ERA: Efficient Serial and Parallel Suffix Tree Construction for Very Long Strings

Essam Mansour[1] Amin Allam[1] Spiros Skiadopoulos[2] Panos Kalnis[1]

[1]Math. & Computer Sciences and Engineering
King Abdullah Univ. of Science and Technology
{fname.lname}@kaust.edu.sa

[2]Dept. of Computer Science and Technology
University of Peloponnese
spiros@uop.gr

ABSTRACT

The suffix tree is a data structure for indexing strings. It is used in a variety of applications such as bioinformatics, time series analysis, clustering, text editing and data compression. However, when the string and the resulting suffix tree are too large to fit into the main memory, most existing construction algorithms become very inefficient.

This paper presents a disk-based suffix tree construction method, called Elastic Range (ERa), which works efficiently with very long strings that are much larger than the available memory. ERa partitions the tree construction process horizontally and vertically and minimizes I/Os by dynamically adjusting the horizontal partitions independently for each vertical partition, based on the evolving shape of the tree and the available memory. Where appropriate, ERa also groups vertical partitions together to amortize the I/O cost. We developed a serial version; a parallel version for shared-memory and shared-disk multi-core systems; and a parallel version for shared-nothing architectures. ERa indexes the entire human genome in 19 minutes on an ordinary desktop computer. For comparison, the fastest existing method needs 15 minutes using 1024 CPUs on an IBM BlueGene supercomputer.

1. INTRODUCTION

The suffix tree [12] is a trie that indexes all possible suffixes of a string S (see Figure 1 for an example). It is used to accelerate many string operations. For instance, finding a substring P inside S without an index takes $\mathcal{O}(|S| + |P|)$ time [3]. With a suffix tree the same operation is done in $\mathcal{O}(|P|)$ time, which is a significant gain given that typically S is several orders of magnitude longer than P. Other operations that can benefit from a suffix tree include approximate string matching, finding the longest common substring of two strings and finding all common substrings in a database of strings. Such queries are essential for many applications such as bioinformatics [8], time series analysis [15], document clustering [4], text editing [1] and compression [5].

Proceedings of the VLDB Endowment, Vol. 5, No. 1

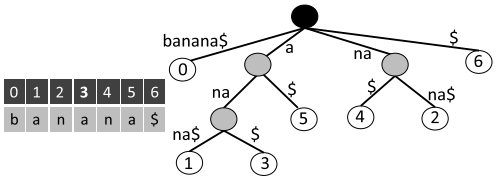


Figure 1: Suffix tree for $S = banana\$$ ($ denotes end-of-string). Edge labels on a path from the root to a leaf correspond to a suffix in S. Leaf labels show the offset of each suffix in S

Fast suffix tree construction is critical, given the rate of data generation by the aforementioned applications [9]. For example, modern DNA sequencers can process multiple samples per hour, whereas financial applications generate continuous streams of time series data. If the string S and the resulting suffix tree can fit in the main memory, there are efficient solutions such as Ukkonen's algorithm [19], which constructs the tree in $\mathcal{O}(|S|)$ time but becomes very inefficient if it runs out of memory. However, the suffix tree for S is more than an order of magnitude larger than S. The human genome, for instance, has roughly 2.6G symbols; the resulting suffix tree occupies around 67GB and challenges the main memory limits of many systems. Other algorithms, such as TRELLIS [13], work well if at least S fits in main memory, but are very inefficient otherwise.

In practice S alone can be much larger than the main memory. For example, operations that involve a database of strings [8] require a *generalized* suffix tree, which is simply the suffix tree of the concatenation of all input strings. For such cases, recently two methods have been proposed: B^{2}ST [2] and WaveFront [7]. Both access S in sequential order which is much faster than random I/Os in modern disks. The serial version of WaveFront is slower than B^{2}ST, but WaveFront is easily parallelizable; this is very important given the size of the targeted problems. Nevertheless, the performance of both algorithms deteriorates as the length of S or the size of the *alphabet* (i.e., set of symbols appearing in S) increase.

In this paper we present ERa[1], a suffix tree construction algorithm that *(a)* supports very long strings and large alphabets; *(b)* is much faster than the existing ones even if memory is very limited; and *(c)* is easily parallelizable. In a nutshell, ERa optimizes dynamically the use of memory and amortizes the I/O cost. Specifically, it divides the problem

[1]ERa stands for *Elastic Range*, for adjusting dynamically the range of the horizontal partitions.

vertically into construction of independent sub-trees, ensuring that each sub-tree can fit into the available memory. Sub-trees are further divided *horizontally* into partitions, such that each partition can be processed in memory with a single sequential scan of the input string S. At each step, horizontal partitions are readjusted based on the evolving shape of the tree, in order to maximize memory utilization. Also, vertical partitions may be grouped together in order to share the I/O cost. The entire plan can be executed in a serial or parallel system.

Our contributions include:

- A serial version of ERa that is *at least* 50% faster than existing serial algorithms. Performance gain is more dramatic for very long strings and large alphabets.
- A parallel version for shared-memory and shared-disk architectures that include ordinary multicore desktop systems.
- A parallel version for shared-nothing systems, such as clusters or cloud computing infrastructures.
- Extensive experimental evaluation with real datasets of very long strings. ERa indexes the entire human genome in 19 minutes on an ordinary 8-core desktop computer with 16GB of RAM. For comparison, the fastest existing method (i.e., the parallel version of WaveFront [6]) needs 15 minutes on an IBM BlueGene/L supercomputer using 1024 CPUs and 512GB of RAM.

The rest of this paper is organized as follows. Section 2 furnishes the preliminaries of suffix trees, whereas Section 3 discusses the related work. Section 4 introduces the serial version of ERa and Section 5 presents the parallel versions. Section 6 discusses our experimental results. Section 7 concludes the paper.

2. BACKGROUND: SUFFIX TREE

Let Σ denote an *alphabet* (i.e., set of symbols). An input string S of length $n+1$ is a sequence $S = s_0 s_1 \cdots s_{n-1}\$$, where $s_i \in \Sigma$, $0 \leq i \leq n-1$ and $\$ \notin \Sigma$; $\$$ is the end-of-string symbol. A *prefix* of S is sequence $s_0 \cdots s_i$ and a *suffix* of S, denoted by S_i, is $s_i \cdots \$$ $(0 \leq i \leq n)$. In this paper, we will consider prefixes of S and prefixes of suffixes of S. To avoid confusion, we will refer to the latter by *S-prefixes*. The unique terminal symbol $\$$ ensures that no suffix S_i is a proper S-prefix of any other suffix S_j $(i \neq j)$.

A *suffix tree* $\mathcal{T}$ is a trie that indexes all suffixes of S. In the rest of the paper, we will use the example string and corresponding tree of Figure 2; the alphabet consists of four symbols $\{A, C, G, T\}$ which is typical in bioinformatics. The main properties of the suffix tree are:

- There exist exactly $n+1$ leaves with node labels from 0 to n. For any leaf v_i, the concatenation of the edge labels on the path from the root to v_i spells out suffix S_i. For example v_{20} corresponds to $S_{20} = TGC\$$.
- Each internal node other than the root, has at least two children and each edge is labeled with a S-prefix of S. If, during construction, a node appears with only one child, then the node and its child are merged and the edge labels are concatenated (this explains edge labels with more than one symbol in the example).
- No two edges out of a node can have edge labels beginning with the same symbol.

