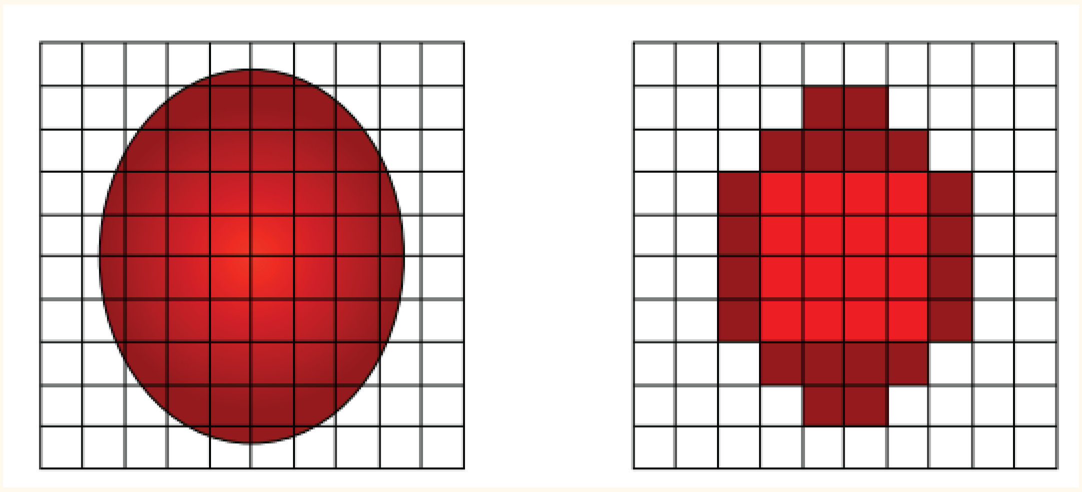
Efficient hybrid search in Vector Database
Undergraduate Research and Innovation Scheme (URIS), PolyU

Study Report
Visual Features Indexing

—

By Anthony Zhe Jun HE



Human vision is a remarkably complex process that starts with the eye and is fully accomplished by specialized regions in the brain.

1. Overview of Visual Features

Digital imaging in some way mimics the human eye. The sensors usually has three types of photo sensitive elements (red, green and blue) and converts the incoming light into electrical signals. The resulting analog signal (模拟信号) is then converted to a digital representation, which consists of two main steps: *sampling* and *quantization*.

Sampling involves selecting a finite number of points within each dimension of image, whereas **quantization** (量子化) implies assigning an amplitude value (振幅值) to each of those points. The result of the digitization process is a *pixel array* (or *raster*), which us a rectangular matrix if picture elements whose values correspond to their intensity (for monochrome images 黑白照片) or color components (for color images).

Contemporary cameras produce pixel arrays with more than 1000 pixels per dimensions, i.e., greater than 1 megapixel (MP)in terms of total number of pixels, with 256 colors per color channel (R, G, or B), resulting in a total of approximately 16 million colors (i.e., 256^3).

Some interesting topics about RGB, PNG and JPEG are mentions staring from Page 28, Visual Information Retrieval using Java and LIRE.

