An Efficient Type-agnostic Approach for Finding Sub-sequences in Data

Bertil Chapuis
Université de Lausanne
bertil.chapuis@unil.ch

Benoît Garbinato
Université de Lausanne
benoît.garbinato@unil.ch

Periklis Andritsos
University of Toronto
periklis.andritsos@utoronto.ca

***Abstract*—In this paper, we present an efficient type-agnostic approach for finding sub-sequences in data, such as text documents or GPS trajectories. Our approach relies on data deduplication for creating an inverted index. In contrast with existing data deduplication techniques that chunk raw sequences of characters arbitrarily, our approach preserves the semantics of the original sequence via the notion of token and can be used to index normalized data. When compared to indexing methods that preserve the semantics and operate on normalized data, our method increases the relevance of the inverted index, reduces its size and improves its performances. Since data normalization is generally not used beyond the scope of textual data, we introduce a framework that helps identify the extent to which data should be normalized regardless of its type. On this basis, we demonstrate with a dataset made of GPS trajectories that our method can be used agnostically: it can be used to index and query data of a completely different type. Finally, we show that the resulting spatial index is characterized by a better discrimination than classical spatial indexing approaches.**

I. Introduction

We are witnessing an unprecedented rise in the demand for data processing. A large part of data consists of ordered sequences that can be processed in a batch or in a streaming fashion. A common need when processing data is to find common sub-sequences between a query and sequences of items. In this paper, we introduce an efficient type-agnostic approach for finding sub-sequences in sequences of data. We refer to a token as a semantic piece of information that should be preserved while handling sequences of data. Tokens can be words in the context of textual data but may also refer to groups of attributes such as the coordinates that compose GPS trajectories. When searching for sub-sequences in data, a common approach consists in segmenting the sequences that compose the dataset. Two very practical segmenting approaches are generally used by indexing solutions for finding sub-sequences in data.

- **N-grams**. The most common way of finding sub-sequences (mainly for textual data) is called *n-grams* [15]. By computing all the possible overlapping sub-sequences of size n in a sequence of tokens, n-grams give exhaustive results when looking for similarities. This approach comes with a high cost in terms of storage requirements, which tends to increase with the overlapping factor n. When used in the context of spatial indexing, the number of possible n-gram combinations rapidly explodes and disqualifies this technique.
- **Hash-based**. Another approach for producing segments, called Hash-based (HB), identifies non-overlapping contiguous sub-sequences of tokens, called *chunks* [6]. Chunks can be identified by comparing the hash sums of the successive tokens to some constant. If the hash sum is equal to the constant, then a chunk boundary is set, otherwise the next token is hashed and the process is repeated until a chunk boundary is found. This approach requires less storage and computing power than n-grams, but gives non-exhaustive results. In addition, since the number of possible hash sums is limited to the number of possible tokens, it tends to produce a lot of small chunks that lead to the detection of short and irrelevant sub-sequences.

When studying these two segmentation approaches, we observe that there exists a clear tradeoff between the n-gram approach (that requires a lot of resources) and the hash-based approach (that trades effectiveness for efficiency). Some data deduplication techniques, such as content-defined chunking (CDC), might be considered as optimized variants of the Hash-based approach. These techniques greatly improve the aforementioned tradeoff, which is of great importance in the context of data storage. However, data deduplication operates on raw sequences of bytes and do not preserve the semantic of tokens.

Contributions: In this paper, we introduce token-based chunking (TB), an approach for segmenting sequences of tokens regardless of their type. The chunking mechanism used by this approach is inspired by CDC but preserves the semantic of tokens and overcomes the issues associated to HB. When compared to other segmenting techniques, TB greatly improves the effectiveness of the index and reduces its size. Normalization is relatively intuitive in the context of textual data. However, the extent to which non-textual data such as GPS trajectories should be normalized is harder to determine. Consequently, we propose a normalization framework that can be used to evaluate and determine the best normalization settings. When used for indexing trajectories, we demonstrate that TB overcomes the discrimination problem that characterizes traditional spatial indexing methods.

The following sections aim at showing that TB can be used as a general purpose technique for building indexes on various kinds of sequential data. In Section II, we give a birds-eye view of our approach. We provide a detailed

description of TB in Section IV. In in section V, we describe our normalization framework and show to which extent data should be normalized regardless of its type. In Section VI, we present a detailed evaluation of TB with textual data. In Section VII, we show how TB can be used to create an index for GPS trajectories. Finally, in Section VIII, we highlight previous works from the data storage and the data deduplication fields.

II. Overall Approach

An inverted index is usually composed of a dictionary of terms, each of which points to a posting list that contains document identifiers. Boolean queries are then used to retrieve documents that contain a set of query terms. Alternatively, phrase queries are used to take the position of the term in the document into account. When building an index aimed at searching for sub-sequences in documents, using words as terms of the dictionary usually gives poor results both in terms of efficiency and effectiveness. Therefore, segmentation techniques such as n-gram, CDC or HB are used to create the terms of the inverted index. The overall approach for creating the index could be summarized in the phases listed hereafter.

1) **Extraction**. This phase depends on the type of the data sequence and extracts meaningful data only. For example, in the case of XML data, the unnecessary tags and attributes are usually removed.
2) **Tokenization**. This phase splits the extracted data into small pieces called tokens. In the case of textual data, punctuation and white spaces can be used to create a sequence of tokens that corresponds to words.
3) **Normalization**. Token normalization comprises all the operations that can be performed on tokens (such as stop-word removal, equivalence classes, case-folding, true-casting, stemming or lemmatization).
4) **Segmentation**. When looking for sub-sequences, segmentation techniques, such as n-grams or HB, are used to compute overlapping or non-overlapping sequences of tokens which are then used as dictionary terms.
5) **Indexing**. This phase creates the inverted index by adding the segments and the document identifiers to the dictionary and the posting lists.