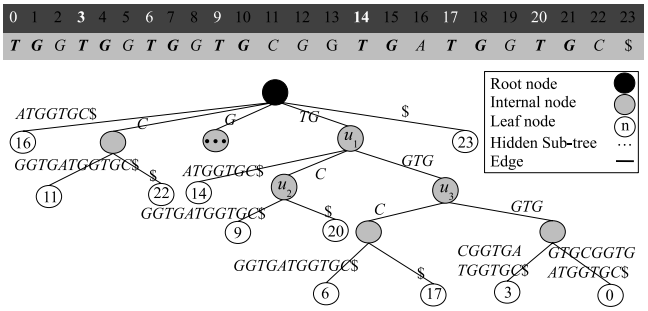


Figure 2: Input string S, where $\Sigma = \{A, C, G, T\}$, and corresponding suffix tree $\mathcal{T}$. For simplicity, $\mathcal{T}_G$ (i.e., the sub-tree under G) is not shown

i	S_i	**Suffix**
0	S_0	**TG**$GTGGTGGTGCGGTGATGGTGC\$$
3	S_3	**TG**$GTGGTGCGGTGATGGTGC\$$
6	S_6	**TG**$GTGCGGTGATGGTGC\$$
9	S_9	**TG**$CGGTGATGGTGC\$$
14	S_{14}	**TG**$ATGGTGC\$$
17	S_{17}	**TG**$GTGC\$$
20	S_{20}	**TG**$C\$$

Table 1: Suffixes sharing the S-prefix TG. i refers to the offset of the suffix in the string of Figure 2

The suffix tree can be divided into a set of sub-trees; $\mathcal{T}_p$ denotes the sub-tree that indexes suffixes sharing a S-prefix p. In the example, $\mathcal{T}$ is divided into $\mathcal{T}_A$, $\mathcal{T}_C$, $\mathcal{T}_G$, $\mathcal{T}_{TG}$, and $\mathcal{T}_\$$. Table 1 shows all suffixes with S-prefix TG; these suffixes will be indexed in $\mathcal{T}_{TG}$. The frequency f_p of a S-prefix p is the number of suffixes in $\mathcal{T}_p$. For example, $f_{TG} = 7$, whereas $f_A = 1$. As we will see later, the frequency is proportional to the amount of memory needed for the construction of the sub-tree. Given the available memory, we can bound the maximum frequency of all p below a threshold by using variable length S-prefixes [7]. For example, each of the S-prefixes in the set $\{A, C, TGA, TGC, TGGTGC, TGGTGGTG\}$ has frequency at most 2. Note that, reducing the maximum frequency increases the number of sub-trees.

Storing S-prefixes in the edge labels requires $\mathcal{O}(n^2)$ space for the tree. Typically, a suffix tree edge stores only two integers representing the starting and the ending index of the S-prefix in S. Using this representation space complexity drops to $\mathcal{O}(n)$. The figures throughout the paper show S-prefixes for clarity. Also, we sort the edges that emanate from a node according to the lexicographical order of their labels. Thus, a depth first search traversal will result in suffixes in lexicographical order.

3. RELATED WORK

This section presents the most important suffix tree construction algorithms classified into three main categories: *in-memory*, *semi-disk-based*, and *out-of-core*. Table 2 summarizes the comparison. It is worth to note that even though suffix trees are useful in a wide range of applications, there also exist specialized index structures for particular applications like genome data [11] and time series analysis [16].

In-memory approaches perform very well as long as the input string and the resulting suffix tree fit in main memory. This category includes algorithms, such as McCreight's

	In-memory	Semi-disk-based				Out-of-core	
Criteria	**Ukkonen**	**Hunt**	**TDD**	**ST-Merge**	**TRELLIS**	**WaveFront**	**B^{2}ST**
Complexity	$O(n)$	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n^2)$
Memory locality	Poor	Good	Good	Good	Good	Good	Good
String access	Random	Random	Random	Random	Random	Sequential	Sequential
Parallel	No	No	No	No	No	Yes	No

Table 2: Comparison of the most important algorithms for suffix tree construction

[12] and Ukkonen's [19]. For a string S of size n, the time complexity of the latter is $\mathcal{O}(n)$, which is optimal. However, this category suffers from poor locality of reference [18]. Once the suffix tree cannot fit in the main memory, the algorithms of this category require on average $\mathcal{O}(n)$ expensive random disk I/Os. Recall that the suffix tree is an order of magnitude larger than the input string. Therefore, in practice in-memory methods are prohibitively expensive even for moderately long strings.

Semi-disk-based methods solve the locality of reference problem by decomposing the suffix tree into smaller subtrees stored on the disk. This category includes Hunt's algorithm [10], TDD [17], ST-Merge [18] and TRELLIS [13]. The latter partitions the input string into several substrings, and constructs the corresponding sub-tree independently for each substring. The resulting sub-trees are stored on the disk. In a second phase, the sub-trees are merged into the final suffix tree. The time complexity is $\mathcal{O}(n^2)$, but as long as the string S fits into memory, the algorithms in this category perform few random I/Os so in practice they are faster than Ukkonen's algorithm. However, if S is larger than the memory, the merging phase generates a lot of random disk I/Os rendering these algorithms very inefficient [2, 7]. It is worth noting that the sub-tree construction phase can be parallelizable but the merging phase is expected to require a lot of communication among processors. We are not aware of any parallel version of semi-disk-based algorithms.

Out-of-core category contains two recent methods that support strings larger than the main memory with reasonable efficiency by avoiding random I/Os. The first method, B^{2}ST [2] is based on suffix arrays [14]. A suffix array is a vector that contains all suffixes of the input string S sorted in lexicographical order. A longest common prefix array is a vector that stores the length of the common prefix between each two consecutive entries in the suffix array. B^{2}ST divides the input string S into several partitions and builds the corresponding suffix array and longest common prefix array for each partition. Then, it merges the suffix arrays of all partitions and generates suffix sub-trees. Note that the tree is constructed in batch at the final phase. This is an advantage of the algorithm because by avoiding the tree traversal for the insertion of each new node, it is more cache friendly. The time complexity is $\mathcal{O}(cn)$, where $c = (2n/M)$ and M is the size of the main memory. If M is comparable to n then c is considered constant and the algorithm performs very well. However, as we mention in Section 1, in practice n is expected to be much larger than M; in such a case the complexity becomes $\mathcal{O}(n^2)$. A drawback of B^{2}ST is the large size of temporary results. The human genome for example is roughly 2.6G symbols whereas the temporary results are around 343GB. Furthermore, a parallel version of the algorithm would incur high communication cost among the processors during the merging phase; we are not aware of any parallel implementation.

WaveFront [7] is the second out-of-core algorithm. In contrast to B^{2}ST, which partitions the input string S, WaveFront works with the entire S on independent partitions of the resulting tree $\mathcal{T}$. Tree partitioning is done using variable length S-prefixes (see example in Table 1), making sure that each sub-tree fits in main memory. Since S may not fit in memory the algorithm may need to read S multiple times. To minimize the I/O cost, WaveFront accesses S strictly in sequential order. Each sub-tree is processed independently without a merging phase, so the algorithm is easily parallelizable. The parallel version has been implemented on an IBM BlueGene/L supercomputer; in absolute time it is the fastest existing method (it can index the human genome in 15 minutes [6]). Nevertheless, the algorithm cannot scale indefinitely, because more sub-trees increase the so-called *tiling overhead* [7]. Internally, WaveFront resembles the block nested loop join algorithm and requires two buffers. For optimum performance, these buffers occupy roughly 50% of the available memory, leaving the rest for the sub-tree. This is a drawback of the algorithm, because less memory leads to smaller and more trees that increase the tiling overhead. Moreover, even though the algorithm expands the sub-tree in layers, it needs to traverse the tree top-down for every new node, increasing the CPU cost.

