On Sanskrit and Information Retrieval

Michaël Meyer
Paris Diderot university
École pratique des hautes études / Paris
michael.meyer@etu.univ-paris-diderot.fr

Abstract

Many Sanskrit texts are available today in machine-readable form. They are of considerable help to philologists, but their exploitation is made difficult by peculiarities of the language which prevent the use of traditional information retrieval systems. We discuss a few possible solutions to improve this situation and present as well a number of strategies to increase retrieval efficiency.

1 Searching Sanskrit Corpora: Purposes and Difficulties

1.1 The Sanskrit Electronic Corpora

Philologists have nowadays at their disposal many digital resources for the study of Sanskrit literature. Among these resources, electronic texts are of peculiar importance. At the time of this writing, about 1,500 such texts are publically available, all electronic archives included,[1] for a total size of around 350 Megabytes of plain text data.[2]

The most comprehensive of these collections, both in terms of quantity and in the variety of subjects embraced, is probably the Göttingen Register of Electronic Texts in Indian Languages [GRETIL].[3] Of importance is also the digital library of the Muktabodha Indological Research Institute [MIRI],[4] which focuses on Tantric literature from the Medieval era. The Thesaurus Indogermanischer Text- und Sprachmaterialien [TITUS],[5] by contrast, mainly focuses on Vedic and Brahmanic literatures.

To our knowledge, the most recent digital library is the Search and Retrieval of Indic Texts [SARIT] repository,[6] managed by Dominik Wujastyk, Patrick McAllister and a few other scholars. At the time of this writing, it offers access to about sixty Sanskrit texts, all of which are encoded in XML format, according to the guidelines of the Text Encoding Initiative [TEI] standard.[7] Peter Scharf also provides texts in this format in his Sanskrit Library.[8] By contrast, most other online repositories typically provide their electronic texts in plain text format, in obscure *ad hoc* formats, or in HTML with very light markup.

Of a different genre are part-of-speech-tagged corpora. We know of only one, the Digital Corpus of Sanskrit [DCS],[9] elaborated by Oliver Hellwig. However, Huet and Lankri (2018) recently developed a Sanskrit corpus manager that provides access to a number of annotated sentences.

[1] It is difficult to give a reliable estimate of the number of *unique* texts input electronically, for two reasons. Firstly, because several scholars have input the same text, sometimes using the same edition, sometimes not, and under various formats. Secondly, because some texts are actually subsets of larger ones, such as the *Bhagavadgītā* relative to the *Mahābhārata*.

[2] By plain text, we here mean Sanskrit text in the International Alphabet for Sanskrit Transliteration [IAST], encoded in UTF-8 and devoid of markup data such as XML tags.

[3] `http://gretil.sub.uni-goettingen.de`.

[4] `http://muktalib5.org/digital_library.htm`.

[5] `http://titus.uni-frankfurt.de`.

[6] `http://sarit.indology.info`.

[7] `https://tei-c.org`.

[8] `https://sanskritlibrary.org`.

[9] `http://www.sanskrit-linguistics.org/dcs`.

1.2 The Importance of Electronic Corpora for Sanskrit Studies

Electronic texts are very useful to philologists. Indeed, philological work in its two forms—edition on the one hand, interpretation and exegesis on the other—requires to discover textual parallels.[10] This is particularly important in the case of Sanskrit literature, because it is rife with citations and glosses. Indeed, scholiasts often cite excerpts from the literature, or paraphrase them, when commenting a text. Identifying the source of these citations can prove difficult, if only because merely vague references to the quoted work are often provided.[11]

Broadly speaking, we can distinguish two types of philological enquiries.

Firstly, searching for textual parallels, i.e., finding the source of a citation, or, conversely, checking whether a passage from a given text is cited elsewhere. These enquiries usually help to reconstruct corrupt passages or to amend them. They are also useful to obtain a better understanding of the meaning of obscure passages, because the original context of the passage or its exegesis in the scholastic literature generally provide crucial information. Finally, they are of considerable importance to estimate the dates of an author or of a text: checking which texts an author cites, and, conversely, which texts cite him, is one of the most effective ways to estimate his date.

Other inquiries are more linguistic in nature. They typically aim at understanding the meaning of rare syntagms, or the meaning of syntagms that are somewhat common in the literature but possess a technical signification in specific texts. This type of philological work is at the origin of the *Tāntrikābhidhānakośa* project, which aims at creating a lexicon of the Tantric terminology. The editors themselves take notice of the importance of electronic texts in their preface to the third volume of the work (Goodall and Rastelli, 2013, 9):

> Whereas the initiators of this project worked with notes and card-indices that they had compiled over a life-time of reading, we are faced with dozens, hundreds, or sometimes even thousands of usages of a given tantric expression at the touch of a search-button. Many instances are therefore inevitably unfamiliar to us, but we must at least attempt to take what is relevant into account. Searching through an electronic library with "grep" thus has considerable and obvious advantages, but carries with it an obligation to take into account more passages than we would otherwise encounter. Furthermore "grepping" is especially helpful for revealing the contours of evolutions in usage for certain expressions.

1.3 Limits of Pattern Matching Tools

To assist philologists in their work, the development of full-text retrieval systems is important. A few have been written over the years, usually to provide search interfaces to specific text collections.[12] Despite the existence of these systems, most researchers generally use pattern matching tools such as `grep`, if only because they are more readily available and allow them to search into their private collection of documents or in their research notes.

These tools, however, are not practical for searching Sanskrit texts. Not so much because of speed issues, since pattern matching engines are nowadays highly optimized and since the volume of data is small enough to be fully cached in-memory, even on a low-end computer. But because their matching strategy, as well as their display facilities, are closely tied to the input data format. Searching Sanskrit documents in several formats, not even talking about distinct transliteration schemes, is generally very messy. Many possible query matches are usually missed because of intricacies of the text representation, such as the use of whitespace or the introduction of special symbols and annotations within the text. If the IAST is used, queries can also return more

[10] We borrow this distinction from Pollock (2018).

[11] For instance, a considerable number of citations are introduced with the words *tad uktam* "this has been said," which tell us absolutely nothing about the origin of the citation.

