INDEXING OF LARGE N-GRAM COLLECTION

Patrik Zajec, Marko Grobelnik
Artificial Intelligence Laboratory,
Jožef Stefan Institute
Jamova cesta 39, 1000 Ljubljana, Slovenia
e-mail: patrik.zajec@ijs.si, marko.grobelnik@ijs.si

ABSTRACT

This paper presents an efficient indexing technique suitable for indexing large n-gram collections with an emphasis on full wildcard query support and speed efficiency. Further we used this technique in building the n-gram search engine, on top of Google's Web 1T 5-gram collection, whose advantages are interactive querying and fast result retrieval with tradeoff on higher memory consumption.

1 INTRODUCTION

In the statistic NLP part of a research project we are using Web 1T 5-gram Version 1 corpus contributed by Google Inc. [1]. The corpus contains English word n-grams with length ranging from unigrams (single word) to five-grams. It contains around 4 billion n-grams and takes around 90 GB of memory.

Due to the corpus size we need an efficient indexing method which enables interactive full wildcard querying support on the data. Query time is our main priority that is why the index is kept in primary memory to eliminate disk IO operations. We have built our own system which is running on a server machine with 512 GB of RAM and has a constant memory usage of 280 GB.

2 RELATED WORK

There is a variety of different approaches and systems for storing and querying large n-gram collections. One class of approaches deals with an efficient representation of data in terms of memory usage while it usually offers only basic membership query.

There are approaches that seek for a tradeoff between memory footprint, speed efficiency and query expressiveness. Usually these are the combinations of fast in memory indexes of data stored on disk, where frequent disk access reduces the performance. Intuitive approach to achieve rich querying capabilities is to store the collection in a relational database and then index the data in desired manner [2], but here speed and memory efficiency suffers.

The work presented in [3] is an approach with in memory B+ tree index to enable wildcard query support but with limitation that wildcards in the query are continuous. Paper [4] describes an offline query pre-processing approach where queries are stored in nested hash table structure and answers are calculated with a single pass through the corpus. [5] summarizes various approaches and presents an efficient storing architecture with n-grams stored in an enhanced prefix tree. It also supports wildcard queries but not as efficiently as our system does.
Main advantages of our system are its **interactivity** (queries are answered online), **full wildcard query support** and very **fast query result retrieval** as searching for n-grams matching a given pattern is fast and straight-forward procedure.

3 INDEXING APPROACH

A. DEFINITION

N-gram is a contiguous sequence of n items. Define the number of items in the sequence as a degree of n-gram – for n-gram Z we mark its degree as $deg(Z)$. As in indexed corpus n-grams are sequences of words, let $Z[i]$ represent the word that appears at position i in the n-gram Z.

Wildcard query is represented with wildcard query pattern which is defined as n-gram Qp of a constant degree D. In our case D equals 5 which is a maximum degree of n-gram from the corpus. For each $i = 1 \ldots D$, $Qp[i]$ is either a word or a wildcard represented as '*' character. Furthermore we define two functions: $isWord(S[i])$ which is **true** if there is a word at position i in given n-gram S and $isWildcard(S[i])$ which is **true** if there is a wildcard at position i in S. We say that n-gram X is a member of the **result set** R of a given wildcard query (n-gram X is said to be a **match**) if for each $i = 1 \ldots D$ the following holds: $isWord(Qp[i])$ and $equals(Qp[i],X[i])$ or $isWildcard(Qp[i])$ – example is shown on figure 1. If $deg(X) < D$ then every such i where $deg(X) < i$ is ignored when determining if X is a match or not.

We define a set of supported wildcard queries Q as a set which contains only wildcard queries our system currently supports – we will need this definition in the next paragraphs.

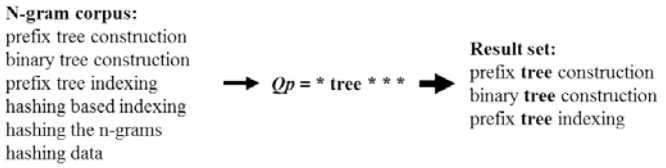

Figure 1: Result set of wildcard query where *Qp=* tree * * ** on the example n-gram corpus.