CDC operates directly on sequences of bytes or characters and Phases 2 and 3 are usually skipped [11], [3]. In contrast, TB operates on token and benefits from the tokenization and normalization phases.

III. Background and Model

The approach proposed in this paper builds on results from three domains, namely *tokenization*, *normalization* and *data deduplication*. To describe how these techniques operate, we first introduce the notion of an alphabet, consisting of a finite set of symbols. Formally, we define Σ_i, an alphabet containing bit sequences of fixed length i, e.g., $\Sigma_1 = \{0,1\}$ and $\Sigma_2 = \{00,01,10,11\}$; so we have $|\Sigma_i| = 2^i$ to be the size of the alphabet Σ_i. In the following we assume that the considered alphabet is Σ_8, the set of all possible bytes that can be used to represent ASCII characters. We also define a *byte word* as a variable size sequence of symbols from Σ_8. Using Kleene closure, we have Σ_8^*, the set of possible byte words. Assuming Σ_8^n is the set of all byte words of exactly size n, we have:

$$\Sigma_8^* = \bigcup_{n \in \mathbb{N}} \Sigma_8^n$$

Hereafter, we introduce the concepts and terminology that are useful to understand our approach.

Tokenization: In the context of texutal data, tokenization splits sequences of characters into words of some language. In other words, tokenization is a process that takes a sequence of bytes as input and produces a sequence of meaningful tokens as output. Tokens belong to the infinite alphabet $T = \Sigma_8^*$. On this basis, we can define tokenization as a function $f_c : \Sigma_8^* \rightarrow T^*$, where T^* is a Kleen closure on T that contains all the possible tokens. For example, assuming the considered language is English and bytes are interpreted as ASCII characters, byte sequence $\langle The\ quick\ brown\ fox \rangle$ is tokenized as $\langle\ \langle The \rangle, \langle quick \rangle, \langle brown \rangle, \langle fox \rangle\ \rangle$, which we simply write $\langle The, quick, brown, fox \rangle$ in the rest of the paper.

Normalization: In many cases, tokens can be different but convey similar semantics. That is the case, for example, when a word starts with a capital letter at the beginning of a sentence. Normalization aims at removing such superficial differences, so that a match can occur on semantically similar tokens. We define it as a function $f_n : T^* \rightarrow T_n^*$, where T_n is a set of tokens that depends on the type of normalization being applied. For example, a case-folding normalization function, which simply replaces capital letters by lower case letters, would produce tokens in a subset of T. On the other hand, a stemming normalization function, which defines heuristics for making similar words converge toward the same tokens, would produce tokens that do not necessarily belong to the english language.

Data Deduplication: Data deduplication can be seen as a particular approach to data compression that operates on large corpora of files, rather than independent files. At the heart of data deduplication techniques, we find chunking algorithms working on multiple files, with the goal to find common data chunks. Formally, a chunking algorithm takes a sequence of data as input, in the form of one (long) byte word $w \in \Sigma_8^*$, and returns a sequence of byte words $r_w = \langle w_1, w_2, \cdots, w_k \rangle$, called *the recipe* of w, such that $\forall w_i \in r_w : w_i \in \Sigma_8^*$ and $w = w_1 \| w_2 \| \cdots \| w_m$. The byte words of recipe r_w are precisely what we call chunks. For example, two possible recipes for byte word $\langle 25763537 \rangle$ are $\langle \langle 25 \rangle, \langle 76 \rangle, \langle 35 \rangle, \langle 37 \rangle \rangle$ and $\langle \langle 257 \rangle, \langle 6353 \rangle, \langle 7 \rangle \rangle$.

Another way to understand recipe r_w consists in introducing new alphabet $C = \Sigma_8^*$, which contains all the possible byte words (the symbols of that new alphabet), and to see r_w as a word built using symbols of C. Note that C is an infinite alphabet, contrary to Σ_8. Using Kleene closure again, we have C^*, the set of all possible recipes. Assuming C^n is the set of all recipes of exactly size n, we have:

$$C^* = \bigcup_{n \in \mathbb{N}} C^n$$

On this basis, we define chunking as a function $f_c : \Sigma_8^* \to C^*$ such that $f_c(w) = r_w$, with w and r_w satisfying the constraints mentioned earlier. Note that in practice, each chunk is stored only once, while being referenced in one recipe or (hopefully) more (hence the deduplication). That is, the recipe is a type of meta-data containing only references to actual chunks from which the original data sequence can be recreated.