[12] We present two of them below in section 2.2.

matches than expected when they start or end with a phoneme which textual representation is a substring of another phoneme, as is the case of the simple vowel *i* relative to the diphthong *ai*.

These issues could be alleviated by preprocessing documents and transliterating them to a simplified version of the Sanskrit Library Phonetic Basic encoding scheme [SLP1] (Scharf and Hyman, 2012, 151–158), which possess the useful property that it needs a single code point—in fact, a singly byte—to represent a Sanskrit phoneme. Amending the original documents would however likely cause problems, if, for instance, annotations in English appear within Sanskrit passages; furthermore, the encoding itself, while convenient for machines, is noticeably hard to decipher for a human reader. A better solution would be to write a pattern matching engine that runs its matching algorithm, not on the actual text, but on a logical representation derived from it on-the-fly. To our knowledge, however, this approach is almost never chosen, probably because any non-trivial preprocessing would considerably impede the performance of the engine. Complicated queries are usually relegated to database systems, or, less frequently, information retrieval systems.

1.4 Difficulties in Indexing Sanskrit Texts

It would thus be beneficial to use a real information retrieval system, both for the sake of efficiency and for the sake of flexibility. But the Sanskrit language does not lend itself easily to text retrieval, because indexing a document generally presupposes that it is possible to recognize its lexical units. This process is straightforward in a lot of languages, but is however highly ambiguous in Sanskrit. The difficulties involved are due to two principal reasons: the scarcity of explicit word boundaries, and the existence of euphony phenomena.

In Indic scripts, word boundaries are indeed not necessarily made explicit with whitespace or punctuation characters. Sequences of graphemes thus do not represent words, properly speaking, but rather sequences of one or more words. We call these *clusters*, by analogy with the meaning of the term in musical terminology, where it designates a group of adjacent sounds.

To facilitate reading, researchers usually introduce, while transliterating a text, as much boundaries as possible, typically by adding whitespace characters, as shown in table 1. We call this process *ungluing*. More specifically, we say that a text is *unglued* if no more boundaries can be introduced into it without altering the graphemes that represent phonemes.

Devanāgarī	सोममय इति दर्शङमङ्गीकृतमत्र ।
Transliteration (glued)	*somamaya iti darśaṅamamaṅgīkṛtamatra* |
Transliteration (unglued)	*somamaya iti darśaṅam aṅgīkṛtam atra* |

Table 1: Several possible representations of a Sanskrit passage. Excerpt of Vṛṣabhadeva's commentary on the first chapter of Bhartṛhari's *Vākyapadīya* (Iyer, 1966, 201, l. 9).

Despite the advantages of this ungluing process, some researchers or copists do not introduce, in transliterated texts, boundaries that were not present in the original text. The rationale for this practice, if any, is unclear to us. A possible explanation could be that preserving the transliterated text in its original glued form eases roundtrip conversions between the original Devanāgarī and its transliterated representation.

A more serious issue resides in the fact that phonemes around word boundaries and at the end of a phrase, before a punctuation mark, can be modified as a result of euphony phenomena (*sandhi*). These transformations obey a set of rules, which can be compactly represented with the notation $\alpha|\beta \to \gamma$;[13] the vertical bar here stands for a word boundary, the variable α represents the end of the left word, β the start of the right one, and γ the result of the application of the rule. To be more accurate, we represent here with α and β the shortest possible strings of phonemes that need to be considered for applying the rule.

[13]We borrow this notation from Gérard Huet.

For our purpose, it is convenient to distinguish three basic types of *sandhi* rules. A few of them, all of which have in common that they operate on vowels, produce as output a single phoneme. This is the case of the rule $a|\bar{\imath} \rightarrow e$, for instance, which dictates that the words *deva* 'god' and *īśvara* 'lord,' when written in sequence, form the string *maheśvara*. But most other rules produce as output a sequence of two or more phonemes that can be unglued in transliterated texts. We write these rules with the notation $\alpha|\beta \rightarrow \alpha'|\beta'$; the vertical bar on the right side of the arrow indicates that it is possible to unglue the text at this point, while α' and β' denote the transformation of α and β, respectively. The rule $ḥ|c \rightarrow ś|c$, for instance, belongs to this category; it dictates that the words *devaḥ* 'god' and *ca* 'and,' when written in sequence, form either the string *devaśca* or *devaś ca*. Finally, a few rules do not involve any gluing. They are applied at the end of a phrase, before a punctuation mark. We represent them with the notation $\alpha|\emptyset \rightarrow \alpha'|\emptyset$, were the symbol $\emptyset$ represents the absence of a phoneme.

Despite the difficulties involved in segmenting Sanskrit texts, programs have been developed to address the issue. Gérard Huet (2003; 2005) thus elaborated an unsupervised parser based on finite-state technologies. By contrast, Oliver Hellwig (2009; 2010) developed a supervised parser based on a hidden Markov model, which he trained on manually annotated sentences from the DCS. These tools are of considerable help for computer-assisted linguistic tasks, but it does not seem to us that they are currently robust enough to be used autonomously, without human supervision, for indexing tasks. Gérard Huet's segmenter—the only one that can be used programmatically at the time of this writing—indeed operates on a finite vocabulary and with a finite set of *sandhi* rules and inflection rules, so that a single unknown word, peculiar form or typing error prevents the segmentation of a full cluster. This is aggravated by the fact that the strings that are worth looking for in an index are typically rare words or syntagms, names of persons, etc., which are the most likely to not be recognized correctly by a tokenizer. Furthermore, many electronic texts contain corrupt passages—either because of typing errors, or because the original manuscript from which the text was copied is itself damaged or corrupt.

Most issues involved in indexing Sanskrit texts would go away if the electronic texts at our disposal were all exhaustively segmented. We do have access, in fact, to segmented electronic texts, most notably the word-reading (*padapāṭha*) of the *Ṛgvedasaṃhitā* and the annotated texts of the DCS. But they only form a small subset of the available electronic texts, and it is unreasonable to assume that this situation is going to evolve significantly in the near future. For the time being, we should thus be content with the data available, and try to make the best of it.