B. METHOD WITH RESTRICTION ON QUERIES

The naïve way of building the result set for an arbitrary wildcard query on the corpus would be to iterate through the whole corpus and for each n-gram check whether it is a match. But as the corpus is quite large we need a better approach.

Let's try to solve a simple problem where our set *Q* contains just queries where *Qp* has no wildcards. Obviously *R* will contain at most one n-gram as this is in fact a membership query. To speed up answering to that type of queries let's store the n-grams in a **prefix tree** [6] as shown in the figure 2. Answering a query is then just a matter of traversing this prefix tree from top to bottom.

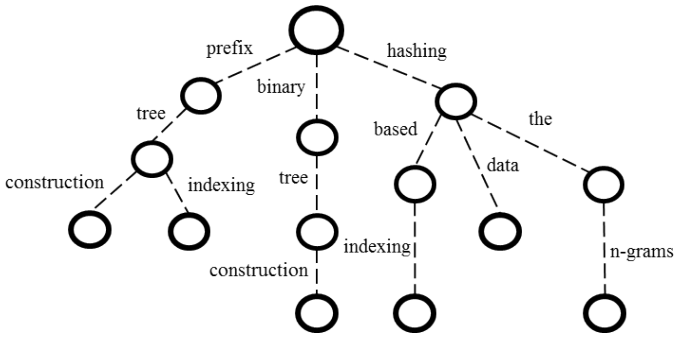

Figure 2: Example corpus stored in prefix tree.

Let's extend our set *Q* with those queries where *Qp* can contain wildcards but with some restrictions. We will split the set of positions from *Qp* in two sets according to a property whether there is a word or a wildcard on a specified position. We call these sets **word positions** *W(Qp)* and **wildcard positions** *C(Qp)*. *W(Qp)* is defined as $\{i;\ 1 \le i \le D$ *and isWord(Qp[i]) }*, *C(Qp)* is defined as $\{i;\ 1 \le i \le D$ *and isWildcard(Qp[i]) }*. Define a **simple query** as query where in its pattern *Qp* all the word positions appear earlier than any of the wildcard ones. In other words $max(W(Qp)) < min(C(Qp))$.

The fact that all the n-grams are stored in prefix tree allows us to use simple tree traversal as we did before until we hit a wildcard in pattern *Qp*. We can realize that all of the n-grams present in the sub-tree rooted at node where we are currently located are exclusively contained in the result set *R*. We can therefore easily generalize searching procedure and extend our set *Q*. But *Q* still contains just a small part of all possible wildcard query patterns.

C. EXTENDING THE SET OF QUERY PATTERNS

Define a **permutation of a n-gram** as a permutation of words in its sequence – we will write permutations in **one-line notation**. Let's permute all the n-grams from the corpus according to some specified permutation π. We then execute an arbitrary simple query on this modified corpus and get a set of results *R'*. If we permute members of *R'* back to their original form we find that *R'* contains same n-grams as if we would execute a query with pattern *Qp'* over original corpus (corpus where n-grams are not permuted). We can get the pattern *Qp'* just by permuting pattern *Qp* according to permutation $inv(\pi)$ where $inv(\pi)$ is inverse of permutation π. Of course pattern *Qp'* does not necessarily correspond to a query from the simple query set *Q*. That is why we cannot execute it in the same efficient way as we can execute simple queries. But we get an important intuition.

If we start with non-simple query pattern *Qc* and transform it to simple query pattern *Qs* with some permutation σ and then make a query on a collection in which all of the n-grams from the corpus are permuted by σ, then querying with pattern *Qs* is possible in the simple tree traversal way as presented before; the results in set *R* are same as they would be if we made a query with pattern *Qc* on the original collection (the results from *R* just have to be permuted with $inv(\sigma)$ to get the original version of resulting n-grams). So we have a procedure to efficiently find the result set for all kinds of wildcard queries. Each query just has to be transformed to simple query by transforming its pattern *Qp* to simple one with some permutation σ and all the n-grams from the corpus have to be permuted by the same permutation σ. On matching stage we can act as if we are dealing with a simple query and efficiently find the set *R* whose members have to be permuted with $inv(\sigma)$ to get the resulting n-grams back in their original form.

