+e^{l-(1+log(g-1))})$, regularization factor g was set to five to match the logarithmic function boost. For the exponential function, $b = h^{(l-1)/h}$, h represents the desired maximum boost value, set to the total number of rank lists.

For low document frequency values (1 to 10), FedWeb's relevance boost seems to degrade when documents are present on more than three lists. CDS relevance boost grows between a logarithmic and a slow exponential rate. For higher document frequencies, CDS relevance growth approaches an exponential function that weighs documents between 1 and the number of rank lists, $\sqrt[h]{h}^{l-1}$ where h is the total number of rank lists.

The linear nrl boost is much stronger than the relevance growth on CDS and FedWeb data. For the logarithmic functions, multiple regularization factors f were tested. At $f = 1$ the weight difference between low nrl documents and high nrl was too small. At $f = 0.01$, the weight difference is closer to CDS and FedWeb data, giving lower boosts to documents to low nrl documents.

B.2 Modeling score decay with rank list position

Document frequency is a key factor for ranking, but it ignores the position of documents on the rank list. Figure B.2 show how multiple functions model the decay in relevance with the decrease in the position on the rank list at multiple scales ($1-10$, $1-100$ and $1-1000$ rank positions). The plotted functions include Borda's linear decay, RRF's inverse decay for multiple k values, exponential decay, and logarithmic decay. The dashed lines represent the (normalized) number of relevant documents at each rank position, for all rank lists in the TREC CDS and FedWeb datasets. In other words, it measures how relevance decays with the position on the rank. For example, if 50 ranks lists have a relevant document at the first rank position and 40 documents at the second rank position, normalized values will be $50/50 = 1$ for $r = 1$ and $40/50 = 0.8$ for $r = 2$. As with the previous section, score values are normalized to enable us to measure the relative differences between functions.

The RRF scores decrease slowly with the rank position. For a $k = 60$, documents ranked in position 50 will have less than half of the score of the documents ranked first in a list. The RRF family of functions are the ones that better fit relevance: relevance decreases with rank position according to RRF's decay. Both FW and CDS relevance decay seems to fit RRF decay with k values of between 10 and 60. It is also interesting to see that the ideal k value seems to increase with the rank position. The RR function presents a faster decay; a document ranked at position 10 has a score that is ten times smaller than a document ranked at the first position; a document ranked 50 gets an almost negligible score. Exponential decay amplifies this difference: a document on position ten will have 1/100 of the score of a document with the rank position of 1.

BordaFuse linear decay (scores are decreased by one for each rank position 1000, 999, 998, ...) is much slower than inverse ranking approaches. Logarithmic decay is even slower: for example, a document on rank position 700 will still have over 3/5 of the score of