2 State of the Art

2.1 Basic Structure of an Information Retrieval System

An information retrieval system, at the very least, consists in an index that maps a set of strings to lists of sorted integers that represent the documents these strings occur in. Generally, the offsets at which each string occurs within each document are recorded as well, so as to make possible phrase searches. These lists of occurrences, technically called *postings lists*, are typically represented as arrays of variable-length integers. The index proper is represented as a dictionary-like data structure, a B^+ tree for instance.

When the documents to index are texts, as is the case for us, an index typically stores the terms that appear in the documents collection, or at least some useful representation of them, such as their stem. But this is in no way mandatory. In particular, a few experimental XML retrieval systems (Büttcher and Clarke, 2005; Strohman et al., 2005) index as well the structure of the document, typically by treating XML tags as if they were terms. This makes possible structured queries with arbitrary nesting of the kind supported by XPath expressions.

Nevertheless, most text retrieval systems available today use a flat data model where structural information is encoded as part of each term, typically by prefixing the term with a binary string that represents the section of the document the term occurs in. This approach is more convenient

to implement and generally reduces the time necessary for evaluating a query. We will soon see that segregating structural information from terms is nonetheless very useful in practice, even for flat text documents that do not have an explicit structure.

2.2 The Existing Sanskrit Information Retrieval Systems

To our knowledge, two Sanskrit text retrieval systems use an information retrieval architecture instead of a pattern matching tool or a traditional database system.

The Gaveṣikā system (Srigowri and Karunakar, 2013) allows searching for the inflected forms of a nominal or verbal stem and supports as well spelling variations. This functionality is implemented by ungluing the text at indexing time[14], and, at search time, by expanding the stem submitted as query to its inflected forms and to alternate spellings of these forms, with the help of a morphological generator. This expansion process does not cover phonetic transformations that result from the application of *sandhi*, so that a number of results are typically missed. Nevertheless, the recall of the system is very high, on par with the DCS word retrieval facilities.

The SARIT corpus also makes use of an information retrieval system, the most interesting feature of which is the support of document attributes. Its indexing strategy is not described anywhere, but we can reasonably assume, by looking at the website documentation and at search results, that the unit of indexing is a cluster. Searching for a string in such a way that all its occurrences are returned thus requires adding wildcards on each side of it, as in **mukha** 'face,' for instance. This somewhat defeats the purpose of using an information retrieval architecture, if only because of efficiency reasons. Indeed, searching for a query string with a leading wildcard typically involves a full traversal of the terms dictionary, followed by a costly merge operation.[15] Searching for the string *mukha* in the GRETIL corpus, for instance, would require examining a dictionary of about 2,785,000 clusters and merging about 7,000 postings lists.

We initially wrote our own retrieval system in 2017, as a practical and convenient replacement for traditional string matching tools. It supports searching for arbitrary substrings, while being aware of gluing phenomena and of phonetic transformations that result from the application of *sandhi*. The indexing strategy we chose at the time was elaborated to maximize recall, in such a way that no potential match can possibly be missed, provided that *sandhi* application between the words in the query is deterministic. We were primarily concerned with this completeness guarantee because it is of the utmost importance when searching for textual parallels. However, we did not pay much attention to the precision of the system. Improving it while still maintaining this completeness guarantee indeed creates a host of new difficulties, as will be evident from our discussion below.

3 Adapting Information Retrieval Techniques to Sanskrit

3.1 Substring Search

We explained above in section 1.4 that segmenting a Sanskrit text accurately is a difficult task. For the sake of retrieval, however, it is not necessary to segment texts in a way that is linguistically meaningful. We can make possible arbitrary substrings searches, without tokenizing the text in lexical units. This is usually done by indexing the n-grams of a document, that is to say, all substrings of length n this document contains.

In information retrieval, the item n stands for is usually a character, sometimes a word. In our case, n represents Sanskrit phonemes, which map to variable-length sequences of bytes in the source text. Within the index, we represent phonemes as code points in one of the Unicode private-use areas,[16] so that it is possible to index both phonemes and assigned code points together while using the UTF-8 encoding for compressing strings. We currently support as input

[14] This detail is not mentioned in the paper, but is patent from the actual implementation: `http://scl.samsaadhanii.in:8080/searchengine`.

[15] It is however possible to make wildcards lookup run in sublinear time, for instance by using the technique described in section 3.1.

[16] `http://www.unicode.org/faq/private_use.html`.

transliteration scheme all the variants of the IAST that are actually used in electronic texts.[17] Support for new transliteration schemes could easily be added, by writing a new transliteration state machine or by extending the existing one.

We use trigrams ($n = 3$) as basic retrieval units, as a compromise between speed, usability and simplicity. Searching for strings which include less than three phonemes is not very useful in practice, except for monosyllabic *mantra*s (*bījamantra*) which comprehend exactly two phonemes, such as AIṂ.

Searching with n-grams is conceptually equivalent to matching phrases. For instance, the query *mantra* can be evaluated by retrieving the postings lists of the trigrams *man*, *ant*, *ntr* and *tra*, and by examining them concurrently so as to find a window of length 6 where these n-grams occur, in this order. Given that n-grams overlap when $n > 1$, it is in this case unnecessary to look for all n-grams in the query string to satisfy the query. To retrieve the documents that match the query string *mantra*, for instance, searching for a window of length 6 where the trigrams *man* and *tra* occur in this order is sufficient.

3.2 Handling of Cluster Boundaries

We have so far explained how to support matching strings of phonemes. We should also discuss what to do with cluster boundaries, i.e., the substrings of the indexed text that do not represent phonemes—most notably, whitespace characters. Given that Sanskrit electronic texts are not necessarily unglued, we want cluster boundaries to be treated as optional during matching. For instance, we want the query string *punar api* to match documents that do contain exactly the string *punar api*, but also those that contain the string *punarapi*.

To address this issue, we first used a crude, but somewhat effective, approach: ignore all cluster boundaries in both the indexed text and the query, thus treating *punar api* and *punarapi* as equivalent, for instance. However, this results in false positives when it happens that two or more clusters, when joined together, contain as a substring what could be a valid word in another context, but is not in this peculiar case. For instance, searching for the term *nara* 'man' returns documents that contain the string *punar api*, because this string is interpreted internally as *punarapi*, which contains *nara* as a substring. If the text is originally glued, as in the string *punarapi*, it is of course impossible to prevent such false positives without a full-fledged segmentation module. But we can at least prevent them when boundaries were introduced in the text that allow us to do so. For this to be possible, the query string itself should be unglued. In the remainder of this paper, we will assume that this is necessarily the case.