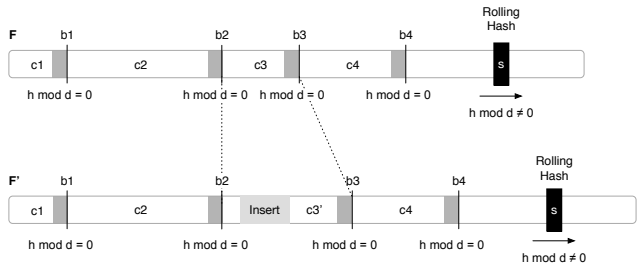

Fig. 1: Content-defined chunking (CDC)

Fixed-size chunks are common but comes with a major drawback: when bytes are added at the beginning of a sequence, all the following chunk boundaries are shifted. CDC solves this issue by detecting addition-resistant chunk boundaries [9]. Figure 1 depicts how a simple CDC algorithm operates. A rolling hash function [21] of size s, depicted here by a black box, slides over a file F one byte after the other. After each move, the rolling hash function computes a hash sum h for the bytes that are located in the window. Chunk boundaries are detected by checking the condition $h \; mod \; d = 0$ on the hash sums produced by the sliding window, where d is a user defined divisor. The divisor d is typically used to manage the average chunk size. Since the boundaries are based on the content, they resist to insertions. If we assume two successive versions of a file, such that F' is a copy of F with an insertion in the middle, the algorithm will be able to find the same boundaries again and to isolate the chunk in which the insertion occurred.

IV. Token-based chunking

As stated in Section I, CDC breaks the semantics of tokens. This problem comes from the fact that CDC relies on rolling hash functions to identify breakpoints. In order to compute hash sums on sequences of bytes, rolling hash functions usually map the symbols of the alphabets Σ_8 to a set of precomputed random irreducible polynomials. Since the alphabet Σ_8 is limited in size, these random polynomials are necessary in order to uniformly and randomly distribute the hash sums produced by the rolling hash function over the hash sum space [21].

Precomputing a set of irreducible random polynomials works well when the size of the alphabet is small and known in advance, such as in the case of Σ_8. In our case, however, we deal with an alphabet made of tokens T, which is potentially infinite, i.e., $|T| = \infty$. Therefore, it is not possible to precompute one irreducible random polynomial per token. Furthermore, in contrast to CDC, we handle sequences of tokens and produce recipes that contain sequences of tokens. Another way to understand this issue consists in introducing a new alphabet $C_T = T^*$, which contains all the possible sequences of tokens. Again, a Kleene closure can be used to define C_T^*, the set of all possible recipes over the alphabet T, such that:

$$C_T^* = \bigcup_{n \in \mathbb{N}} C_T^n$$

Based on this definitions, we can formally define TB as the function $f_{tb} : T^* \to C_T^*$. The algorithm we introduce does not rely on a precomputed set of irreducible random polynomials in order to detect chunk boundaries and satisfy this definition by producing chunks that consist of sequences of tokens.

The algorithm is decomposed into two parts. The first part is given in Algorithm 1 and is responsible for detecting chunk boundaries. The main configuration parameters are a minimum (*min*) and a maximum (*max*) chunk size, a divisor (d) and a window size (s). As the tokens are consumed, a rolling hash function produces hash sums and new chunks are produced when these hash sums meet a certain criterion, represented here by the expression $h \bmod d = 0$. TB differs from CDC in the sense that it consumes tokens instead of bytes and produces sequences of tokens instead of sequences of bytes.

Algorithm 1 Token-based chunking algorithm

```
initialize(min, max, d, s):
    chunk ← {∅}
    hash ← a rolling hash function of size s
read(token):
    chunk ← chunk :: token
    h ← hash.slide(token)
    if |chunk| ≥ min and
       (h mod d = 0 or |chunk| ≥ max) then
      write(chunk)
      chunk ← {∅}
    end if
```

The second part of the algorithm is the rolling hash function given in Algorithm 2. Since it is not possible to precompute random irreducible polynomials for a vocabulary of an unknown size, we assume that the hash sums produced by hashing the tokens are random enough to replace the precomputed irreducible random polynomials. To do so, we choose to use a fast non-cryptographic hash function called Murmur Hash, which produces 32 bit integers. A fixed size sliding window of hash sums is maintained over the sequence of tokens to represent the incoming and outgoing tokens. Our evaluation setup shows that the sums produced by this rolling hash function reaches the desired properties and produces balanced hash sums.

The segments generated by this algorithm can be used as dictionary terms and overcome the following problems. First,

Algorithm 2 Token-based rolling hash function

initialize(s):
  $a \leftarrow 31$
  $b \leftarrow a^s$
  $hash \leftarrow$ a murmur hash function
  $window \leftarrow$ an array of size s filled with 0
  $position \leftarrow 0$
  $h \leftarrow 0$
slide(token):
  $in \leftarrow hash.digest(token)$
  $out \leftarrow window[position]$
  $window[position] \leftarrow in$
  $position \leftarrow (position + 1) \mod s$
  $h \leftarrow a * h + in - b * out$
  return h

a common pitfall of HB lies in the fact that the hash sum of a single token is used to identify chunk boundaries. Therefore, HB could identify boundaries for very frequent tokens such as *the*. In contrast, our algorithm is not sensitive to token frequency since it computes hash sums over a sliding window. Second, the main problem associated to CDC relies in the fact that it can break the semantics of the sequence of tokens. Unlike CDC, our method preserves the semantics of tokens. In addition, the introduction of thresholds tends to mitigate the risk of extracting small or large segments that would impact precision and recall negatively. As we will show later, our algorithm is characterized by a much improved effectiveness and efficiency than its counterparts.

V. Normalization framework

When dealing with textual data, the extent to which normalization is performed is often based on simple intuitions. However, these intuitions are not valid when dealing with different types of data, such as GPS trajectories. In this section, we introduce a normalization framework for evaluating the extent to which normalization should be performed regardless of the data type. By observing the evolution of precision and recall, we show with a very simple textual dataset when one should start and stop normalizing data.