D. TRANSFORMING NON-SIMPLE PATTERN TO SIMPLE ONE

Finding σ to transform *Qc* to *Qs* is an easy task. What makes the query pattern non-simple is the fact that there exists a wildcard position which is later followed by a word position. Permutation σ just has to map all the words in front of the wildcards. The catch is that if we want to use an efficient matching method also all the n-grams have to be permuted by σ and stored in an **auxiliary collection**. That is why we have to find the minimum set of permutations *M* so that for each possible wildcard query pattern there exists a permutation from *M* that transforms it to simple pattern. Size of *M* has to be as small as possible because each of permutations form *M* will have a corresponding auxiliary collection of all the n-grams from the corpus permuted according to it. The fact that actual order of words in permuted pattern is irrelevant as long as they are all in front of the wildcards helps us to significantly decrease the upper bound on the number of permutations as there is usually more than one candidate permutation σ that satisfies this transformation property. More formally we have to find such set of permutations M, that for each possible query pattern *Qp* there exists such a permutation σ from *M* that

im(σ(W(Qp))) = {1,…,|W(Qp)|} and *im(σ(C(Qp))) = {|W(Qp)| + 1,…,D}*. It's quite obvious that *|M|* can be bounded by the number of different sets *W(Qp)*. Since size of the set *W(Qp)* is limited with maximum degree of a pattern *Qp* (in our case *D = 5*) we only have 2^5 different possible sets. The upper bound on *|M|* is therefore 32 but can be improved even further.

E. MINIMUM SET OF PERMUTATIONS

Let *M(d)* represent a subset of *M* containing minimum number of permutations required to transform any wildcard query pattern *Qp* where *|W(Qp)| = d* to simple one. We will build the set M inductively on *d*.

For *d = 1* clearly *|M(d)| = D* as for each possible word position a permutation that maps it to 1 is needed.When constructing *M(d)* where *d > 1* we can find for each possible set *W(Qp)* of size *d* the permutation from *M(d – 1)* that maps *d – 1* elements of its elements to *{1,…,d – 1}* and then extend this permutation to map the remaining element to *d*.

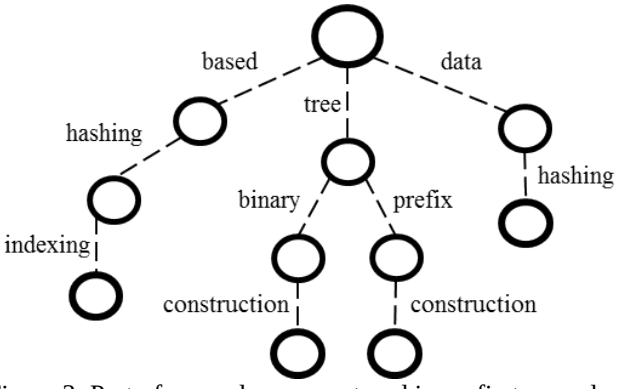

Figure 3: Part of example corpus stored in prefix tree and permuted by *σ = (2 1 3)*.

We can notice that the size of *M(d)* is the same as the number of different possible sets *W(Qp)* of size *d* which equals to *C(D,d)* (where *C(n,k)* represents a combination). As *C(n,k)* is maximal when k equals *n / 2* (integer division) we can set the lower bound on size of *|M|* to *C(D,D / 2)* as there are *C(D,D / 2)* permutations in set *M(D / 2)*. But can we always construct *M* so that its size actually matches the lower bound? It turns out we can, as long as we reuse (and extend) all the permutations from *M(d – 1)* when constructing *M(d)* – where *|M(d – 1)| ≤ |M(d)|*. As all the permutations from *M(d – 1)* are also included (in the extended form) in *M(d)* there always exists a permutation from *M(d)* which transform a pattern with *|W(Qp)| = d - 1* to simple one. It implies that permutations from the set *M(D / 2)* are enough to correctly transform all possible *W(Qp)* where *|W(Qp)| ≤ D / 2*. We can combine the permutations from *M(D / 2)* with those from *M(d)* where *D / 2 < d* to construct the minimum set *M* with *C(D,D / 2)* elements.
For indexing our corpus the minimum set of permutations contains exactly 10 different permutations (in appendix), each of which has its corresponding auxiliary collection with n-grams permuted according to it. Therefore our index is 10 times as large as the corpus.