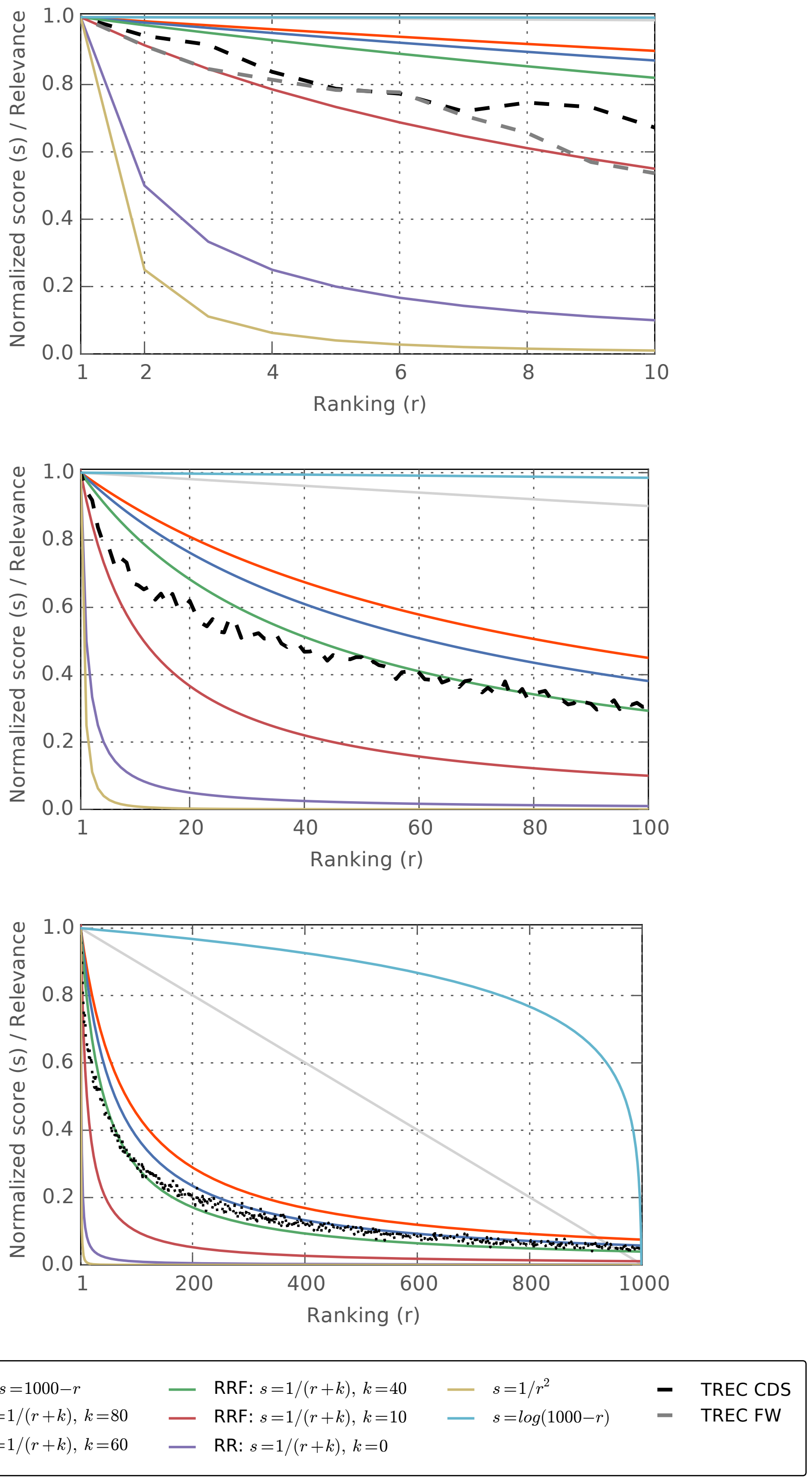


Figure B.2: Rank versus score using multiple functions. The X-axis represents the document position on the rank list. The Y-axis represents the score given by the rank fusion function. Each chart varies the scale of the X-axis ($1-10$, $1-100$, and $1-1000$).

a document ranked 1. Both approaches return scores which are far from the relevance information observed for the CDS and FedWeb datasets.

B.3 Frequency-boosted Reciprocal Rank Fusion

The previous section showed that there is a correlation between the number of rank lists documents appear, *nrl* and relevance, validating the Chorus Effect. But RRF does not explicitly boost documents with *nrl*. This section proposes and benchmarks *Frequency-boosted Reciprocal Rank Fusion*, Thus, this section proposes two "frequency boosted" versions of RRF, based on logarithmic and exponential *nrl* boosts to RRF document scores:

$$\text{LogN\_RRF}(i) = log(nrl(i) + \sigma) \times \sum_{k=1}^{nrl(i)} \frac{1}{R_k(i) + k}, \tag{B.1}$$

$$\text{exp\_RRF}(i) = \sqrt[h]{h}^{nrl(i)-1} \times \sum_{k=1}^{nrl(i)} \frac{1}{R_k(i) + k}, \tag{B.2}$$

where *N(i)* is the number of times a document appears on a rank list (document frequency), R_k*(i)* is the rank of document i on the kth rank, and h is the exponential regularization factor.

Tables B.1 and B.2 benchmark RRF and the proposed variants to : (i) test different types of *nrl* boosts to approach the relevance curve for the TREC FW and CDS data in Figure B.1 and (ii) vary the value of k to approximate the observed relevance decay on Figure B.2. These results show that the impact of k is small; the main improvement in performance happens the value of k goes from zero to 10. For large values of $k > 60$, performance either stabilizes,Table B.2 or decays slightly, Table B.1. Regarding applying explicit *nrl* boosts as provided by LogNRRF and ExpRRF, the impact in performance is very small: LogNRRF had slightly better results for the CDS 14 and FW 14 data, but the improvement is not consistent across datasets and values of k. These results shows that adding a explicit boost to documents that appear on more rank lists to RRF has a limited effect on performance. RRF is able to adequately model relevance decay with rank position by the sum of the inverse rank positions, without the need for an explicit *nrl* boost. The small effect of the k value on retrieval performance matches existing rank fusion literature [30].

