One general remark. Our text collection is small by today’s standards, and computers are very fast and are becoming faster with time, so I think we can ignore efficiency issues as long as this is not an issue. Doing so has the advantage that we can get fancy, functionality-wise, and that it is easier to try things out. Being efficient is a trade-off, and the most obvious one is that the more efficient the system is, the more complex the code becomes and the harder it is to maintain it.

Another aspect we can mostly ignore is the stability and the resiliency of the database. Good databases have buili

In information retrieval, we distinguish two basic kinds of search functionality, which can be subsumed as “text search” and “attributes search”. Text search involves searching with some degree of approximation: finding occurrences of a term in a text, or finding occurrences of a substring, etc. Per contrast, attributes search involves direct comparison between values, with operators like “equal”, “not equal”, “smaller than”, “greater than”, etc. This generally concerns numbers (dates, prices, etc.) or sets of strings where we want a direct comparison to be performed, for instance tags.

In practice, this distinction is not clear-cut, because it is common to use complex attributes like lists of numbers, etc. It still remains useful. Historically, information retrieval systems typically deal with text search, while traditional databases deal with exact values, but even this distinction is fuzzy because there is some overlap between the two types of software.

# Text search

Information retrieval systems index terms, much like indexes in books. The goal is also the same: instead of scanning each page looking for occurrences of a word, we just look at the index, saving much time. But it is difficult to determine word boundaries and Sanskrit and other Indic languages. These languages don’t lend themselves well to this. Let me assume for now that we have no way to determine word boundaries in these languages.

In our case, I believe we can do without an index. The first reason for doing so is that approximate matching of the kind I will propose to support cannot be optimized without an index. The dataset is small enough and computers are so fast that it is unnecessary to add this extra level of complexity, at least for now. I propose to use the most obvious solution, which is to process the full text for each request.

Text matching is fairly complicated in practice. One of the most difficult issues, as far as I’m concerned, is to map together several distinct representations of the same text. This mapping is necessary, for instance, when highlighting matching passages in search results and when searching several representations concurrently. In some cases, it is not possible to figure out what exactly matches in the input text.

## Exact matching

Fregex ggg

## Approximate matching aka fuzzy search

There are several ways to implement this kind of functionality, depending on what exactly we want to do.

### Levenshtein distance

#### Basic principle

When searching short strings (typically a word), the most used metric is the *Levenshtein distance* aka *edit distance*. In its most generic form, the Levenshtein distance between two strings A and B is the minimum number of *edit operations* required for transforming A into B, where edit operations are either:

1. an insertion: *god* -> *go****o****d*
2. a deletion: *go****o****d* -> *god*
3. a replacement: *hel****l*** -> *hel****p***

The Damerau-Levenshtein distance adds a fourth edit operation:

1. a transposition: *s****la****t* -> *s****al****t*

For instance, the Levenshtein distance between *sleep* and *slept* is two, because at least two edit operations are required for turning *sleep* into *slept*, to wit:

1. delete *e*: *sleep* -> *slep*
2. insert *t*: *slep* -> *slept*

#### Variations

It is possible to assign different costs to each edit operation instead of using a uniform one. For instance, we can assume that insertions produce very different strings and thus assign a cost of 2 to this operation, while keeping a cost of 1 for the other ones.

Edit operations can also be assigned costs that consider the characters that are inserted, deleted or replaced. For instance, we can assume that replacing *b* with *v* is more likely than replacing *b* with *k*, and thus assign a lower cost to the first transformation.

Assigning costs is not necessarily worthwhile. Ideally, costs should be derived from statistics computed on the corpus. For spelling correction tasks, there is plenty of data available, but we don’t have any.

I our case, I do not think it is useful to consider transpositions. It makes sense when the goal is to account for typing errors, where it is not unusual to swap letters, but we are not dealing with this scenario.

The Levenshtein distance is typically used for comparing a query term (that might be misspelled, etc.) submitted by the user to the terms in a lexicon. There are well-known, efficient algorithms for doing this. A more complicated use case of the Levenshtein distance is for searching substrings. The problem becomes, informally: find the substrings of the text B that closely resemble the string A. This is much harder to implement.

#### Approximate matching libraries

I have already written some code for the typical use-case viz. searching for a term in a dictionary. I have not used non-uniform costs but this is straightforward to supplement. However, to obtain a decent performance, costs should be hardcoded viz. added in the source code. I can also allow dynamic modifications but this is much more difficult to implement efficiently.

There are two available libraries for this: TRE (<https://github.com/laurikari/tre>) and Python’s regex library ([https://pypi.org/project/regex](https://pypi.org/project/regex/)). TRE has a smaller code base, which makes it easier to modify, thus I plan to use it. The regex library has more features but is also much more complicated. Another salient advantage of TRE is that is guaranteed to run in linear time, both for exact matching and approximate matching. The regex library offers no such guarantee, most likely because it is using backtracking (the most common technique used in regular expressions library), which runs in exponential team and is thus vulnerable to spamming. TRE comes with a command-line program agrep that behaves like grep but also supports approximate matching based on the Levenshtein distance. This can be used to test the functionality.

### Soundex-like matching

The Levenshtein distance is relatively generic and works reasonably well across languages. When we have some knowledge about the phonetics of the languages we want to search, it is also possible to simplify the representation of both the query string and the lexicon or corpus we want to interrogate before comparing them.

A classic example of this approach is the Soundex algorithm (<https://en.wikipedia.org/wiki/Soundex>). It was designed for searching names of American people. It basically defines classes of sounds (for instance, b, f, p and v are treated as the same sound) and drops insignificant sounds. The original string is turned into some kind of ASCII signature.

Elaborating this kind of thing is not easy and requires domain-specific knowledge. We want a good balance between precision and recall. It is difficult to define rules that work well across different languages. This requires experimentations. I have heard about Soundex-like experiments for Sanskrit at least, but it does not seem anything definitive has been formulated so far.

It is of course possible to use separate Soundex-like for each language, or group of language. In any case, you need to inform me about the rules appropriate to the languages we are dealing with. This requires expertise which I do not have.

#### What kind of approximate matching?

A clearly useful use case is searching with a mobile phone or similar device where it is generally not possible to input diacritical marks or non-ASCII characters. This is equally useful for all languages involved. Removing diacritical marks from a string can be done automatically to a fairly accurate degree, without using explicit mapping tables.

TODO: also deal with aspiration and doubling

### N-Grams-based matching

A third kind of approximate matching technique is useful when searching text units that are unambiguously delimited. For instance, the user might want to find all padas that resemble another one which he submits as query. In such cases, the Levenshtein distance is not really appropriate. The longer the text, the less useful it becomes, because it is very sensitive to word order.

A common way to deal with this is to use n-grams. For instance, the trigrams (when n = 3) of dharma dhar, arm, rma. The Jaro-Winkler distance compares sets of n-grams. The annoying aspect of this metric is that it does not deal well with repetitions.

## Matching across several clusters

Need backtracking if we want to support <choice>, I do not think it is reasonably feasible to modify TRE to do that without backtracking. But we should restrict backtracking and use some point of alignment.

# Attributes filtering