# Information Retrieval

Information retrieval is **finding material (usually documents)** of **unstructured nature** (usually text) that **satisfies an information need** from within **large collections** (usually stored on computers)

Structured data => SQL databases

Semi Structured data => JSON, XML, CSV

Unstructured data => Raw text

**Structured data**: Data in SQL databases needs to conform to a particular interface or schema in order for it to be stored in a particular table. All of the records in a particular relational database, in a particular table will have the same shape. We interact with it using formal languages like SQL.

**Semi Structured data**: We can build tools that parse information that takes the form of JSON, XML or CSV. There is a lot of flexibility and a lot rooms for errors to be introduced in terms of the structure of particular records.

**Information need:** user desire to find some particular information from a system in order to satisfy a particular need. This will influence the type of search engine that we build, and the particular purpose for that search engine.

## Basic assumptions

The assumption of information retrieval is that we need to return information that is relevant to the user’s information need, which is why collections should have a fixed size.

**Collection:** Fixed set of documents

**Goal:** Retrieve documents with information that is relevant to the user’s information need and helps the user complete a task.

## The classic search model

Diagram

Description automatically generated

## How good are the retrieved docs?

When evaluating search, we can evaluate retrieval using two parameters:

**Precision:** Fraction of the retrieved docs that are relevant to user’s information need (Basically through surveys)

**Recall:** What was ignored that was relevant. Fraction of relevant documents in the collection that were retrieved

There are other measures but precision and recall are the most important ones.

# Indexing Data

If we work with smalls amounts of data, we can just use regular expression to linearly scan through that data to extract the information we are looking for (using grep for instance). Linear search is slow for large corpora, and we also want to consider tank retrieval eventually.

Indexing and preprocessing of data is needed when working with big amounts of data so that we can efficiently query it.

The first type of index we will be working with is the **Term Document Incidence Matrix.** This is a data structure that will allow us to do a specific type of querying against our data which uses something called the **Boolean retrieval** model.

## Boolean retrieval

Simple idea that we combine terms with Boolean operators. E.g.: Term1 & term2.

# Term Document incidence matrix

One way that we can support Boolean retrieval model search is by building a **term document incidence matrix. Term Document Incidence Matrix** is a table that lays out the words between different sources, such that word appearance is either 1 or 0 (depending on if the word is contained in a document)

To construct a term document matrix we need to tokenize our documents, meaning we need to separate all terms in our documents.

Text, letter

Description automatically generated

The terms are words (keep in mind composed words like new York. New York is a single concept and therefore a single term. We don’t want to separate new and york).

The collection of documents is also known as a **corpus**. Multiple corpuses are called **corpora**. In the previous image we have one corpora that contains 3 documents, and each of them contains a different number of tokens.

For creating the term document incidence matrix we first need to get the unique words in all of the documents and use them as columns for our matrix

Background pattern

Description automatically generated with low confidence

In the horizontal axis we put the documents ids.

Graphical user interface, application

Description automatically generated

Now we need to fill our matrix with zeroes and ones. Depending or not a term occurs in a document.

A whiteboard with writing on it

Description automatically generated with medium confidence

This is how the term document incidence matrix should look like. We can use this data structure to do our Boolean retrieval.

Now, if we want to look for the words Jackie AND Memphis we will take the vector for each row: “Jackie” and “Memphis” and we will do a bitwise AND

101

100

100 <- result

This means that we want to return document 1 because it contains both words.

If we want to do an OR query we would perform a bitwise OR.

It’s easy to perform **Boolean retrieval** queries once we have indexed our data in the **Term Document Incidence Matrix.**

From this **Term Document Incidence Matrix we can get Incidence Vectors (Term incidence vector) which are each row of binary values in our matrix. Each term can be mapped into a vector term = (x1, …xn). Where the x values can be either 0 or 1. Incidence vectors can be useful in Binary Independence Model (BIM) to model a probability that a document is relevant via probability in terms of the Term Incidence Vector.**

There are some limitations with this structure though. We are going to have many columns in our matrix as documents in our **corpora.** We are going to have as many terms in our rows as terms we have in our **Lexicon** (set of unique terms). We might have a very big and **sparse** (vas majority of the values in the matrix will be zero) data structure, and it contain mostly zeroes.

To counter this problem, we can use an **Inverted Index**. We will see that later.

# Inverted Index

In the Term Document Incidence Matrix (TDIM) we tokenized our terms and put them in the column. The size of the grows too quickly. And it is not feasible for retrieval systems with large **corpus**.

The **Inverted Index** will be much more efficient in terms of space for the task of indexing our terms from the **corpus.**

We have to tokenize our documents, where we have a token for each term. Along the vertical axis we will put every token we see in our corpus.

Text, letter

Description automatically generated

Associated with each of these terms, we will put the number corresponding to the document id:

Text, letter

Description automatically generated

Note that we have duplicate terms, and we have a pointer to the document id itself as opposed to the sparse representation with zeroes and ones for every document in our collection.

Now we need to order them alphabetically:

Text, letter

Description automatically generated

From here we create our **inverted index** structure. We will group by term. And that group term will point to an array like structure containing all the document ids where that term occurs in:

A whiteboard with writing on it

Description automatically generated with medium confidence

We have a dictionary of terms and a posting lists structures. We also need to store the document frequency for each term (the length of its posting list).

A whiteboard with writing on it

Description automatically generated with medium confidence

We store the document frequency to help us in the algorithms we will further use: merging posting lists more efficiently, relevance (term frequency, inverse document frequency (TF-IDF). Help us determine which terms in our corpus are more relevant, taking term frequency and inverse of document frequency), etc.

The **dictionary is often store in memory**, and the **posting lists are usually stored in disk** (access memory is more efficient than access from disk (even ssd)).

Chart

Description automatically generated with medium confidence

If we are going to perform Boolean Queries, then we need to think of an algorithm that will allow us to extract all of the document ids from the posting lists.