4 INDEX CONSTRUCTION

N-grams from the corpus are stored in 10 different collections. Each collection has a corresponding permutation according to which n-grams stored in it are permuted. Collections are implemented as prefix trees so that finding the result set can be done in a simple and efficient way as described before. Another advantage of prefix tree is that it can be fairly efficiently compressed once it is built. We are building collections sequentially; building procedure is divided in two stages. In first stage n-grams from the corpus are permuted and sequentially inserted in prefix tree which is stored in primary memory. When built, prefix tree takes about 240 GB of memory as it is not efficiently represented for a sake of faster insertion. On the second stage tree is compressed and in the end it takes only about 30 GB of memory. As compression we mean that nodes, edges and words are represented in a more succinct way which slightly reduces speed efficiency when traversing the tree. At the end the whole index takes 280 GB and is loaded in primary memory to achieve fastest result retrieval. Index construction took around 24 hours on a server machine with 512 GB of RAM.

5 EFFICIENCY

When the query pattern *Qp* is known it is transformed to simple query pattern *Qs* by the right permutation. Then the collection (prefix tree) which corresponds to this permutation is identified. Once we have the right prefix tree finding the result set is just a matter of simple tree traversal. Results can be served all at once or by iteratively calling *getNextResult()* function. While the first match to given query pattern is found instantaneously due to straight-forward matching procedure, the system is capable to retrieve "just" 10^5 matches per second. The slowdown of its performance lies in succinct representation which results in slower tree traversal due to the complex structure of data.

6 CONCLUSION

We have described a novel indexing schema whose main advantages are full wildcard query support, interactive querying and speed efficiency. Even though it is currently a static indexing approach it can be extended to a dynamic version quite easily. Its usage is not limited just on word n-grams.

REFERENCES

[1] Thorsten Brants and Alex Franz. 2006. Web 1T 5-gram corpus version 1. Technical report, Google Research.

[2] S. Evert. 2010. Google web 1t 5-grams made easy (but not for the computer). In Proceedings of the NAACL HLT 2010 Sixth Web as Corpus Workshop, WAC-6 '10, pages 32–40.

[3] Ceylan, H., and Mihalcea, R. 2011. An efficient indexer for large n-gram corpora. In Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies (ACL-HLT 2011), System Demonstrations, Portland, OR, USA, pp.

103–8. Stroudsburg, PA: Association for Computational Linguistics.
[4] T. Hawker, M. Gardiner, and A. Bennetts. 2007. Practical queries of a massive n-gram database. In Proceedings of the Australasian Language Technology Workshop 2007, pages 40–48, Melbourne, Australia.
[5] Michael Flor. 2013. A fast and flexible architecture for very large word n-gram datasets. Natural Language Engineering, 19(1), 61-93.
[6] Fredkin, E. 1960. Trie memory. Communications of the ACM 3(9): 490–9.

APPENDIX

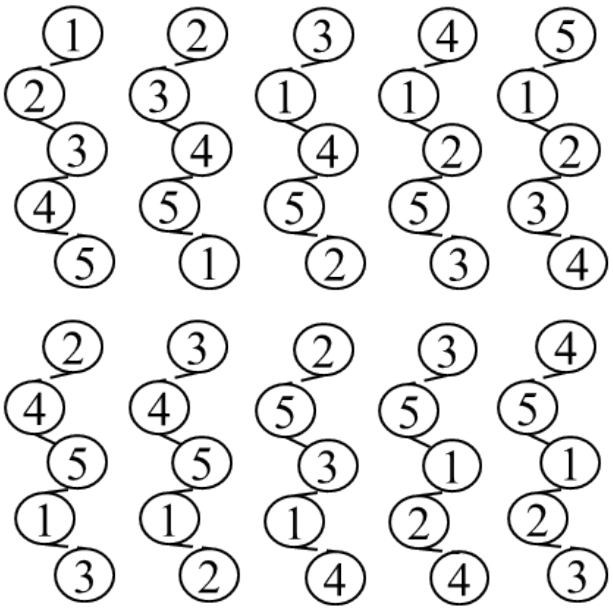

Figure 4: Minimum set of permutations for indexing n-grams with degree up to 5 used in our index. Permutations are written in one-line notation.