Our approach, Elastic Range (ERa) is closer to WaveFront, therefore there is no merging phase and it is easily parallelizable. However, ERa is significantly faster than WaveFront since it is based on properties that allow a level by level construction mechanism that performs clever memory management and minimizes the tiling overhead. Also, ERa amortizes the I/O cost by grouping together sub-trees where appropriate. Finally, ERa avoids multiple traverses of the sub-tree, achieving much lower CPU cost.

4. ELASTIC RANGE (ERa)

Elastic Range (ERa) is a novel approach that divides the problem vertically and horizontally (see Figure 3). Vertical partitioning splits the tree into sub-trees $\mathcal{T}_{p_1} \dots \mathcal{T}_{p_n}$ that fit into the available memory using variable length S-prefixes similarly to [7, 10]. ERa goes a step further by grouping together sub-trees to share the I/O cost of accessing the input string S. Horizontal partitioning is applied independently in each sub-tree in a top-down fashion. The width of the horizontal partitions is adjusted dynamically (hence the name elastic range) based on how many paths in the sub-tree are still being processed. This allows ERa to use only a small part of the memory for buffers, rendering the algorithm cache-friendly and minimizing the tiling overhead. Each group represents an independent unit; groups can be processed serially or in parallel. The resulting sub-trees are assembled in the final suffix tree by a trie on the top. The trie is constructed with the S-prefixes used for vertical partitioning and is very small (e.g., the trie for the human genome is in the order of KB). The rest of this section describes the

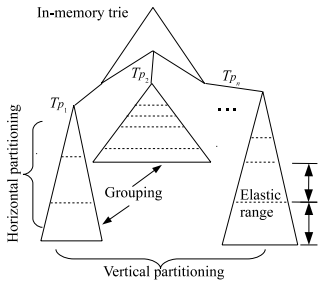


Figure 3: Problem decomposition in ERa

serial version of ERA. The parallel versions are discussed in Section 5.

4.1 Vertical Partitioning

Let p be a S-prefix and $\mathcal{T}_p$ be the sub-tree that corresponds to p. Recall from Section 2 that f_p is the number of suffixes with S-prefix p (see example in Table 1). Each suffix corresponds to a leaf node in $\mathcal{T}_p$ and it is shown [10] that the number of internal nodes is equal to the number of leaves. The size in bytes of $\mathcal{T}_p$ is $2f_p \cdot sizeof(tree\_node)$. Let MTS be the size of the memory reserved for the sub-tree. $\mathcal{T}_p$ can fit in the memory only if $f_p \leq \mathcal{F}_M$, where

$$\mathcal{F}_M = \frac{MTS}{2 \cdot sizeof(tree\_node)} \quad (1)$$

To partition $\mathcal{T}$ into sub-trees that fit in MTS, we employ the idea of variable length S-prefixes [7, 10]. The algorithm starts by creating a working set containing one S-prefix for every symbol in the alphabet Σ. Then the entire input string S is scanned to calculate the frequencies of each S-prefix in the working set. At the end of this step, each S-prefix whose frequency is at most $\mathcal{F}_M$ is removed from the working set. The remaining S-prefixes are extended by one symbol and the process is repeated until the working set is empty. In the example of Table 1, assume $\mathcal{F}_M = 5$. Since $f_{TG} = 7$, we extend TG by one symbol and get $f_{TGA} = 1$, $f_{TGC} = 2$ and $f_{TGG} = 4$ that are all at most 5 and are removed from the working set; note that $f_{TGT} = 0$ since there is no TGT substring in S. The worst case complexity is $\mathcal{O}(n^2)$ time, where n is the size of S. In practice, for typical values of MTS the algorithm runs in $\mathcal{O}(n)$. The human genome, for instance, requires 5 to 6 iterations when MTS is 1 to 2GB.

However, the algorithm from [7] has a serious drawback: it generates unbalanced sub-trees that waste a lot of memory. In the previous example, the available memory can support frequencies up to $\mathcal{F}_M = 5$ but the frequencies of the resulting sub-trees are much smaller. Each small sub-tree is processed independently and accesses S multiple times; therefore there are a lot of redundant I/Os. Also, a parallel implementation would waste resources because the CPUs that process the smaller sub-trees will be idle for long time. To avoid these problems, we propose a grouping phase after the initial partitioning.

We use a simple heuristic for grouping: The set of S-prefixes from the previous phase are put in a linked list sorted in descending frequency order. The head of the list (i.e., the S-prefix with the highest frequency) is added in a new group. Then, the list is traversed and S-prefixes are added to the group as long as the sum of the frequencies in the group is at most $\mathcal{F}_M$. The process is repeated until all S-prefixes are processed (see Algorithm VERTICALPARTITIONING). In the previous example, this heuristic groups TGG and TGA together, whereas TGC is in a different group. TGG and TGA share a common S-prefix TG but this is a coincidence. The algorithm works with all S-prefixes generated from S and may group together two or more completely unrelated S-prefixes.

Algorithm: VERTICALPARTITIONING
Input: String S, alphabet Σ, $\mathcal{F}_M$ (see Equation 1)
Output: Set of $VirtualTrees$

```
1  VirtualTrees := ∅
2  P := ∅                                   // linked list of S-prefixes
3  P' := { for every symbol s ∈ Σ do generate a S-prefix p_i = s }
4  repeat
5      scan input string S
6      count in S the appearances f_{p_i} of every S-prefix p_i ∈ P'
7      for every p_i ∈ P' do
8          if 0 < f_{p_i} ≤ F_M then add p_i to P
9          else for every symbol s ∈ Σ do add p_i s to P'
10         remove p_i from P'
11 until P' = ∅;
12 sort P in descending f_{p_i} order
13 repeat
14     G := ∅            // group of S-prefixes in a virtual tree
15     add P.head to G and remove the item from P
16     curr := next item in P
17     while NOT end of P do
18         if f_curr + SUM_{g_i ∈ G}(f_{g_i}) ≤ F_M then
19             add curr to G and remove the item from P
20         curr := next item in P
21     add G to VirtualTrees
22 until P = ∅;
23 return VirtualTrees
```

A group of S-prefixes defines a *virtual* sub-tree that is processed as a single unit. When the input string S is read from the disk, it is used by the entire group, therefore the I/O cost is amortized. Also, in a parallel environment, the utilization of resources is much better. Obviously, when MTS is large, more sub-trees can be grouped together and the gain is larger.

4.2 Horizontal Partitioning

During this step ERA constructs the suffix sub-tree $\mathcal{T}_p$ for a S-prefix p where $\mathcal{T}_p$ fits in the available main memory budget (Section 4.1). We base our method on properties of the suffix-tree (Proposition 1) that have not be exploited by previous approaches. Initially, in Section 4.2.1, we devise Algorithm COMPUTESUFFIXSUBTREE that exploits these properties to optimize access to the input string S. Then, Section 4.2.2 further extends the idea to also optimize main memory access (Algorithm SUBTREEPREPARE).