In information retrieval, precision and recall are often used to measure the effectiveness of an index, so we start by briefly reminding these metrics. Precision corresponds to the fraction of retrieved items that are relevant. In other words, $precision = tp/(tp+fp)$, with tp (true positive) the number of relevant items retrieved and fp (false positive) the number of irrelevant items retrieved. Recall corresponds to the fraction of relevant items that are retrieved. More formally, $recall = tp/(tp+fn)$, with tp (true positive) the number of relevant items retrieved and fn (false negative) the number of relevant items that have not been retrieved.

Another important measure when looking for similarities is the Jaccard Similarity coefficient. In our context, this coefficient can be used to measure the similarity between two recipes, which corresponds to sets of chunks. Therefore, given two recipes $r_1, r_2 \in C_T^*$, their similarity coefficient is:

$$J(r_1, r_2) = \frac{|r_1 \cap r_2|}{|r_1 \cup r_2|}$$

We previously stated that a good normalization function should make highly similar sequences of tokens converge to more similar recipes. So, given two highly similar sequences $s_a, s_b \in T^*$, a good normalization function f_n should have the following property:

$$J(f_{tb}(f_n(s_a)), f_{tb}(f_n(s_b))) > J(f_{tb}(s_a), f_{tb}(s_b))$$

Unfortunately, this metric does not indicate when someone should stop making normalization more aggressive in the context of chunks. Given an index and a query that use the same normalization function, precision should remain stable and should be close to 1, since the probability of having two large identical sequences of words in unrelated documents is low. On the other hand, recall should increase since more relevant items will be found in the set of relevant items. Therefore, we can say that precision and recall can be used to identify the optimal extent of a normalization function and we demonstrate it in the following paragraphs.

In the case of textual data, a tokenizer is used to split a given text into a sequence of tokens that belong to the alphabet T. A sequence of tokens can be altered during the normalization phase to make highly similar tokens converge toward the same token. In order to illustrate more practically the effect of data normalization on a small dataset, we consider a set S of four sequences of tokens $s_1, s_2, s_3, s_4 \in \Sigma_8^*$. We assume that the deduplication of these sequences results in four recipes $r_1, r_2, r_3, r_4 \in C^*$ containing a single chunk after deduplication, such that:

$r_1 = f_{tb}(s_1)$ = $\{\langle A, fox, runs\rangle\}$
$r_2 = f_{tb}(s_2)$ = $\{\langle The, foxes, run\rangle\}$
$r_3 = f_{tb}(s_3)$ = $\{\langle Master, fox, is, running\rangle\}$
$r_4 = f_{tb}(s_4)$ = $\{\langle But, chickens, are, running, faster\rangle\}$

We now consider a query such that $r_q = f_{tb}(s_q) = \{\langle A, fox, runs\rangle\}$ and assume that both s_1 and s_2 are relevant answers. If s_q is used to retrieve relevant items in S by looking for exact duplicates, we expect a single result that corresponds to s_1. In that case, precision would be $1/(1+0) = 1$ and recall would be $1/(1+1) = 1/2$. We now consider a normalization function for textual data f_n^a which removes common stop-words, applies some case-folding rules and does some stemming on verbs and adjectives. The resulting recipes after normalization and deduplication may look like this:

$r_1^a = f_{tb}(f_n^a(s_1))$ = $\{\langle fox, run\rangle\}$
$r_2^a = f_{tb}(f_n^a(s_2))$ = $\{\langle fox, run\rangle\}$
$r_3^a = f_{tb}(f_n^a(s_3))$ = $\{\langle master, fox, run\rangle\}$
$r_4^a = f_{tb}(f_n^a(s_4))$ = $\{\langle chicken, run, fast\rangle\}$

By using the same normalization function on the query s_q, we end-up with the recipe $r_q^a = f_{tb}(f_n^a(s_q)) = \{\langle fox, run\rangle\}$ and

we expect two results which correspond to the two relevant sequences of token s_1 and s_2. In that case, precision would still be $2/(2+0) = 1$ but recall would improve at $2/(2+0) = 1$. We can now easily show that our assertion regarding Jaccard similarity is true since:

$$J(r_1^a, r_2^a) = \frac{|\{\langle fox, run\rangle\} \cap \{\langle fox, run\rangle\}|}{|\{\langle fox, run\rangle\} \cup \{\langle fox, run\rangle\}|} = \frac{1}{1} = 1$$

is greater than:

$$J(r_1, r_2) = \frac{|\{\langle A, fox, runs\rangle\} \cap \{\langle The, foxes, run\rangle\}|}{|\{\langle A, fox, runs\rangle\} \cup \{\langle The, foxes, run\rangle\}|} = \frac{0}{2} = 0$$

In order to determine when one should stop making a normalization function more aggressive, we consider a second normalization function f_n^b which only retains nouns and drops all the other words. The resulting recipes after normalization and deduplication may look like this:

$r_1^b = f_{tb}(f_n^b(s_1))$ = $\{\langle fox\rangle\}$
$r_2^b = f_{tb}(f_n^b(s_2))$ = $\{\langle fox\rangle\}$
$r_3^b = f_{tb}(f_n^b(s_3))$ = $\{\langle fox\rangle\}$
$r_4^b = f_{tb}(f_n^b(s_4))$ = $\{\langle chicken\rangle\}$

In that case, precision would drop to $2/2+1 = 2/3$ and recall would remain stable at $2/(2+0) = 1$. As a consequence, we get an idea of when it is sound or not to normalize in the context of TB. While recall improves, the data can be normalized more aggressively. On the contrary, a drop in precision indicates that normalization is too aggressive and the tokens do not capture what characterizes the sequence anymore.