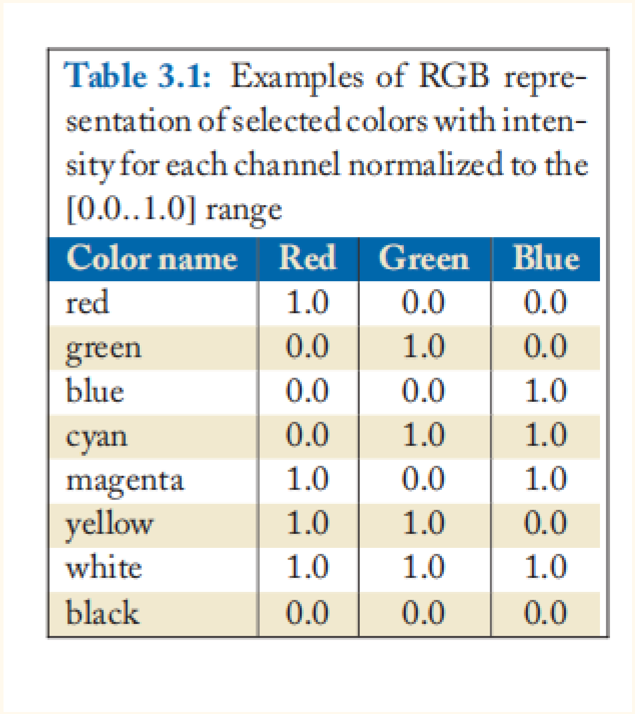


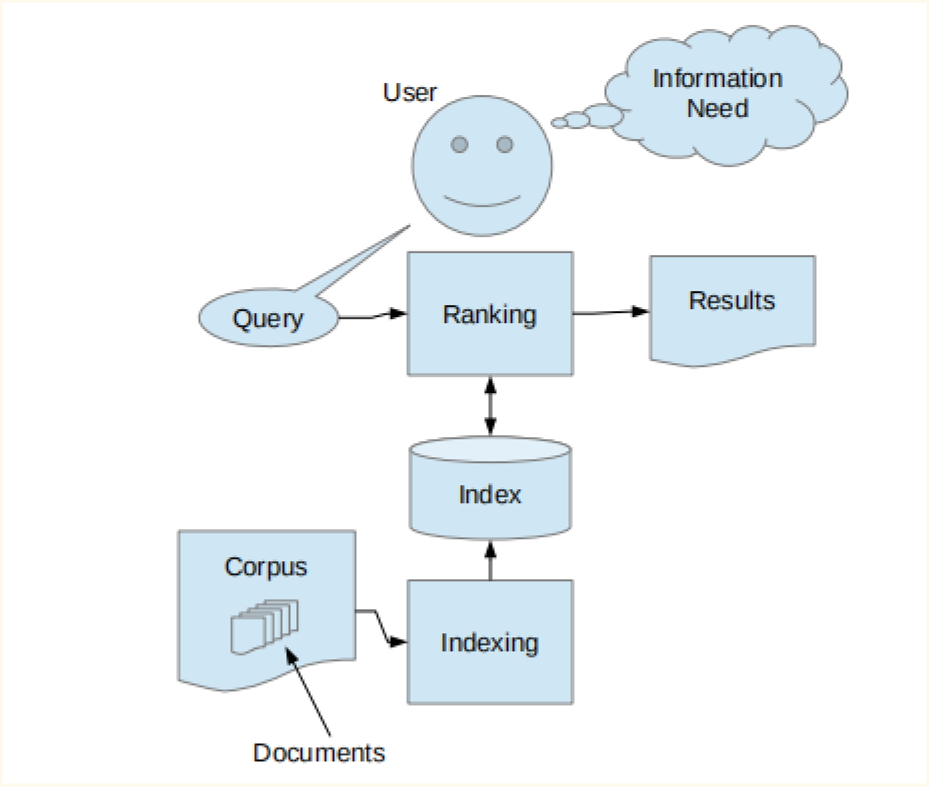
Table 3.1: Examples of RGB representation of selected colors with intensity for each channel normalized to the [0.0..1.0] range

Color name	Red	Green	Blue
red	1.0	0.0	0.0
green	0.0	1.0	0.0
blue	0.0	0.0	1.0
cyan	0.0	1.0	1.0
magenta	1.0	0.0	1.0
yellow	1.0	1.0	0.0
white	1.0	1.0	1.0
black	0.0	0.0	0.0

2. Indexing Overview

In Figure 3, it is a general overview of the architecture of information retrieval.