4.2.1 Optimizing String Access

To illustrate the key idea of our method, we will need the following notation. Let e be an edge of $\mathcal{T}_p$. We denote by *(i)* $label(e)$ the label of e, *(ii)* $parent(e)$ the unique parent of e, and *(iii)* $pathlabel(e)$ the concatenation of edge labels on the path from the root to e. We consider nodes u_1, u_2 and u_3 of the suffix-tree illustrated in Figure 2 and make the following observations:

1. If an edge e connects to a leaf then $pathlabel(e)$ appears only once in S. For instance, edge $e = (u_1, 14)$ that connects to leaf 14 has $pathlabel(e) = TGA \cdots \$$ that appears only once in S.
2. If an edge e has a label of more than one symbols, say

Algorithm: ComputeSuffixSubTree

Input: String S, S-prefix p
Output: The suffix sub-tree $\mathcal{T}_p(Nodes, Edges)$

```
1 root := new Node(root)
2 u' := new Node
3 e' := new Edge(root, u')
4 Label e' with S-prefix p
5 BranchEdge(S, T_p(Nodes, Edges), e')
6 return T_p(Nodes, Edges)
```

Algorithm: BranchEdge

Input: String S, suffix sub-tree $\mathcal{T}_p(Nodes, Edges)$, edge $e(u_1, u_2)$

```
1  Y is a set containing the symbols that follow pathlabel(e) in S
2  if pathlabel(e) appears once in S then                  // Leaf node
3      Label e with label(e) ··· $
4  else if |Y| = 1 then // Same symbol s_1 after pathlabel(e) in S
5      Extend the label of e to include symbol s_1
6      BranchEdge(S, T_p(Nodes, Edges), e)
7  else for each s_i do
8      u' := new Node
9      e' := new Edge(u_2, u')
10     Label e' with s_i
11     BranchEdge(S, T_p(Nodes, Edges), e')
```

$s_1 s_2 s_3 \cdots$, then $pathlabel(parent(e)) \cdot s_1$ is always followed by s_2 in S, $pathlabel(parent(e)) \cdot s_1 s_2$ is always followed by s_3 in S and so on. For instance, for edge $e = (u_1, u_3)$ having label GTG, TGG is always followed by T in S where $TG = pathlabel(parent(e))$.

3. If an edge e is branched into another edge e' then $pathlabel(e) \cdot s$, where s is the first symbol of $label(e')$, appears (at least once) in S. For instance, edge $e = (u_1, u_2)$ is branched into edge $e' = (u_2, 9)$ and $TGCG$ appears in S, where *(i)* $TGC = pathlabel(e)$ and *(ii)* G is the first symbol of $label(e')$.

Interestingly, the above observations are general properties of the suffix-tree that are captured formally by the following proposition.

Proposition 1. *Let S be a string and e an edge of its suffix-tree.*

1. *Edge e is connected to a leaf node iff $pathlabel(e)$ appears only once in S.*
2. *If $label(e) = s_1 \cdots s_k$, then substring $pathlabel(parent(e)) \cdot s_1 \cdots s_{i-1}$ is always followed by s_i in S $(1 < i \leq k)$.*
3. *Edge e is branched into edges $e^1, \ldots, e^j$ iff $pathlabel(e) \cdot s^i$ $(1 \leq i \leq j)$ appears at least once in S where $s^1, \ldots, s^j$ are distinct symbols formed by the first symbols of $label(e^1)$, $\ldots$, $label(e^j)$ respectively.*

Contrary to previous suffix-tree construction approaches, Proposition 1 provides us with a method to build the suffix sub-tree $\mathcal{T}_p$ of a S-prefix p level by level in a breadth-first fashion. This is achieved using Algorithms ComputeSuffixSubTree and BranchEdge. In more detail, Algorithm ComputeSuffixSubTree creates an edge e' labeled with p. Then, BranchEdge is executed; it computes set Y that stores all symbols appearing after p in S. Following, the algorithm considers the 3 cases identified by Proposition 1. An example is shown below:

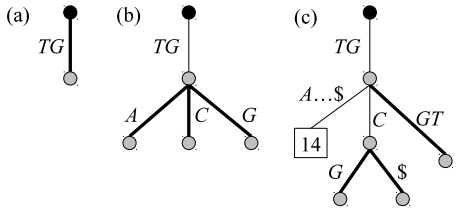


Figure 4: Constructing the suffix sub-tree of TG (Example 1). Thin edges are finalized while thick edges need further processing.

Example 1. *Let us construct, using Algorithm* ComputeSuffixSubTree*, the suffix sub-tree $\mathcal{T}_{TG}$ of S-prefix TG for the string S presented in Figure 2. In each step of the algorithm, we illustrate in Figure 4 the constructed suffix-tree.*

Initially, Algorithm ComputeSuffixSubTree *creates an edge labeled with TG and executes Algorithm* BranchEdge *(Figure 4(a)). Since $Y = \{A, C, G\}$, Algorithm* BranchEdge *branches the current edge into 3 new edges labeled with A, C and G (Figure 4(b)) and is recursively executed for these new edges. While processing these edges, the algorithm determines that* (a) *the edge labeled with A connects to a leaf with label 14 (i.e., the offset of S-prefix $TGA\ldots\$$ in S),* (b) *the edge labeled with C should be branched into two new edges labeled with G and \$ respectively and* (c) *the edge labeled with G should be extended to include symbol T (Figure 4(c)). Algorithm* BranchEdge *proceeds in a similar manner until sub-tree $\mathcal{T}_p$ is created.*

The heart of the suffix sub-tree construction mechanism lies in Algorithm BranchEdge. For the clarity of presentation, this algorithm is illustrated in its simplest form (i.e., recursive) and without any optimization. The most costly operation of Algorithm BranchEdge is the construction of set Y since it requires a complete scan of the input string S. Such a scan is required for every modified edge. For instance, in Figure 4(b) BranchEdge scans S three times, one for each thick edge (labeled with A, C and G). Also, for each scan the algorithm reads and stores in Y only one symbol after each occurrence of $pathlabel(e)$ in S.

The actual implementation of BranchEdge is iterative (non recursive) and has three major optimizations:

1. The cost of scanning S is amortized for all the edges of a level. For Figure 4(b), a single scan of S is needed to process all thick edges A, C and G.
2. For each scan of S, the algorithm reads a range of symbols. This means that Y is now a set of strings (instead of a set of symbols). The exact size of the range depends on the available main memory (see Section 4.4). In total, by reading l symbols, we reduce the scans of S by a factor l.
3. The algorithm constructs the sub-tree $\mathcal{T}_p$ for a S-prefix p. If more sub-trees are grouped in the same virtual tree (see Section 4.1), then each scan of S updates the edges of all sub-trees in the virtual tree.

From this point onwards, we consider that the ComputeSuffixSubTree algorithm uses the above described optimized version of BranchEdge.

4.2.2 Optimizing Memory Access

The experimental evaluation and profiling of the Algorithm ComputeSuffixSubTree showed that a significant

amount of time is spent on updating the constructed (in main memory) suffix sub-tree $\mathcal{T}_p$. This is mainly due to the fact that the construction process requires memory accesses that may not be sequential nor local. To address this issue, we propose a novel two step approach formed by a *preparation* and a *construction* step.

The preparation step is executed by Algorithm SubTreePrepare. It extends the optimized version of BranchEdge and employs Proposition 1 to construct *not* the sub-tree but a novel intermediate data structure.

The construction step is performed by Algorithm BuildSubTree that utilizes the data structure produced by the preparation step to construct the suffix sub-tree in batch. By decoupling the sub-tree construction from the preparation, we localize memory accesses and avoid costly traversals of the partial sub-tree for each new node.