VI. Evaluation

In this section, we compare TB with some other state of the art segmentation methods used for building inverted indexes and finding similarities in document corpora. Table I enumerates these methods and illustrates some possible segments produced by consuming the sequence of four tokens $\langle the, quick, brown, fox\rangle$. As illustrated, the methods based on n-gram compute all the possible contiguous overlapping sequences of tokens. HB produces non-overlapping chunks of variable size [6]. CDC finds chunk boundaries based on bytes and, hence, a token can be divided arbitrarily [11], [3].

Dataset: For each evaluated methods, we indexed a full dump of the English version of Wikipedia (50GB of raw textual data). Except for CDC, all the evaluated methods operates on normalized tokens. The normalization procedure was performed using Apache Lucene [1], a suite of tools that among others includes natural language processing methods. We performed the following normalization steps: tokens are normalized to lowercase characters; numeric tokens are filtered out; rare big tokens (larger than 1000 characters) are filtered out; english possessive forms are removed; common stop words are filtered out; stemming is performed.

Queries: In order to evaluate the effectiveness and efficiency of the different methods, we introduce two search scenarios. The first one consists in searching the dataset for *exact sentences*. In this case, the set of queries is built by randomly choosing wikipedia articles and, within each one, randomly picking two consecutive sentences. For each consecutive sentences, we then verify their uniqueness in the dataset and eliminate the one that occurs several times. Thus, this set contains 973 queries which are guaranteed to have at most one relevant result. The second scenario consists in searching the dataset for *cross references*, which typically corresponds to cases of plagiarism detection. We build this set of queries by combining the queries of the first set into 486 queries that are guaranteed to have at most two relevant results. This scenario is important because phrase queries, which accounts for the position of the terms in the document, cannot be used in the case of n-gram indexes, resulting in a loss of precision.

Method	Possible chunks
1-gram	$\{\langle the\rangle, \langle quick\rangle, \langle brown\rangle, \langle fox\rangle\}$
2-gram	$\{\langle the, quick\rangle, \langle quick, brown\rangle, \langle brown, fox\rangle\}$
3-gram	$\{\langle the, quick, brown\rangle, \langle quick, brown, fox\rangle\}$
4-gram	$\{\langle the, quick, brown, fox\rangle\}$
HB	$\{\langle the\rangle, \langle quick, brown, fox\rangle\}$
CDC	$\{\langle the\ quic\rangle, \langle k\ brown\ fox\rangle\}$
TB	$\{\langle the, quick\rangle, \langle brown, fox\rangle\}$

TABLE I: Indexing methods

Configuration: Table II lists all of the configuration parameters we used to conduct our experiments. Parameter "Unit" shows if the method mentioned in the corresponding column operates on tokens or on bytes. The "Window size", "Min size" and "Max size" parameters depend on the unit mentioned above and specify constraints on the size of the chunks produced. The "Divisor" and "Backup divisor" are used in order to detect chunk boundaries as explained in Section III. We defined the "Min size", "Max size" and "Divisor" parameters according to the recommendations described by Eshghi et al. [9] with the intent to reach a desired average chunk size. In order to compare the segmentation methods in a fair manner, we targeted configuration parameters that would generate segments of approximately 52 bytes in average.

Environment: We evaluated our indexes with Apache Lucene [1], which is a state of the art information retrieval library developed in java. This library provides the necessary components for configuring n-gram indexes. We implemented some custom components for building HB, CDC and TB indexes. Furthermore, the library contains a good set of benchmarking tools that we used as a basis to measure precision, recall, as well as performances. Regarding our hardware configuration, we ran all our benchmarks using a Dell Power Edge T110 II with an Intel Xeon CPU clocked at 3.50GHz and 16GB of RAM.

Chunk distribution: Figure 2 depicts the chunk size distribution for the three non-overlapping segmentation methods we evaluated. As illustrated here, HB produces a lot of small chunks, which results in a lot of irrelevant query results (Figure 5) and in a poor throughput (Figure 6). CDC solves this issue by introducing the *min* and *max* thresholds but the arbitrary

Method	1-gram	2-gram	3-gram	4-gram	HB	CDC	TB
Unit	Token	Token	Token	Token	Token	Byte	Token
Window size	-	-	-	-	-	23	3
Min size	-	-	-	-	-	41	4
Max size	-	-	-	-	-	248	27
Divisor	-	-	-	-	6	48	3
Backup Divisor	-	-	-	-	-	24	-
Normalization	Yes	Yes	Yes	Yes	Yes	No	Yes
Exact sentence (query type)	Phrase	Phrase	Phrase	Phrase	Boolean	Boolean	Boolean
Cross reference (query type)	Boolean	Boolean	Boolean	Boolean	Boolean	Boolean	Boolean