In text retrieval, the use of an inverted index (or inverted list) is common practice. Essentially, an inverted index is an index data structure that stores a mapping from words to their locations in a set of documents. An inverted index is organized by a term dictionary, containing all the terms of all indexed documents. For each term there is a list indicating in which documents the respective term appears; these lists are called posting lists. At search time, only the terms used in the query need to be looked up in the term dictionary. The lists of documents referenced by



the query terms are then used to compute the rank of each document occurring in at least one of the list elements. The relevance function is typically based on tf*idf and cosine coefficient, which conveniently work well together with the index structure of an inverted list.

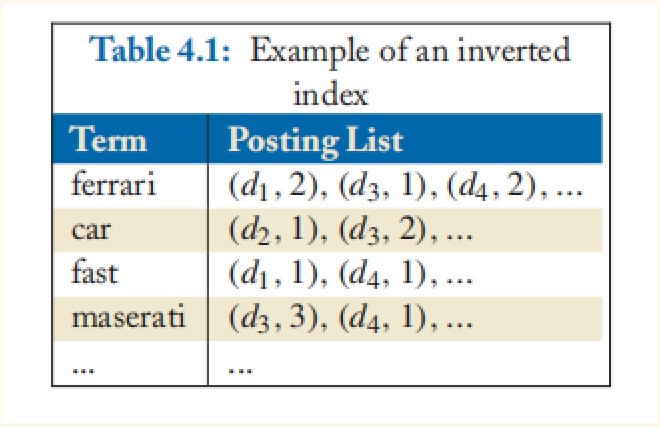
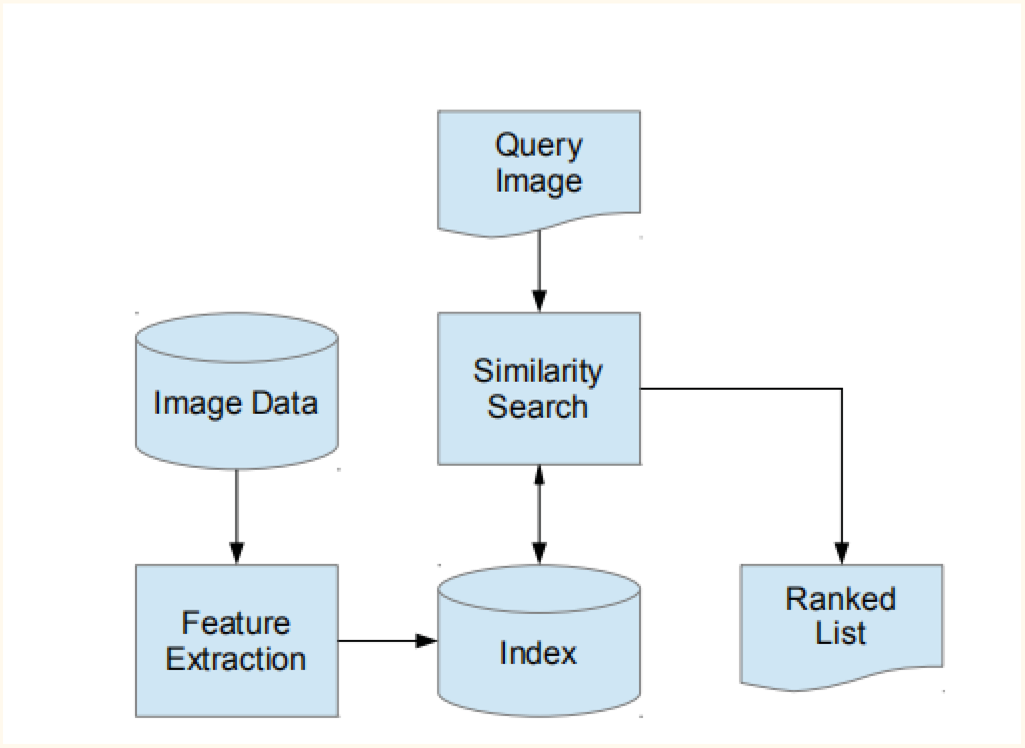


Table 4.1: Example of an inverted index

Term	Posting List
ferrari	$(d_1, 2), (d_3, 1), (d_4, 2), \ldots$
car	$(d_2, 1), (d_3, 2), \ldots$
fast	$(d_1, 1), (d_4, 1), \ldots$
maserati	$(d_3, 3), (d_4, 1), \ldots$
...	...