The crux of the proposed method is the intermediate data structure. It consists of array **L** that stores the leaves and array **B** that stores branching information. More precisely, array **L** stores the positions of the input S-prefix p in S, i.e., the leaves of the sub-tree. The order of the leaves in **L** is such that the corresponding suffixes are lexicographically sorted, i.e., $S_{L[i]} \leq S_{L[i+1]}$. Array **B** is more involved and consists of triplets of the form $(c_1, c_2, \mathit{offset})$ where c_1 and c_2 are symbols and *offset* is an integer. Intuitively, $\mathbf{B}[i]$ describes the relation between the branch Br_{i-1} that leads to $\mathbf{L}[i-1]$ and the branch Br_i that leads to $\mathbf{L}[i]$ $(1 \leq i)$. Specifically, *offset* is the number of symbols in the common path of Br_{i-1} and Br_i (this corresponds to the size of the common longest S-prefix of $S_{\mathbf{L}[i-1]}$ and $S_{\mathbf{L}[i]}$). Symbol c_1 (respectively c_2) is the first symbol of the branch to $\mathbf{L}[i-1]$ (respectively $\mathbf{L}[i]$) after their separation. For instance, using **L** and **B** we can represent $\mathcal{T}_{TG}$ (Figure 2) as follows:

	0	1	2	3	4	5	6
L	14	9	20	6	17	3	0
B		$(A, C, 2)$	$(G, \$, 3)$	$(C, G, 2)$	$(G, \$, 6)$	$(C, G, 5)$	$(C, G, 8)$

For example, *(a)* $\mathbf{L}[0] = 14$ since the lexicographically smallest suffix is S_{14} (Table 1), and *(b)* $\mathbf{B}[5] = (C, G, 5)$ since the branch leading to $\mathbf{L}[4] = 17$ separates after 5 symbols (i.e., $TGGTG$) from the branch leading to $\mathbf{L}[5] = 3$ and C, G are the first symbols after the separation (Figure 2).

To compute arrays **L** and **B** that correspond to the sub-tree $\mathcal{T}_p$ of a S-prefix p, we employ Algorithm SubTreePrepare (recall that **L** stores the leaves and **B** the branching information of $\mathcal{T}_p$). The algorithm uses 4 auxiliary data structures of size $|\mathbf{L}|$, namely **I**, **A**, **R** and **P**. During the process the order of the elements in **A**, **R**, **P** and **L** may change. Intuitively:

Array R is the main memory buffer of the input string S. Specifically, $\mathbf{R}[i]$ stores symbols required to construct the branch leading to leaf $\mathbf{L}[i]$.

Array I is an index that holds information that restores the original order of leaves in the string S. More precisely, the position of the ith leaf in S may be accessed using $\mathbf{L}[\mathbf{I}[i-1]]$. If $\mathbf{I}[i] = \mathit{done}$ then the corresponding branch is completed. In other words, leaves $\mathbf{L}[\,\mathbf{I}[0]\,], \ldots, \mathbf{L}[\,\mathbf{I}[|\mathbf{L}|-1]\,]$ appear in that order in S. Thus, to fill **R**, S is sequentially read until the symbols pertaining to leaves $\mathbf{L}[\mathbf{I}[0]], \ldots, \mathbf{L}[\mathbf{I}[|\mathbf{L}|-1]]$ are found and stored in $\mathbf{R}[\,\mathbf{I}[0]\,], \ldots, \mathbf{R}[\,\mathbf{I}[|\mathbf{L}|-1]\,]$, respectively (Lines 10-12 of SubTreePrepare). Overall, array **I** is of paramount importance since it allows us to fill the buffers of **R** in a single sequential scan of S (and thus retain the properties of Algorithm BranchEdge).

Algorithm: SubTreePrepare

Input: Input string S, S-prefix p
Output: Arrays **L** and **B** corresponding suffix sub-tree $\mathcal{T}_p$

```
1  L contains the locations of S-prefix p in string S
2  B := ()
3  I := (0, 1, ..., |L| − 1)
4  A := (0, 0, ..., 0)
5  R := ()
6  P := (0, 1, ..., |L| − 1)
7  start := |p|                                  // Start after S-prefix p
8  while there exist an undefined B[i], 1 ≤ i ≤ |L| − 1 do
9      range := GetRangeOfSymbols                 // Elastic range
10     for i := 0 to |L| − 1 do
11         if I[i] ≠ done then
12             R[ I[i] ] := ReadRange(S, L[ I[i] ] + start, range)
                   // ReadRange(S, a, b) reads b symbols of S
                   starting at position a
13     for every active area AA do
14         Reorder the elements of R, P and L in AA so that R is
           lexicographically sorted. In the process maintain the
           index I
15         If two or more elements {a_1, ..., a_t} ∈ AA, 2 ≤ t, exist
           such that R[a_1] = ··· = R[a_t] introduce for them a new
           active area
16     for all i such that B[i] is not defined, 1 ≤ i ≤ |L| − 1 do
17         cs is the common S-prefix of R[i − 1] and R[i]
18         if |cs| < range then
19             B[i] := (R[i − 1][|cs|], R[i][|cs|], start + |cs|)
20             if B[i − 1] is defined or i = 1 then
21                 Mark I[ P[i − 1] ] and A[i − 1] as done
22             if B[i + 1] is defined or i = |L| − 1 then
23                 Mark I[ P[i] ] and A[i] as done // Last element
                       of an active area
24     start := start + range
25 return (L, B)
```

Array A identifies the active areas of the process. Elements i and $i+1$ belong to the same active area if $\mathbf{A}[i] = \mathbf{A}[i+1]$. If $\mathbf{A}[i] = \mathit{done}$ then element i is completed.

Array P stores the order of appearance in the string S of the leaves in **L**. If $\mathbf{P}[i] = x$ then leaf $\mathbf{L}[i]$ corresponds to the $x+1$ appearance of S-prefix p in S. **P** is used in Lines 21 and 23.

We will illustrate Algorithm SubTreePrepare using the following example.

Example 2. *We will construct arrays* **L** *and* **B** *of the suffix sub-tree* $\mathcal{T}_{TG}$ *of S-prefix* TG *for the string* S *presented in Figure 2. The algorithm starts by initializing all necessary structures (*$\mathbf{I}, \mathbf{A}, \mathbf{R}, \mathbf{L}$ *and* **B***). Then, the algorithm decides to read ranges of 4 symbols*[2] *from the input string* S *(range = 4) to fill the buffers of* **R** *(Lines 9-12). The values of the variables up to this point are as follows:*

Trace 1

	0	1	2	3	4	5	6
I	0	1	2	3	4	5	6
A	0	0	0	0	0	0	0
R	*GTGG*	*GTGG*	*GTGC*	*CGGT*	*ATGG*	*GTGC*	*C$*
P	0	1	2	3	4	5	6
L	0	3	6	9	14	17	20

For instance, if $i = 3$ *the algorithm considers position* $\mathbf{I}[3] = 3$ *and reads from* S *symbols CGGT that correspond*

[2]We will discuss how the range is determined in Section 4.4

to the range $= 4$ *symbols after position* $\mathbf{L}[\,\mathbf{I}[i]\,] + start = 9 + 2 = 11$.

Following (Lines 13-15), the algorithm considers all elements (since they belong to the same active area marked with 0*) and reorders* $\mathbf{R}$, $\mathbf{P}$ *and* $\mathbf{L}$ *so that* $\mathbf{R}$ *is lexicographically sorted, while maintaining array* $\mathbf{I}$. *The algorithm continues executing Lines 16-23 which compute array* $\mathbf{B}$. *The results of this iteration are illustrated below:*

Trace 2

	0	1	2	3	4	5	6
I	5	6	3	*done*	*done*	4	*done*
A	*done*	*done*	*done*	1	1	2	2
R	*ATGG*	*CGGT*	*C*\$	*GTGC*	*GTGC*	*GTGG*	*GTGG*
P	4	3	6	2	5	0	1
L	14	9	20	6	17	0	3
B		$(A, C, 2)$	$(G, \$, 3)$	$(C, G, 2)$		$(C, G, 5)$	

Note that Lines 13-15 place at the fourth position $(i = 3)$ *leaf* $\mathbf{L}[3] = 6$ *that corresponds to the lexicographically fourth suffix of TG (i.e.,* S_6*). The fact that the current position (*3*) was moved from position* 2 *of the initial order is marked by* $\mathbf{I}[2] = 3$. *Also, Line 15 identifies two more active areas denoted by* 1 *and* 2 *in* $\mathbf{A}$.