A possible way to prevent this kind of false positives is to include cluster boundaries in the n-grams generated at indexing time, and to amend the user query in such a way that the cluster boundaries it contains need not be present in the text for a document to be considered matching. Representing cluster boundaries inside the n-grams generated at indexing time involves interpreting sequences of characters that do not represent phonemes as a single character, say _, and to emit n-grams as usual. The string *punar api*, for instance, thus results in the trigrams *pun*, *una*, *nar*, *ar_*, *r_a*, *_ap* and *api*. Interpreting the user query in such a way that cluster boundaries are optional at search time is, however, more involved. The simplest solution would be to generate all possible forms the query string could take on inside a document, and search for the union of these strings. Following this process, the query string *punar api ca*, for instance, would be expanded to the union of the strings *punar_api_ca*, *punar_apica*, *punarapi_ca* and *punarapica*, all of which would then be segmented into n-grams for evaluating the query.

This method, however, would produce a highly redundant query. A convenient way to improve it becomes readily apparent if we observe that we ultimately want to produce a query that is semantically equivalent to a regular expression where cluster boundaries are optional, such as *punar_?api_?ca*. In other words, we want to produce a query that recognizes the language of

[17] A few phonemes can be written in different ways, for instance the *anusvāra*, which is represented as *ṁ* or *ṃ* depending on the source text. We systematically ignore Vedic accents.

the finite-state automaton denoted by such a regular expression. Instead of selecting n-grams from the query string proper, we can thus convert it to a finite-state automaton, amend this automaton to make cluster boundaries optional, determine and minimize it so as to obtain an automaton similar to the one presented in figure 1, and finally extract n-grams directly from this representation.

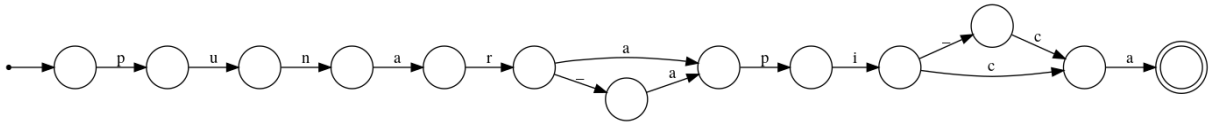

Figure 1: Minimal acyclic finite-state automaton denoted by the regular expression *punar_?api_?ca*

There is however a simpler solution to the problem of cluster boundaries. Instead of representing cluster boundaries inside the n-grams generated at indexing time, we can instead keep indexing texts as if they did not contain any cluster boundaries, and index separately cluster locations. To do that, it is necessary to amend the n-grams tokenizer so as to make it emit a special token C each time a new cluster is encountered, with the same position as the first n-gram in the cluster. The resulting postings list C thus records the start offset of all clusters in the document collection. Query strings must be processed in a similar way, so as to obtain a list of n-grams to look for, on the one hand, and a list of cluster start offsets, on the other. The evaluation of the query follows the process we described above in section 3.1, up to the point where an interval $[a, b]$ of the document that matches the query n-grams is delimited. At this point, an additional test is required to determine whether the segmentation of the text is compatible with the one of the query: if all cluster boundaries c that occur within the delimited interval of the document such that $a < c < b$ also appear at the same relative position in the query, the delimited passage can be considered a match; otherwise, it must be discarded.

Compared to the first solution, this approach presents the disadvantage that it requires additional storage. But it can also be much faster to execute, provided that the index layout is modified in the way described below in section 4.3.

3.3 Handling of *Sandhi*

The most vexing difficulty to take care of is however the handling of *sandhi*. Briefly put, we want a query string to match, not only its original form, but also all the forms it could take on inside a text as a result of the application of *sandhi*. For instance, we want the query *devaḥ* to match, not only the string *devaḥ*, but also the strings *devaś* in *devaśca* 'and the god,' *devo* in *devo'pi* 'but the god,' and so on.

3.3.1 Noncontextual *Sandhi* Expansion

In our first attempt at the task, we used a simple generation approach that only takes into account a single side of each *sandhi* transformation rule, treating the other as if it did not matter for the application of the rule. To be more explicit, we treated rules of the type $\alpha|\beta \rightarrow \gamma$ as if they could be read as $\alpha|* \rightarrow \gamma$ or $*|\beta \rightarrow \gamma$, where the wildcard symbol * stands for an arbitrary sequence of zero or more phonemes; similarly, we treated rules of the type $\alpha|\beta \rightarrow \alpha'|\beta'$ as if they could be read as $\alpha|* \rightarrow \alpha'|*$ or $*|\beta \rightarrow *|\beta'$; and we treated rules of the form $\alpha|\emptyset \rightarrow \alpha'|\emptyset$ as $\alpha|* \rightarrow \alpha'|*$.

To implement this approach, we wrote a *sandhi* application module in the most straightforward way,[18] and systematically exercised it so as to generate two lookup tables T_{left} and T_{right}. T_{left} maps a given phoneme to the set of forms it could take on as a result of *sandhi* application when it occurs at the beginning of a word. Conversely, T_{right} maps sequences of one or two phonemes

[18] At the time we started developing our system, Gérard Huet's *sandhi* engine was not yet publicly downloadable.

to the forms they could take on when they occur at the end of a word. A record of each table is reproduced in table 2, together with sample rules from which each record entry was derived.

T_{left}	
Entry	Sample rule
$\bar{u}$	$u\|u \rightarrow \bar{u}$
o	$a\|u \rightarrow o$
u	$k\|u \rightarrow k\|u$

T_{right}	
Entry	Sample rule
v	$u\|a \rightarrow v\|a$
uv	$u\|a \rightarrow uv\|a$
$\bar{u}$	$u\|u \rightarrow \bar{u}$
u	$u\|k \rightarrow u\|k$

Table 2: *Sandhi* expansion of the phoneme u in T_{left} and T_{right}

At query time, we look up the beginning and the end of the query string submitted by the user in the tables T_{left} and T_{right}, respectively, and use the retrieved data to construct a query that matches all possible forms the original query string could take on inside a text. This generation process is performed by creating, from the user query and the data retrieved in T_{left} and T_{right}, a minimal acyclic finite-state automaton similar to the one presented in figure 2, before extracting n-grams from this representation, in the manner described above in section 3.2.