TABLE II: Configuration parameters

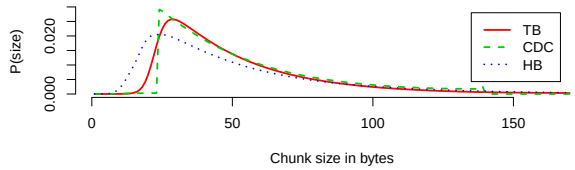

Fig. 2: Chunk distribution

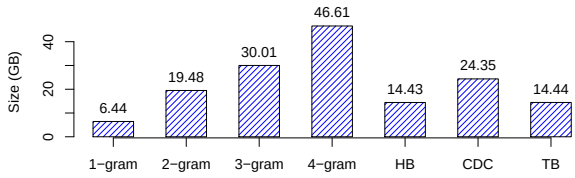

Fig. 3: Index size

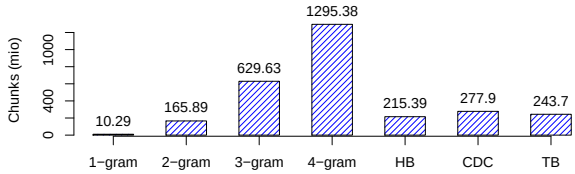

Fig. 4: Number of dictionary terms

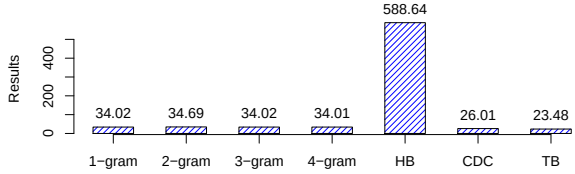

Fig. 5: Number of results for exact sentence queries

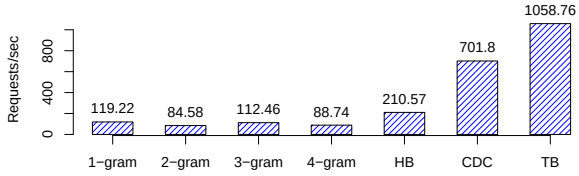

Fig. 6: Throughput for exact sentence queries

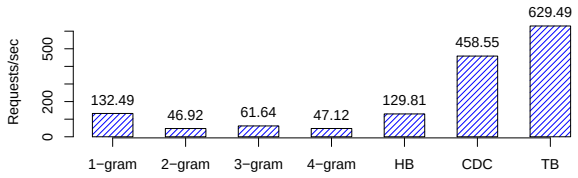

Fig. 7: Throughput for cross reference queries

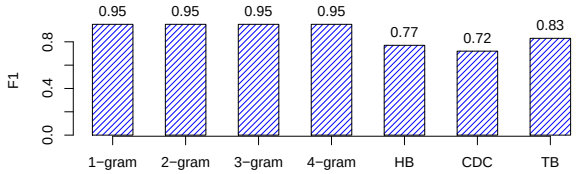

Fig. 8: F1 score for exact sentence queries

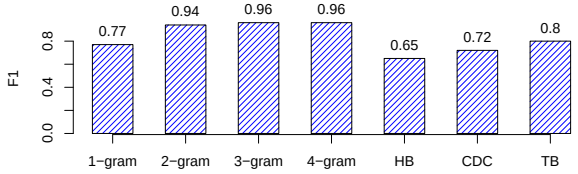

Fig. 9: F1 score for cross reference queries

chunk boundaries result in a loss of relevant query results. Therefore, TB reaches the best tradeoff in terms of efficiency and effectiveness by operating on tokens instead of bytes and avoiding arbitrary chunk boundaries.

Index size: Figure 3 and 4 depict the index size and the number of dictionary terms for the methods we compared. Here, we notice that, as the size of the n-gram increases, so does the index in terms of size and dictionary terms. From this perspective, HB and TB are more efficient than CDC, confirming the positive effect of avoiding arbitrary chunk boundaries.

Efficiency: Throughput is a measure of efficiency which corresponds to the number of requests per second that our setup can handle. Figures 6 and 7 depicts the throughput for the methods we compared and for the search scenarios we evaluated. Regarding this metric, we first notice that the throughput decreases as the size of the n-gram grows. This comes from the index size which grows fast as the size of the n-gram increases and from the fact that n-gram methods account for the position of the terms in the documents. In contrast, since the segments extracted by HB, CDC and TB are large and non-overlapping, it is not necessary to account for their positions in the documents. The poor throughput of HB is explained by the great number of irrelevant results (Figure 5), whereas the medium throughput of CDC is explained by the index size (Figure 3). Therefore, By addressing these two issues, TB is the most efficient method with a maximum throughput of 1058.76 requests per second.

Effectiveness: The F_1 score is a measure of effectiveness which corresponds to the harmonic mean of precision and recall. In Figure 8, we notice the cost of adopting HB, CDC or TB over n-grams in terms of effectiveness when searching for exact sentences. On one hand, n-gram methods are the best in terms of effectiveness but perform poorly in

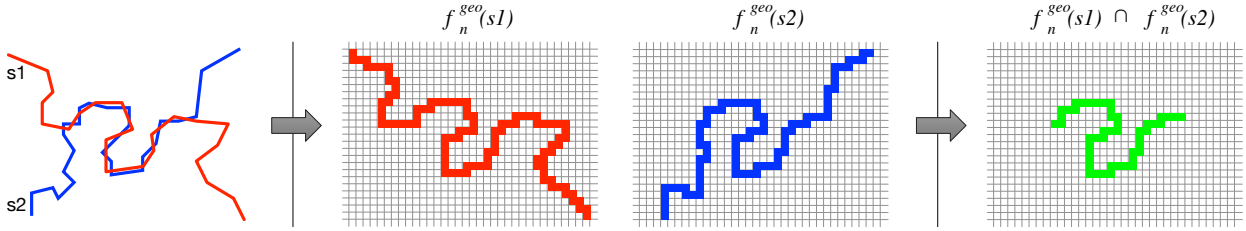

Fig. 10: Normalization and similarity detection in GPS trajectories

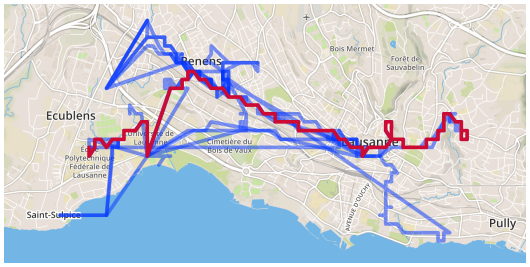

Fig. 11: A trajectory query (red) and some results that share sub-sequences (blue)