Also note that, for $i = 1$, *Lines 16-23 focus on* $\mathbf{R}[0] = ATGG$ *and* $\mathbf{R}[1] = CGGT$, *which do not have a common S-prefix (i.e,* $|cs| = 0$*). Thus, the algorithm sets* (a) $\mathbf{B}[1] = (\mathbf{R}[0][0], \mathbf{R}[i][0], 2 + 0) = (A, C, 2)$ *and* (b) $\mathbf{I}[\mathbf{P}[0]] = \mathbf{I}[4] =$ *done and* $\mathbf{A}[0] =$ *done. The equation in* (a) *illustrates that sub-tree branches after start* $+ |cs| = 2$ *symbols (i.e., TG) and follows from Proposition 1, Case 3. The equations in* (b) *show that suffix* $S_{\mathbf{L}[0]} = TGATGG \cdots \$$, *that appears only once in S, does not need any further processing (follows from Proposition 1, Case 1).*

In each iteration, the construction of $\mathbf{B}$ *involves sequential access of array* $\mathbf{R}$ *and can be performed very efficiently by a single memory scan. The next (and final) iteration of the while loop (Lines 8-24) considers only* $i \in \{0, 1, 2, 5\}$ *for which* $\mathbf{I}[i] \neq$ *done and fills only the appropriate elements of* $\mathbf{R}$ *(*$5, 6, 3, 4$ *respectively). After the execution of Lines 13-23, the structures are update as follows:*

Trace 3

	0	1	2	3	4	5	6
I	*done*	*done*	*done*	*done*	*done*	*done*	*done*
A	*done*	*done*	*done*	*done*	*done*	*done*	*done*
R				*GGTG*	\$	*TGCG*	*TGGT*
P	4	3	6	2	5	0	1
L	14	9	20	6	17	3	0
B		$(A, C, 2)$	$(G, \$, 3)$	$(C, G, 2)$	$(G, \$, 6)$	$(C, G, 5)$	$(C, G, 8)$

Note the reorder of the elements of $\mathbf{R}$, $\mathbf{P}$ *and* $\mathbf{L}$ *for* $i \in \{5, 6\}$ *that correspond to the lexicographical sorting of active area tagged with* 2.

Summarizing, Algorithm SubTreePrepare retains the sequential access of the input string S (using array $\mathbf{I}$) but also constructs $\mathbf{I}$ and $\mathbf{B}$ using sequential main memory access. Algorithm BuildSubTree takes these structures and builds the corresponding suffix sub-tree using also sequential memory access.

Example 3. *We continue Example 2. Algorithm* BuildSubTree *creates an edge that links the root with the lexicographically first leaf* $\mathbf{L}[0] = 14$. *This edge is labeled with*

Algorithm: BuildSubTree
Input: Arrays $\mathbf{L}$ and $\mathbf{B}$
Output: The corresponding suffix sub-tree $\mathcal{T}_p$

```
1  root := new Node(root)
2  u' := new Node
3  e' := new Edge(root, u')
4  Label e' with S_{L[0]}          // The suffix that corresponds L[0]
5  Label u' with L[0]               // First (lexicographically) leaf
6  Push e' to Stack
7  depth := |label(e')|
8  for i := 1 to |B| − 1 do
9      (c1, c2, offset) := B[i]
10     repeat
11         Pop an edge se(v1, v2) from the Stack
           depth := depth − |label(se)|
12     until depth ≤ offset;
13     if depth = offset then
14         u := v1
15     else
16         Break edge se(v1, v2) into edges se1(v1, vt) and
           se1(vt, v2)
17         Label se1 with the first offset symbols of label(se)
18         Label se2 with the remaining symbols
19         u := vt
20         Push se1 to Stack
21         depth := depth + |label(se1)|
22     u' := new Node
23     ne := new Edge(u, u')
24     Label ne with S_{L[i]}       // The suffix that corresponds L[i]
25     Label u' with L[i]           // Next (lexicographically) leaf
26     Push ne to Stack
27     depth := depth + |label(ne)|
28 return T_p
```

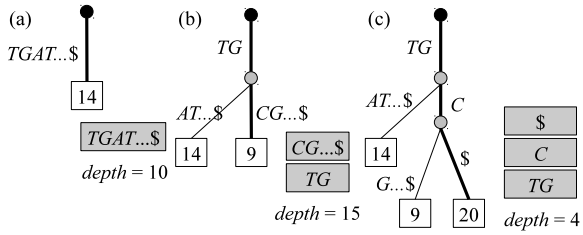


Figure 5: Trace of the BuildSubTree algorithm for the first four entries of Trace 3. The stack is depicted in gray next to each sub-tree

$TGAT \cdots \$$, *which is the suffix that corresponds to* $\mathbf{L}[0]$. *Also this edge is pushed to the stack and variable depth is initialized to* 10 *(i.e., the size of the label). All the above are illustrated in Figure 5(a). Then, the algorithm goes through the following iterations:*

The $\mathbf{1}^{st}$ **iteration** *considers* $\mathbf{B}[1] = (c_1, c_2, offset) = (A, C, 2)$. *Edge* $TGAT \cdots \$$ *is popped. Since offset* $= 2$, *this edge breaks into two edges labeled with TG and* $AT \cdots \$$ *(Lines 15-21). Moreover, a new edge is created that* (a) *links edge TG and the second leaf* $\mathbf{L}[1] = 9$ *and* (b) *is labeled with* $CG \cdots \$$. *Also, edges TG and* $CG \cdots \$$ *are pushed to Stack and depth* $= |label(TG)| + |label(CG \cdots \$)| = 2 + 13 = 15$. *This iteration is depicted in Figure 5(b).*

The $\mathbf{2}^{nd}$ **iteration** *considers* $\mathbf{B}[2] = (G, \$, 3)$ *and proceeds in a similar manner. It is depicted in Figure 5(c).*

The remaining iterations are similar and are omitted.

4.3 Complexity Analysis

Let S be the input string, $n = |S|$ be its length, LP be the longest path label in the suffix tree of S and $\mathbf{L}$ be the largest

list that stores the offset of each occurrence of a S-prefix p. In the worst case, $|LP| = \mathcal{O}(n)$ and $|\mathbf{L}| = \mathcal{O}(n)$. To see this, consider $S = aaaa\$$ for which $n = 4$, $|LP| = 3$ (since S-prefix aaa appears at positions 0 and 1) and $|\mathbf{L}| = 4$ (since S-prefix a appears 4 times).

Algorithm SubTreePrepare in each iteration of the while loop (Lines 8-24) retrieves *range* symbols for each entry of $\mathbf{L}$ and sorts them lexicographically. Thus, each iteration takes $|\mathbf{L}| \log |\mathbf{L}|$ time. Moreover, each iteration is performed $\frac{|LP|}{range}$ times. Therefore, the overall worst case complexity of SubTreePrepare is $|\mathbf{L}| \cdot \log |\mathbf{L}| \cdot \frac{|LP|}{range}$ which is $\mathcal{O}(n^2 \log n)$ time. Algorithm BuildSubTree generates one leaf node for each entry of $\mathbf{L}$. To this end, it accesses the stack up to $|LP|$ times. Therefore, its worst case complexity is $|\mathbf{L}| \cdot |LP|$ which is $\mathcal{O}(n^2)$.