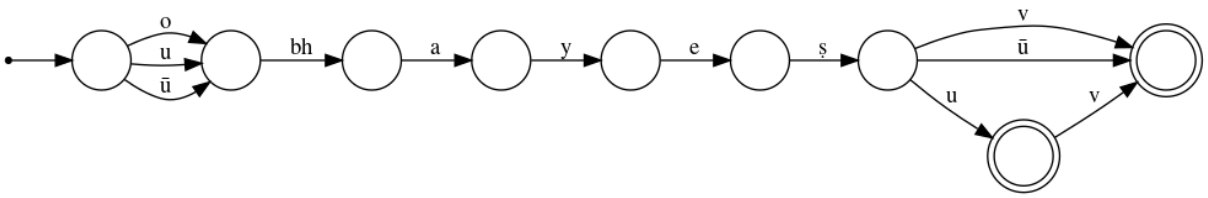

Figure 2: The query string *ubhayeṣu* after *sandhi* expansion

This *sandhi* expansion process does not require additional storage and is fast enough to be performed online, because the number of forms α and β can take on after *sandhi* application is small in practice. It is, however, incomplete, since it does not attempt to apply *sandhi* between the words of a query string, essentially ignoring the fact that *sandhi* application is a nondeterministic process and could thus produce several distinct query strings. Furthermore, it also returns many false positives, since it does not take into account the phonetic context of the query string within a document.

3.3.2 Contextual *Sandhi* Expansion

The errors generated by our *sandhi* expansion technique fall into three basic categories:

1. To begin with, *sandhi* expansion is performed even at the very beginning of a phrase, where no *sandhi* can possibly occur. The query string *iti* 'thus' thus ends up erroneously matching the verb *eti* 'he goes' when this verb appears at the beginning of a phrase, because of rules such as $a|i \rightarrow e$.

2. Similarly, false positives can occur at the end of a phrase, before a punctuation mark. In this configuration, the only rules that should be taken into account are those of the form $\alpha|\emptyset \rightarrow \alpha'|\emptyset$.

3. But the vast majority of false positives occur within clusters. For instance, the query string *devaḥ* ends up incorrectly matching the string *devaśabda* 'divine sound,' because *devaḥ* is expanded to *devaś* due to the rules $ḥ|c \rightarrow ś|c$ and $ḥ|ch \rightarrow ś|ch$, and because *devaś* is a substring of *devaśabda*.

False positives of the types 1 and 2 could be addressed to by implementing a filtering mechanism similar to the one we described above for cluster boundaries. To make this possible, the start and end positions S and E of each *daṇḍa*s-delimited text segment should be recorded in the index, and n-grams in the query tree should be annotated to reflect the conditions under which they can be considered to match. For instance, the trigram *eti* generated from the query string *iti* should be annotated with a flag that dictates that it can only be considered to be a match if no postings in S possess the same position. Similarly, the trigram *evo* generated from the query string *devas* and the rule $as|a \rightarrow o|'$ should be annotated with a flag that dictates that it can only be considered to be a match if E does not occur three positions ahead of it.

This solution is feasible for false positives of the types 1 and 2, but not so much for those of the type 3. The main problem is that performing contextual checks becomes in this case prohibitively expensive. To test whether the transformation of a k to a g at the end of a word is appropriate in a given context, for instance, we would have to check the postings list of about 27 phonemes. The cost of query evaluation could be improved by indexing phoneme classes so as to reduce the number of postings list that need to be examined concurrently. If, say, all sonants except nasals were indexed under a single postings list S, we could determine whether the transformation of a k to a g at the end of a word is appropriate by examining just S.

However, it seems to us preferable to take the reverse approach, that is to say, to resolve possible results of *sandhi* application at indexing time. To test this approach, we wrote a transducer that maps each possible string that could result from the application of *sandhi* to the set of rules that could have generated it. Instead of attempting to determine whether a given reading is correct, as would a full-fledged tokenizer, we assume it necessarily is, and index it as such. To be more specific, we generate two extra tokens α_{right} and β_{left} that represent respectively the values α and β of *sandhi* rules of the form $\alpha|\beta \rightarrow \gamma$ or $\alpha|\beta \rightarrow \alpha'|\beta'$, each time such a rule is recognized in the source text. These extra tokens are affected the same position as the string γ, α' or β' they stand for. For instance, the phoneme $\bar{a}$ in the word *uvāca* triggers the generation of the tokens a_{left}, a_{right}, $\bar{a}_{\text{left}}$ and $\bar{a}_{\text{right}}$, with duplicates removed. Similarly, we emit one extra token α_{right} each time a *sandhi* rule of the form $\alpha|\emptyset \rightarrow \alpha'|\emptyset$ is recognized in the source text.

With this approach, the query process is greatly simplified. We start by looking up the first phoneme of the query string in the index so as to retrieve its postings list of the form β_{left}, if any. The same operation is performed for the last one or two phonemes of the query string, so as to retrieve the corresponding postings list of the form α_{right}. The remainder of the string is then segmented into trigrams as usual, and a phrase query is finally constructed from these three types of elements. For instance, the query string *ubhayatas* results into the four tokens u_{left}, *bhay*, *yat* and as_{right}, which are then combined to construct a phrase query.

This approach is appropriate for a small number of documents, but might be less feasible for a large documents collection. Indexing *sandhi* rules indeed considerably increases the size of the index and produces huge postings lists, an effect that is compounded by the fact that, to make possible searching for strings which comprehend between 3 and 6 phonemes included, bigrams ($n = 2$) and unigrams ($n = 1$) must also be indexed. To support a real workload, it might be necessary to prune bigrams and unigrams that are useless for matching the text; if, for instance, a string of three phonemes that cannot possibly involve phonetic modifications—say *abhi*—appears at the beginning of a verse, indexing its bigrams and unigrams is unnecessary, because its trigram will necessarily be selected over them at search time.