terms of throughput. On the other hand, when comparing TB with n-gram methods, we notice a loss of approximately 10% percent in terms of effectiveness (F_1) for a gain of 800% in term of efficiency (throughput). As highlighted here, when compared to HB and CDC, TB clearly optimizes this tradeoff. Figure 7 depicts the F_1 score for the second search scenario which consists in searching for cross references. As mentioned earlier, it is not possible to use phrase queries in this context and we notice a drop in the F_1 score for the 1-gram method. Consequently, since the other n-gram methods use more storage than TB and have a much lower throughput, the choice regarding this use case is not a matter of tradeoff anymore.

VII. Toward Spatial Data

In this section, we empirically show that, by abstracting the notion of token, our approach can be used to create a spatial index and perform trajectory-based queries. Given a trajectory, the idea is to find all the trajectories in the dataset that share some common sub-sequence with the query. Today, most trajectory indexes use spatio-temporal bounding boxes in order to create search trees. For instance, that is the case for the QuadTree, the RTree, and the TBTree [10], [19]. When dealing with trajectories, bounding boxes that span over three dimensions introduce a lot of dead space. The SETI index capitalizes on the special nature of the temporal dimension to address this discrimination issue [8]. This kind of index performs well when the queries involve spatio-temporal intervals. However, their performances decrease drastically once the queries are themselves based on trajectories. As a result, similarity and distance measures are used to filter the results and detect which trajectories actually share some common sub-sequence with the query. Computing similarities for a huge result set that includes a lot of outliers introduces an important overhead. Since our approach indexes sub-sequences, this filtering step is not necessary.

Dataset: In order to perform our experiment, we used a set of GPS trajectories gathered by Nokia from 2009 to 2011 [13]. We grouped the recorded GPS locations per user and per day to produce daily trajectories. The resulting dataset contains 32'144 distinct trajectories located in the area of Lausanne, Switzerland. GPS trajectories are composed of several GPS locations each one having several properties such as longitude, latitude and time. This kind of sequences differ from the kind of data we previously examined, since several dimensions are involved. Thus, using L, we denote the alphabet composed of all the possible longitude/latitude coordinates, and using L^*, we denote the set of all possible sequences of GPS coordinates. In our case, the notion of time is simply carried by the fact that sequences are ordered.

Normalization: GPS tracking devices are not synchronized and might showcase different sampling rates. Therefore, two persons following the same path will end-up with different sequences of GPS locations. This issue can be solved by normalizing the data. In order to find sub-sequences in the trajectories, we normalized the coordinates using a hash function called GeoHash, which subdivides the longitude/latitude coordinate system into cells [18]. In our case, GeoHash maps any longitude/latitude coordinates into cells of approximately 150m by 150m. The center of the cell is then used as the normalized coordinate. Such a normalization function could be defined as $f_n^{geo} : L^* \rightarrow L^*_{geo}$ with $L_{geo} \subseteq L$. Figure 10 illustrates the fact that, after normalization, trajectories converge toward something more identical. The sequences that result from normalizing the coordinates can directly be consumed by TB. Our configuration produces chunks that have an average length of 10 normalized coordinates which corresponds to an average distance of 1,5 kilometers by sub-sequences.

Preliminary results: We compared the resulting index with two common spatial indexes, namely the Quad Tree [10] and the Sort-Tile-Recursive Tree[22] implemented in JTS [2]. To perform our experiment, we picked a trajectory in the dataset and queried the three indexes for similar trajectories. Since the dataset is very dense, the Quad Tree returned 22'304 results and the Sort-Tile-Recursive Tree discriminated slightly better with 18'070 results, which, in both cases, represents a great number of outliers that confirms the discrimination problem. Since our inverted index has a dictionary made of unique trajectory sub-sequences, it preserves information regarding

the similarity of the trajectories. As a result, the query returned the 36 matches depicted in Figure 11 which are guaranteed to share some sub-sequences of more or less 1,5 kilometers with the query.

VIII. Related Work

The idea of computing all the overlapping sub-sequences of terms in a document was first introduced by Mamber et al. in [15]. Brin et al. also described some methods for copy detection that include n-gram and hash-based segmentation [6]. The term shingle, introduced by Broder et al. [7], is often used as a substitute to n-gram. Data deduplication is mainly used in the context of data storage and data synchronization [17], [20], [16]. It usually relies on CDC [9], [5] for identifying identical sub-sequences in data. In order to avoid redundancies, chunks are identified by their hash sums and stored in content addressable storage. Recipes consist of lists of chunk hash sums which are used to reconstruct the original data. Muthitacharoen et al [17], improve CDC by introducing a maximal and a minimal chunk size which positively impact the compression ratio. In [9], Eshghi et al. introduce a backup divisor which allows to avoid arbitrary cuts when the maximal threshold is reached. More recently, other interesting variants, such as bimodal content-defined chunking [12] and frequency-based chunking [14], have been proposed to achieve even better compression rates. Two distinct documents whose recipes share common chunks are related to each other with a high probability. This assumption has been used by Forman et al. to identify near-duplicates in very large collections of manuals and technical documents [11]. In [3], Bhagwat et al. generalize the idea of using recipes to build inverted indexes for near-duplicate search. They create an inverted index where the dictionary terms correspond to the hash sums of chunks and the postings correspond to document identifiers.