However, in practice and in all application scenarios $\mathbf{L} \ll n$ and $|LP| \ll n$ hold. In fact it is reasonable to expect that $\mathbf{L}$ and $|LP|$ are orders of magnitude smaller than n. Thus, the overall expected complexity bound of ERa is much better than the worst case bound. This is also verified by the experimental evaluation, which demonstrated that ERa scales almost linearly to n.

4.4 Memory Allocation and Disk Access

Efficient allocation of the available memory is critical because, if more memory is available for the sub-tree, vertical partitioning will generate fewer virtual trees, hence the I/O cost will be lower. Let MTS be the maximum tree size and $f_p = |\mathbf{L}|$ be the frequency of the sub-tree $\mathcal{T}_p$ of S-prefix p. Recall from Section 4.1 that $|\mathbf{L}| \leq \mathcal{F}_M$. ERa divides the available memory into three parts (see Figure 6):

Retrieved data area. It contains the input buffer **BS** and the array **R** of next symbols. It also contains a small area (less than 1MB) for the trie that connects sub-trees.

Processing area. It contains data structures that are used during construction. These include arrays **I**, **L**, **P**, **A** and **B**. The size of all of these arrays is of factor $|\mathbf{L}|$. **L** together with **B** consume almost 40% of the available memory.

Suffix tree area. Its size (i.e., MTS) is roughly 60% of the total available memory.

The size of **BS** is relatively small and should be a multiple of the block size of the underlying I/O subsystem; in our environment 1MB was adequate. The size of **R** affects the range of symbols to be fetched in each scan (Line 5, Algorithm SubTreePrepare). A large **R** minimizes the number of string scans while a small **R** avoids unnecessary reads and frequent cache misses. These occur when algorithm reads *range* symbols from S but only few of them are needed to determine that it corresponds to a leaf and does not need further processing. The proper size of **R** mainly depends on the alphabet size, which determines the branching factor of the tree. Intuitively, to build suffix trees with a larger branching factor, we require more concurrent active areas and thus a larger size of **R**. In our experiments, we found that a good size for small alphabets (e.g, DNA data) is 32MB whereas for large alphabets (e.g., Protein data) it should be 256MB (Figure 8).

Observe that the processing and the suffix tree areas in Figure 6 overlap. SubTreePrepare uses part of the suffix tree area to store arrays **I**, **A** and **P**. Recall that the sub-tree is constructed in batch by Algorithm BuildSubTree

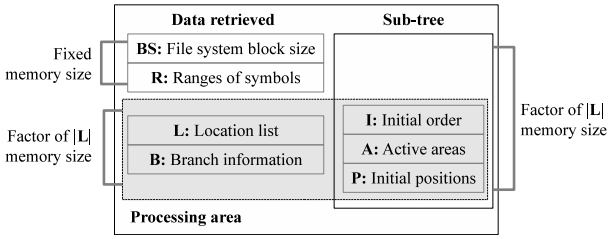


Figure 6: Allocation of the available memory. The processing and sub-tree area overlap

which only needs arrays **L** and **B**; therefore, **I**, **A** and **P** can be safely overwritten.

ERa implements dynamic memory management to reduce significantly the I/O cost. Recall that Algorithm SubTreePrepare scans the string $\frac{|LP|}{range}$ times. While the size of **R** is constant, the number of active areas in **L** are reduced after each iteration if new leaves are discovered; inactive areas do not need space in **R**. Let $|\mathbf{L}'| \leq |\mathbf{L}|$ be the number of **L** entries that belong to active areas at the current iteration. Line 9 in SubTreePrepare calculates the range of next symbols to prefetch as $range = \frac{|\mathbf{R}|}{|\mathbf{L}'|}$. In practice, after a few scans of S a lot of areas become inactive and *range* becomes large, leading to a dramatic improvement in I/O cost. The gain becomes more significant as the size of the input increases. Our experiments revealed that, for very long strings, the performance of the algorithm is doubled.

ERa also optimizes the disk access pattern. Previous methods (e.g., WaveFront) at each scan read the entire string in sequential order. The intuition is that *(a)* sequential order avoids the seek time, therefore it is roughly an order of magnitude faster than random I/Os in modern disks; and *(b)* since the probability of finding at least one required symbol within each disk block is high, only a few blocks will be unnecessarily fetched if the entire S is read. While *(b)* is true for the initial iterations of ERa, we observed that, as more leaves are discovered and areas become inactive, the probability of fetching a block that does not contain any required symbol increases significantly. For this reason, we implemented a simple heuristic: If a block (or a continuous range of blocks) is not expected to contain any necessary symbol, we skip these blocks by performing a random seek. Whether next block(s) contain at least one necessary symbol can be determined by the information in **I** and *range*. Note that, even though a random seek is performed, the seek time is expected to be very small because the next block is physically very close to the current disk head position. The experiments show a gain of up to 10%.

5. PARALLEL CONSTRUCTION

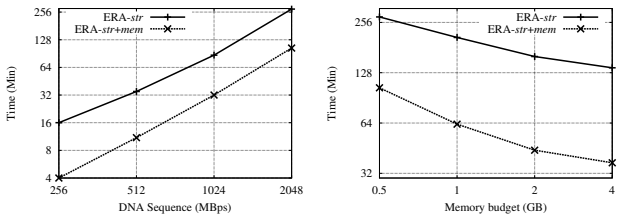
Indexing very long strings can use parallel computing resources and aggregated CPU power to achieve better performance. Most existing suffix tree construction algorithms (including the recent B^{2}ST) are not easily parallelizable because each thread processes a small portion of the string S and there is a costly phase that merges thread results. To the best of our knowledge, the most successful parallel construction algorithm is PWaveFront [6].

Horizontal partitioning of ERa is easily parallelizable because each process is independent and there is no merging phase (Section 4.2.2). We developed two parallel versions of

ERa: one for shared-memory and shared-disk systems (e.g., typical multicore desktops) and a second one for shared-nothing architectures (e.g., computer clusters or cloud computing infrastructure). We did not parallelize the vertical partitioning phase since its cost is low.

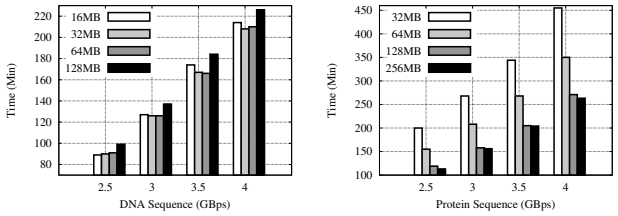
Shared-memory and shared-disk. This category contains multicore systems where cores share the main system's RAM and disk. A master thread at one of the cores generates groups of variable length prefixes and divides these groups equally among the available cores including itself. The main advantage of this architecture is that the input string is available to all cores. A significant drawback is the bottleneck at the memory bus and I/O subsystem when multiple cores attempt to access the string. Therefore, scalability is expected to be limited.

Shared-nothing architecture. In this architecture, each node has its own disk and memory; thus the aggregated I/O and memory bandwidth scale with the number of nodes. Again, a master node generates groups of variable length prefixes and divides them equally among the available nodes including itself. Since each node works independently, this architecture has the potential to scale-up very well. Note, however, that during initialization the input string should be transmitted to each node; this is the main drawback of this architecture. We expect that the problem can be minimized by using an appropriate parallel file system.