3.4 Searching for Inflected Forms Given a Stem

When searching for a peculiar syntagm, as opposed to a phrase, it is often desirable to obtain, in the set of search results, documents that contain various forms of this syntagm, such as its plural form, instead of just the specific form that was submitted in the query. This functionality is important for the Sanskrit language, because its morphology is particularly rich.

To make possible this type of functionality, it is customary to use a stemmer, that is to say,

a program that maps an inflected form to its stem, or at least to a string that is not a valid stem but that could stand for it in some way.[19] This approach is of course only feasible if lexical units can be distinguished in the first place. It is thus not suited to the indexing framework we described above.

However, it is possible to use the reverse approach, that is to say, to generate, at query time, all possible inflected forms of a stem submitted as query, and to search for the union of the resulting strings. This is the approach used by Srigowri and Karunakar (2013). The same strategy can be used with our indexing scheme, save for the fact that each generated inflected form must itself be segmented into *n*-grams. To generate a compact query tree, we can construct a minimal acyclic finite-state automaton such as the one presented in figure 3 and extract *n*-grams from this representation, as done above in section 3.3.1.

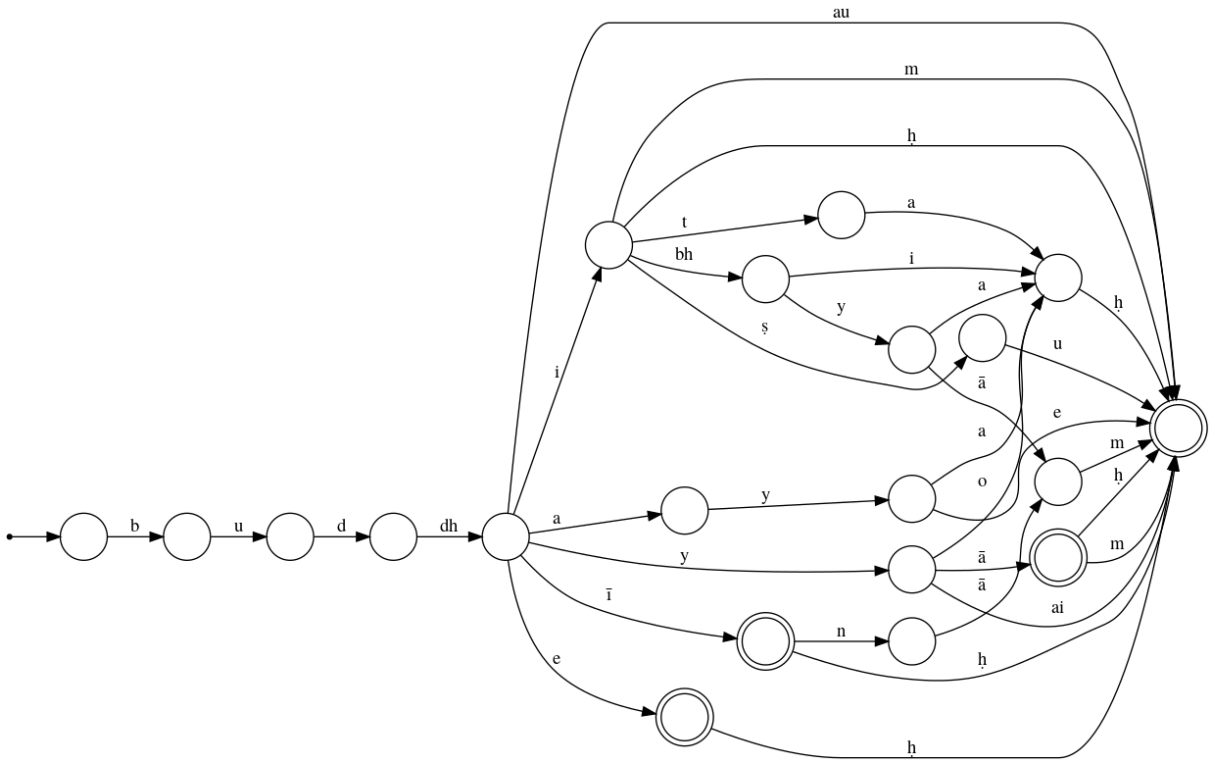

Figure 3: Inflected forms of the nominal stem *buddhi* (fem.) 'intelligence' represented as a finite-state automaton

4 Increasing the Efficiency of the System

4.1 Combining Documents Representations

We assumed so far that existing tokenizers for the Sanskrit language are not yet robust enough to be used autonomously for indexing tasks. This does not imply, however, that they should not be used at all. Relying exclusively on the output of a real tokenizer would not suffice, for the reasons given above in section 1.4. But we could still index both the output of such a tokenizer and the *n*-grams generated as described above.

This strategy has been used with some success in Chinese information retrieval (Zhang et al., 2000), and would thus probably benefit us too. However, the very process of indexing several readings of the same text is in itself technically challenging, because *sandhi* application often modifies the length of a word, which complicates phrase matching. We have not yet elaborated a robust enough strategy to address this issue. Furthermore, it is not yet clear to us how exactly

[19] Algorithmic stemmers such as the one of Porter (1980) do not necessarily return valid stems.

the output of a full-fledged tokenizer could be used to produce a better ranking of possible query matches. At the very least, we could provide to the user a visual cue of the estimated correctness of a match by highlighting possible matches with different colors, or maybe different shades of the same color.

4.2 Optimizing the Selection of *N*-grams

In all the information retrieval systems we have studied so far, no special attention is given to the fact that query strings which contain a number of characters that is not a multiple of n when $n > 1$ can be segmented in different ways while minimizing the number of n-grams in the query. Furthermore, it is implicitly assumed that pruning as many n-grams as possible in the query string is necessarily beneficial, as far as retrieval time is concerned.

This assumption, however, is not necessarily correct. For the time of retrieval is by far dominated by the decoding of postings lists, not by the lookup of n-grams in the index. A more accurate estimation of the cost of the evaluation of a query can thus be obtained by examining the frequency of each query n-gram in the collection, an information that is usually readily available in the index data structure. Instead of arbitrarily pruning n-grams, we should thus attempt to select the combination of n-grams that minimizes the overall number of postings to be decoded while satisfying the query. This amounts to interpreting the user query as if it was a directed acyclic graph, where vertices represent n-grams and edges represent the length of the postings list of the target n-gram, so as to find the shortest path from the first n-gram to the last one.