In contrast, our approach does not operates on raw data and can agnostically be used to index different kind of data. Similarity detection in textual data has been studied for several decades and lots of techniques have been investigated. For example, techniques based on local maxima and minima, sometimes referred to as Winnowing, are used to filter hash values [23], [4]. Our future work, will explore how techniques commonly used with textual datasets can be leveraged with different types of data.

IX. Conclusion & Future Work

In this paper, we introduced token-based chunking, a generic approach for finding sub-sequences in data. We studied its characteristics in terms of storage requirements, throughput, precision and recall and demonstrated that it performs better than its traditional counterparts at trading effectiveness for efficiency. We showed that, by operating on tokens, this technique can be used agnostically on various types of data. We introduced a framework that helps at identifying the extent to which data should be normalized regardless of its type. In addition, we empirically demonstrated that token-based chunking can efficiently index GPS trajectories. Finally, we showed that the resulting index have better discrimination characteristics than traditional spatial indexing approaches. To our knowledge, the usage of a segmentation methods inspired by data deduplication in the context of spatial indexes has not been explored before and our preliminary results are promising. This opens exiting new research avenues and we plan to investigate the properties of this novel kind of spatial index in the future.

References

[1] "Apache lucene," http://lucene.apache.org/.

[2] "JTS topology suite," http://www.vividsolutions.com/jts/.

[3] D. Bhagwat, K. Eshghi, and P. Mehra, "Content-based document routing and index partitioning for scalable similarity-based searches in a large corpus," in *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2007, pp. 105–112.

[4] N. Bjørner, A. Blass, and Y. Gurevich, "Content-dependent chunking for differential compression, the local maximum approach," *Journal of Computer and System Sciences*, vol. 76, no. 3-4, pp. 154–203, 2010.

[5] D. R. Bobbarjung, S. Jagannathan, and C. Dubnicki, "Improving duplicate elimination in storage systems," *ACM Transactions on Storage (TOS)*, vol. 2, no. 4, pp. 424–448, 2006.

[6] S. Brin, J. Davis, and H. Garcia-Molina, "Copy detection mechanisms for digital documents," in *ACM SIGMOD Record*, vol. 24, no. 2. ACM, 1995, pp. 398–409.

[7] A. Z. Broder, S. C. Glassman, M. S. Manasse, and G. Zweig, "Syntactic clustering of the web," *Computer Networks and ISDN Systems*, vol. 29, no. 8, pp. 1157–1166, 1997.

[8] V. P. Chakka, A. Everspaugh, and J. M. Patel, "Indexing large trajectory data sets with seti," in *CIDR*, 2003.

[9] K. Eshghi and H. K. Tang, "A framework for analyzing and improving content-based chunking algorithms," *Hewlett-Packard Labs Technical Report TR*, vol. 30, p. 2005, 2005.

[10] R. A. Finkel and J. L. Bentley, "Quad trees a data structure for retrieval on composite keys," *Acta informatica*, vol. 4, no. 1, pp. 1–9, 1974.

[11] G. Forman, K. Eshghi, and S. Chiocchetti, "Finding similar files in large document repositories," in *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*. ACM, 2005, pp. 394–400.

[12] E. Kruus, C. Ungureanu, and C. Dubnicki, "Bimodal content defined chunking for backup streams." in *FAST*, 2010, pp. 239–252.

[13] J. K. Laurila, D. Gatica-Perez, I. Aad, B. J., O. Bornet, T.-M.-T. Do, O. Dousse, J. Eberle, and M. Miettinen, "The mobile data challenge: Big data for mobile computing research," in *Pervasive Computing*, 2012.

[14] G. Lu, Y. Jin, and D. H. Du, "Frequency based chunking for data de-duplication," in *Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2010 IEEE International Symposium on*. IEEE, 2010, pp. 287–296.

[15] U. Manber *et al.*, "Finding similar files in a large file system." in *Usenix Winter*, vol. 94, 1994, pp. 1–10.

[16] D. T. Meyer and W. J. Bolosky, "A study of practical deduplication," *ACM Transactions on Storage (TOS)*, vol. 7, no. 4, p. 14, 2012.

[17] A. Muthitacharoen, B. Chen, and D. Mazieres, "A low-bandwidth network file system," in *ACM SIGOPS Operating Systems Review*, vol. 35, no. 5. ACM, 2001, pp. 174–187.

[18] G. Niemeyer, "Geohash," http://geohash.org/, 2008.

[19] D. Pfoser, C. S. Jensen, Y. Theodoridis *et al.*, "Novel approaches to the indexing of moving object trajectories," in *Proceedings of VLDB*, 2000, pp. 395–406.

[20] S. Quinlan and S. Dorward, "Venti: A new approach to archival storage." in *FAST*, vol. 2, 2002, pp. 89–101.

[21] M. O. Rabin, *Fingerprinting by random polynomials*. Center for Research in Computing Techn., Aiken Computation Laboratory, Univ., 1981.

[22] P. Rigaux, M. Scholl, and A. Voisard, *Spatial databases: with application to GIS*. Morgan Kaufmann, 2001.

[23] S. Schleimer, D. S. Wilkerson, and A. Aiken, "Winnowing: local algorithms for document fingerprinting," in *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. ACM, 2003, pp. 76–85.