In Content-based Information Retrieval (CBIR), a typical searching scenario uses a data structure called *index*. Figure 4 shows a basic process of searching an image repository and the role of the index.

Indexing an image consists of **extracting low-level features from the image data** and **storing a representation of the extracted features into the index**.



The CBIR search process using a query-by-example (QBE) paradigm consists of presenting a query image to the system, indexing the example image ("online"), and performing a similarity search against the previously stored index. The results of the search process are usually presented as a ranked list of images, sorted in decreasing order of similarity to the query image.

3. The naive approach of Visual Features Indexing and its issues.

A simple approach to indexing consists of extracting and encoding the relevant image features before search time, and storing them in a database, or in memory, to have them readily available at search time. This basically leads to a list (or set if there is no intrinsic order) of data entries, which includes (i) a pointer to the image file or URI, and (2) the extracted low-level features encoded in a suitable way.

At search time this list or set is traversed linearly and similar images are collected for a result set, which must be ordered according to their relevance relative to the query. During the traversal process, each and every data entry has to be: (i) read from disk; (ii) decoded; (iii) compared to the query image's encoded features; and (iv) eventually added to the result set. For n data entities this results in a computational complexity of *O(n)*.

It is important to note that computational complexity - especially for small values of n - depends strongly on constants that are not apparent in the O(n) formulation.

1. **Properties of the actual medium in which the data entities are stored.** SSDs, HDDs, main memory and network storage all have very different characteristics influencing the search process.
2. **Decoding of data entities.** The data entries in the resulting list or set are often encoded (i.e., compressed) features. There is an inherent tradeoff involved in the process: while decoding takes time, compression, on the other hand, can allow for faster transfers between different media, e.g., from HDDs to main memory.
3. **Object access and manipulation overhead.** In high-level programming languages, such as Java and C#, each and every object has its properties stored in a header (16 bytes for a java.lang.Object instance on a 64-bit Oracle VM). This header eventually adds to the size of the data entities, especially if you use nested objects and arrays of objects

4. LIRE

LIRE is a Java library designed to enable the quick deployment of *Visual information retrieval* (VIR) solutions. It can be used to build image retrieval applications from scratch. LIRE is based on Lucene, a text search engine that provides capabilities such as inverted indexing, search, and fast random access to text indexes.

The main tasks of LIRE are: (i) to extract low-level features from images; (ii) to index the low-level features; and (iii) to perform the search in a database of indexed features.

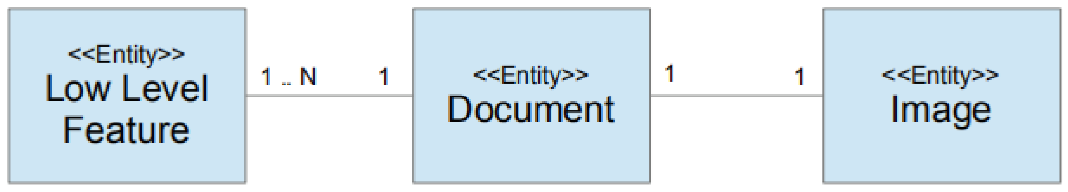


Figure 5.1: LIRE building blocks. An image corresponds to exactly one Document, which contains between 1 and N low-level features from the image.

The architecture of LIRE can be understood with the help of a simple entity-relationship diagram. An image corresponds to exactly one *Lucene Document* instance, which contains between 1 and N low-level features from the image.