Table B.1: Fusion results for RRF and boosted variants (*log* and *exp*) for multiple values of k on the TREC CDS dataset. k values between 40 and 100 yielded similar results as $k = 40$

	RRF		LogNRRF		ExpNRRF	
k	NDCG@20	P@10	NDCG@20	P@10	NDCG@20	P@10
TREC CDS 14						
0	0.2676	0.3200	0.2679	0.3200	**0.2694**	0.3267
10	**0.2734**	**0.3333**	**0.2750**	0.3333	0.2678	**0.3300**
20	0.2713	0.3300	0.2705	0.3333	0.2678	0.3267
40	0.2718	**0.3333**	0.2707	**0.3400**	0.2657	0.3233
60	0.2699	0.3300	0.2702	0.3300	0.2644	0.3200
80	0.2695	0.3267	0.2684	0.3233	0.2661	0.3167
100	0.2675	0.3233	0.2657	0.3200	0.2679	0.3133
TREC CDS 15						
0	0.2510	0.3667	0.2525	0.3667	0.2574	**0.3633**
10	0.2610	**0.3800**	0.2642	**0.3800**	0.2634	0.3500
20	0.2640	**0.3800**	0.2640	**0.3800**	**0.2657**	0.3567
40	0.2653	0.3567	0.2673	0.3567	0.2646	0.3567
60	**0.2677**	0.3533	**0.2675**	0.3500	0.2622	0.3433
80	0.2654	0.3433	0.2639	0.3433	0.2615	0.3400
100	0.2608	0.3433	0.2596	0.3433	0.2562	0.3367

Table B.2: Fusion results for RRF and boosted variants (*log* and *exp*) for multiple values of k on the TREC FW dataset. k values between 40 and 100 yielded similar results as $k = 40$

	RRF		LogNRRF		ExpNRRF	
k	NDCG@20	P@10	NDCG@20	P@10	NDCG@20	P@10
TREC FW 13						
0	0.3171	0.5220	0.4954	0.7120	0.3392	0.5600
10	**0.4969**	0.7260	0.4949	0.7280	0.4951	0.7280
20	0.4960	**0.7300**	0.4960	**0.7300**	0.4960	**0.7300**
40	0.4961	**0.7300**	**0.4961**	**0.7300**	**0.4961**	**0.7300**
TREC FW 14						
0	0.2234	0.3620	**0.3510**	0.5300	0.2345	0.3820
10	0.3489	0.5400	0.3476	0.5440	0.3478	0.5440
20	0.3476	0.5460	0.3486	**0.5460**	0.3478	**0.5460**
40	**0.3486**	**0.5460**	0.3486	**0.5460**	**0.3486**	**0.5460**

Appendix

C

Proof for the estimation of the redundancy factor $\hat{r}$

Consider m the number of nodes and s the number of non-null coefficients for a documents' sparse hash. Our goal is to determine the expected percentage of nodes that will have at least one of the coefficients. In other words, on how many nodes will each document be indexed. This is a variation of the birthday problem or the hash collision problem, detailed in Fienberg and Hoaglin [36], Equation 4.

Assuming that the distribution of documents across nodes is balanced, and that the distribution of coefficients across nodes is random, the probability of each coefficient being assigned to a node is $1/m$. Choose a coefficient out of the s coefficients. The probability of these coefficient falling on a different node than the other $s-1$ coefficients is $\sigma^{s-1} = (\frac{m-1}{m})^{s-1}$, and the probability of a collision is $1-\sigma^{s-1}$. The expected number of occupied nodes, $\widehat{rm}_s$, for the first case is $1+\widehat{rm}_{s-1}$. For the second case, is $\widehat{rm}_{s-1}$.

$$\begin{aligned}
\widehat{rm}_s &= \sigma^{s-1}(1+\widehat{rm}_{s-1}) + (1-\sigma^{s-1})\widehat{rm}_{s-1} \\
&= \sigma^{s-1} + \widehat{rm}_{s-1}
\end{aligned}$$

as $\widehat{rm}_1 = 1$, we can rewrite the expression as:

$$\begin{aligned}
\widehat{rm}_s &= \sigma^{s-1} + \sigma^{s-2} + \ldots + \sigma^1 + 1 \\
&= (\sigma^s - 1)/(\sigma - 1) \\
&= m(1-\sigma^s) \\
&= m\left(1 - \left(\frac{m-1}{m}\right)^s\right)
\end{aligned} \tag{C.1}$$

The percentage of occupied nodes $\hat{r}$ becomes:

$$\hat{r} = \frac{\widehat{rm}}{m} = 1 - \left(\frac{m-1}{m}\right)^s \tag{C.2}$$