An example is given in figure 4, for the query string *mantraḥ* and $n = 3$. Edges are annotated with the actual frequency of the n-gram they point to in the GRETIL corpus. The cost of decoding the postings list of the first trigram *man* can be ignored, just as the cost of decoding the postings list of the last trigram *raḥ*, since both trigrams must necessarily be selected for the query to match. In this case, the optimal path is 0:*man* → 2:*ntr* → 4:*raḥ*, which involves decoding $203,356 + 37,885 + 76,636 = 317,877$ postings. By contrast, choosing the path 0:*man* → 3:*tra* → 4:*raḥ*, which also involves the minimum number of trigrams necessary to match the query, would lead to decoding nearly two times more postings.

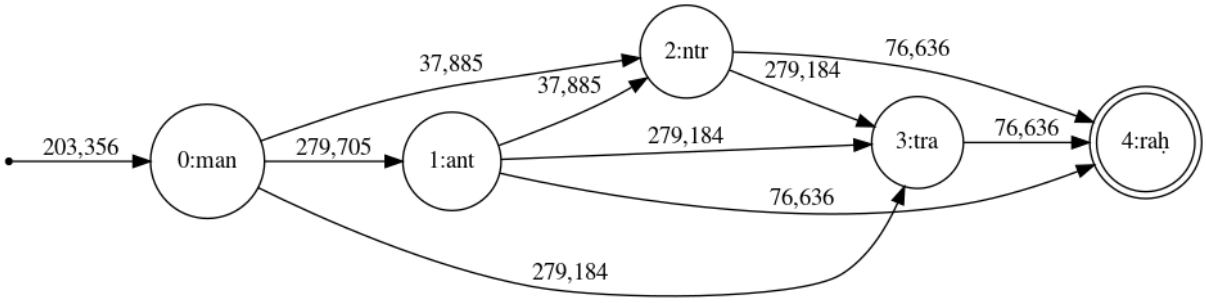

Figure 4: The query string *mantraḥ* represented as a directed acyclic graph of trigrams

A further optimization is also possible, this time for all $n > 0$, when a query string contains several occurrences of a given n-gram. In this situation, query tree nodes can be shared, so that the postings list of a n-gram that occurs several times in the query need to be decoded only once. Accordingly, the contribution of a given n-gram to the cost of query evaluation should be taken into account only once during the n-grams selection process.

4.3 Optimizing the Representation of Postings Lists

The indexing strategies we discussed so far are very costly in terms of processing time if a traditional index layout is used for representing postings lists. Indeed, lists of documents identifiers are usually separated, both conceptually and physically—on disk or in-memory—from the lists of

integers that represent the positions of a given term within a given document. The primary motivation for this data layout is the idea that the queries that do not involve positional matching should not incur the cost of decoding lists of positions. In our case, however, positional queries are almost always necessary. We would thus benefit from inlining lists of positions within lists of documents.

It seems however beneficial, in terms of implementation complexity and in terms of expressiveness at the very least, to go one step further and merely index positions, essentially treating the documents collection as if it was a single long document. Doing this allows better granularity at retrieval time. If documents boundaries are indexed in the way we proposed to index clusters previously in section 3.2, it indeed becomes possible to constrain matching, not merely to a single document, but also to a sequence of documents. Other structural information could be indexed as well, such as verse boundaries or paragraph boundaries, to make possible more expressive queries.

But the main advantage of this index layout lies in the fact that it lends itself more conveniently to the optimization of positional queries. For these queries can often be evaluated without reading in full the postings lists of the terms they are made of. To give but one example, if a document contains the string *a a a a a a a b* and we are looking for the phrase *a b*, all postings of *a* up to the last one are irrelevant, and thus we can jump directly to the last occurrence of *a*. For this to be possible, postings lists must be made addressable, at least to some degree.

To address this problem, Moffat and Zobel (1996) propose to split each postings list into several chunks, and to prefix each of these chunks but the last one with the identifier of the first document in the next chunk, together with a pointer to this chunk. This solution can of course be extended to positions lists. It saves processing time, since chunks that are irrelevant for the evaluation of a query need not be decoded, and can be skipped over. However, it does not save much disk or memory bandwidth, if at all, because a chunk, or at least its initial part, must necessarily be fetched from disk or from memory in order to retrieve the location of the next one. This suggests that we should store chunks pointers separately from the chunks themselves.

To do that in a way that is amenable to disk storage, we propose to store postings lists contiguously on disk, while introducing a logical separation in fixed-size pages. A single postings list can thus cross several pages, and, conversely, a single page can hold several postings lists. The postings lists that cross several pages can then be indexed by allocating new pages and store there pointers to each page the list occupies at the lower level, together with a copy of the first position of each delimited chunk of the list at the lower level. This process can be repeated again to introduce further levels of indexing. Briefly put, this amounts to constructing a kind of skip list (Pugh, 1990) that is laid out similarly to a B^+ tree, over the whole set of postings lists in the index.

An example of this data layout is given in figure 5. It describes a two-levels index that contains six pages, on which are laid out the postings lists of three distinct terms *x*, *y* and *z*. Positions that belong to the same postings list are represented with the same color.

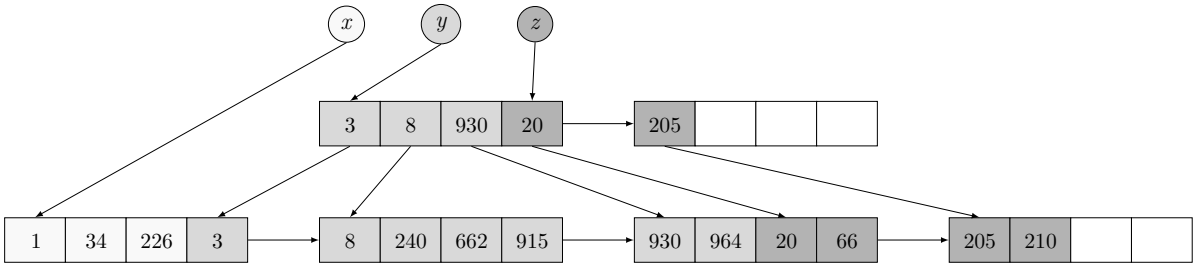

Figure 5: Representation of postings lists

Conclusion

We have discussed in the above several possible ways to model, in an information retrieval system, a few peculiarities of Sanskrit phonetics and morphology, and described as well a number of possible optimizations to reduce processing time. This however only scratches the surface of the functionalities an information retrieval system is expected to provide.

The most pressing goal, for the time being, is to elaborate an architecture that strikes a good balance between the system's precision, its recall, and its efficiency in terms of time and space. In particular, the interaction of the strategies we described above deserves special consideration, because their compounding effect can easily lead to excessively complicated queries. It might be necessary to adopt several distinct retrieval strategies depending on the query and the user's expectations. To help alleviate the issue, it is desirable to give more control to the user over the query process, so that he can choose whether a quick but possibly incomplete or inaccurate answer is preferable to a more accurate, but slower one. Accordingly, it is necessary to elaborate an evaluation methodology to test the time and space efficiency of the system.

Much also remains to be improved in the area of query expressivity. We did not discuss how to support, within our framework, Boolean operators, proximity operators and containment operators—searching within a given number of documents, of paragraphs or of lines, for instance. It is also desirable to formalize a query syntax that gives full control to the user over the search process; most notably, over its linguistic features: the handling of *sandhi* and the expansion of stems to their inflected forms.

Acknowledgements

I am grateful to the anonymous reviewers for their helpful corrections and suggestions.

References

Stefan Büttcher and Charles L.A. Clarke. 2005. Indexing time vs. query time trade-offs in dynamic information retrieval systems. In *University of Waterloo Technical Report CS-2005-31*, Waterloo, Canada.

Dominic Goodall and Marion Rastelli. 2013. *Tāntrikābhidhānakośa III: Dictionnaire des termes techniques de la littérature hindoue tantrique / A Dictionary of Technical Terms from Hindu Tantric Literature / Wörterbuch zur Terminologie hinduistischer Tantren.* Sitzungsberichte, Österreichische Akademie der Wissenschaften, Philosophisch-Historische Klasse; Beiträge zur Kultur- und Geistegeschichte Asiens. Verlag der Österreichische Akademie der Wissenschaften, Wien.

Oliver Hellwig. 2009. `SanskritTagger`. In Gérard Huet, Amba Kulkarni, and Peter Scharf, editors, *Sanskrit Computational Linguistics: First and Second International Symposia* (Institut national de recherche en informatique et en automatique, Rocquencourt, France, Oct. 29–31, 2007 and Brown University, Providence, RI, USA, May 15–17, 2008), number 5402 in Lecture Notes in Artificial Intelligence, pages 266–277, Berlin and Heidelberg. Springer.

Oliver Hellwig. 2010. Performance of a lexical and POS tagger for Sanskrit. In Girish Nath Jha, editor, *Sanskrit Computational Linguistics: 4th Symposium* (Jawaharlal Nehru University, New Delhi, India, Dec. 10–12, 2010), number 6465 in Lecture Notes in Artificial Intelligence, pages 162–172, Berlin and Heidelberg. Springer.

Gérard Huet and Idir Lankri. 2018. Preliminary design of a Sanskrit corpus manager. In Gérard Huet and Amba Kulkarni, editors, *Computational Sanskrit & Digital Humanities: Selected Papers Presented at the 17th World Sanskrit Conference* (University of British Columbia, Vancouver, July 9–13, 2018), pages 259–276, New Delhi. D K Publishers Distributors Pvt. Ltd.

Gérard Huet. 2003. Lexicon-directed segmentation and tagging of Sanskrit. XII$^{\text{th}}$ World Sanskrit Conference, Helsinki, Finland, Aug. 2003. URL: `http://gallium.inria.fr/~huet/PUBLIC/wsc.pdf` (accessed 2018/10/10).

Gérard Huet. 2005. A functional toolkit for morphological and phonological processing, application to a Sanskrit tagger. *Journal of Functional Programming*, 15(4):573–614.

K.A. Subramania Iyer. 1966. *Vākyapadīya of Bhartṛhari with the Commentaries Vṛtti and Paddhati of Vṛṣabhadeva: Kāṇḍa I.* Number 32 in Deccan College Monograph Series. Deccan College, Postgraduate and Research Institute, Poona.

Alistair Moffat and Justin Zobel. 1996. Self-indexing inverted files for fast text retrieval. *Transactions on Information Systems*, 14(4):349–379.

Sheldon Pollock. 2018. "Indian philology": Edition, interpretation, and difference. In Silvia D'Intino and Sheldon Pollock, editors, *L'espace du sens: Approches de la philologie indienne / The Space of Meaning: Approaches to Indian Philology*, number 84 in Publications de l'Institut de civilisation indienne, pages 3–45, Paris. Collège de France.

Martin F. Porter. 1980. An algorithm for suffix stripping. *Program*, 14(3):130–137.

William Pugh. 1990. Skip lists: A probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676.

Peter M. Scharf and Malcolm D. Hyman. 2012. *Linguistic Issues in Encoding Sanskrit.* Motilal Banarsidass, Delhi.

Srigowri and Karunakar. 2013. Gaveṣikā: A search engine for Sanskrit. In Malhar Kulkarni and Chaitali Dangarikar, editors, *Recent Researches in Sanskrit Computational Linguistics: Fifth International Symposium* (IIT Mumbai, India, Jan. 4–6, 2013), Delhi. DK Printworld.

Trevor Strohman, Donald Metzler, Howard Turtle, and W. Bruce Croft. 2005. Indri: A language model-based search engine for complex queries. In *Proceedings of the International Conference on Intelligent Analysis, Technical Report.*

Jian Zhang, Jian-Yun Nie, Jianfeng Gao, and Zhou Ming. 2000. On the use of words and *n*-grams for Chinese information retrieval. In *IRAL '00: Proceedings of the Fifth International Workshop on Information Retrieval with Asian Languages*, pages 141–148, Hong Kong, China. ACM.