(a) Variable string size (b) Variable memory size

Figure 7: Serial execution time of horizontal partitioning methods; DNA dataset. (a) 512MB RAM; (b) $|S|$=2GBps



(a) DNA (b) Protein

Figure 8: Tuning the size of R: 32MB for DNA ($|\Sigma| = 4$); 256MB for Protein ($|\Sigma| = 20$)

6. EXPERIMENTAL EVALUATION

This section presents the performance evaluation for the serial and parallel versions of ERa. We compare our work against the two existing *out-of-core* approaches, B^{2}ST and WaveFront, and a *semi-disk-based* approach: Trellis. For B^{2}ST [2] and Trellis [13], we downloaded the serial implementation from the authors' sites. There is no parallel version and the existing implementations support only strings with 4 symbols. WaveFront was not available, so we implemented our own serial version following [7] and a parallel version following PWaveFront [6].

We used large real datasets: *(a)* The Human Genome[3] with size roughly 2.6GBps[4] and alphabet of 4 symbols; *(b)* DNA[5], containing 4GBps from an alphabet of 4 symbols, which is the concatenation of horse, zebra-fish and human DNA sequences; *(c)* the Protein[6] dataset containing 4GBps from an alphabet of 20 symbols, and *(d)* the English text from Wikipedia[7] containing 5G characters from an alphabet of 26 symbols.

6.1 Serial Version

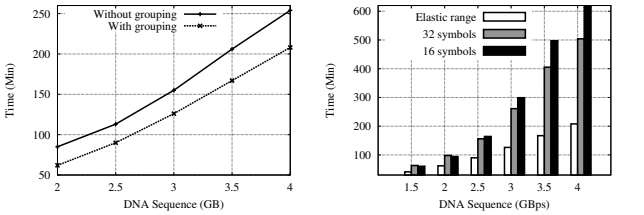
All serial methods were implemented in C, except Trellis that was implemented in C++, and compiled with gcc version 4.4.1 in Linux. The experiments were executed on a machine with two quad-core Intel CPUs at 2.67GHz and 24GB RAM. As the main focus is on *out-of-core* approaches, our experiments used a ratio of memory budget to input string size that is up to 1:5. We limited the available memory using `ulimit -v`, and turned off the virtual memory (`swapoff -a`).

Horizontal partitioning. In this experiment, we compare the two horizontal partitioning methods proposed in Section 4.2 for ERa. The 1st approach (ERa-*str*) uses Algorithms ComputeSuffixSubTree and BranchEdge that tune string access (Section 4.2.1), while the 2nd approach (ERa-*str+mem*) uses Algorithms SubTreePrepare and BuildSubTree that tune string and memory access (Section 4.2.2). We varied the size of the input string from 256MBps to 2048MBps (DNA dataset) while setting the memory size to 512MB, as shown in Figure 7(a). Moreover, Figure 7(b) compares the construction time for a 2GBps DNA sequence with memory size varying from 0.5 to 4GB. These experiments demonstrate that ERa-*str+mem* (Section 4.2.2) offers significant improvements.

ERa tuning. Here we tune the size of **R** (i.e., read-ahead buffer for next symbols) that significantly affects the performance of ERa (see also Section 4.4). Larger **R** means less scans of the string while smaller **R** avoids unnecessary reads. Since the size of **R** depends on the size of the alphabet, we seek the proper size of **R** for a small alphabet (DNA dataset of 4 symbols) and a large alphabet (Protein dataset of 20 symbols). To this end, we fix the memory to 1GB and generate inputs with 2.5 to 4GBps from the prefixes of DNA and Protein datasets. Figure 8(a) shows that 32MB is a good value for the DNA dataset, whereas Figure 8(b) shows that 256MB is appropriate for the Protein dataset, which has larger alphabet. The results for English were similar to Protein, since both datasets need 5 bits to encode each symbol; therefore, we used $|\mathbf{R}|$=256MB for English, too.

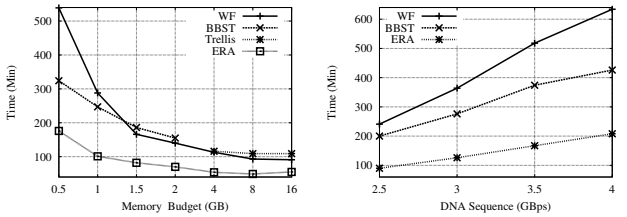
Unless otherwise mentioned, the following experiments use the disk seek optimization, described in Section 4.4. This optimization improved the performance of the serial version

[3] `http://webhome.cs.uvic.ca/~thomo/HG18.fasta.tar.gz`
[4] GBps: Giga Base pairs - equivalent to 10^9 symbols
[5] `http://www.ensembl.org/info/data/ftp/index.html`
[6] `http://www.uniprot.org/uniprot/?query=&format=*`
[7] `http://en.wikipedia.org/wiki/Wikipedia:Database_download`



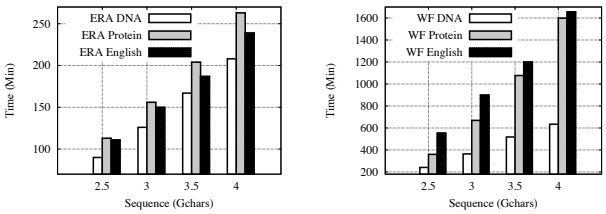
(a) Effect of virtual trees (b) Effect of elastic range

Figure 9: Effect of virtual trees and elastic range. DNA dataset; 1GB RAM; serial execution



(a) Variable memory; Human Genome (b) Variable string size; DNA; 1GB RAM

Figure 10: Serial execution time versus (a) available memory and (b) string size



(a) ERa (b) WaveFront

Figure 11: Serial execution time of (a) ERa and (b) WaveFront versus variant alphabets; 1GB RAM

by roughly 10% (see Section 6.2 for more details).

Vertical partitioning. Recall from Section 4.1 that vertical partitioning in ERa extends WaveFront by grouping the sub-trees into virtual trees to amortize the I/O cost. Figure 9(a) compares the effect of using the virtual trees versus no tree grouping, for the DNA dataset and 1GB RAM. Virtual trees achieve at least 23% better overall performance.

Elastic range. The next experiment shows the effect of the elastic range approach. Recall from Section 4.4 that as more areas become inactive, ERa uses the space that becomes available in **R** (whose size is constant) to prefetch more symbols for the active areas. Figure 9(b) compares the performance of elastic range against two alternatives that use static ranges of 16 and 32 prefetched symbols; elastic range is 46% to 240% faster and the gain increases for very long strings. Note that using a larger static range is not a good alternative to the elastic range. For example, 32 symbols is 22% faster than 16 symbols for string size equal to 4GBps, but it is 13% slower than 16 symbols for $|S| =$ 1.5GBps.

Comparison against WaveFront, B^{2}ST and Trellis. The following experiments compare ERa against WaveFront and B^{2}ST. We have allotted to all algorithms the same amount of memory. B^{2}ST allocates the memory to the input and output buffers and the intermediate data, such as suffix arrays. For WaveFront, the best setting according to [7] divides the memory equally between the processing space, the input buffers and the sub-tree. In contrast, ERa first allocates memory for **R** (according to Figure 8), 1MB for the input buffer and 3MB for the trie index. 60% of the remaining memory is allocated to the sub-tree and the rest is used for processing space (i.e., arrays **B** and **L**). **A**, **P** and **I** are located temporally in the area of the sub-tree, as discussed in Section 4.4. Because of the better allocation, ERa can construct larger sub-trees than WaveFront using the same amount of memory.

Figure 10(a) compares the construction time for the Human Genome dataset with memory size ranging from 0.5 to 16GB. ERa is consistently twice as fast compared to the best competitor, where string size is larger than the memory budget (out-of-core construction). It is worth noting that, while WaveFront is slightly faster than B^{2}ST for large memory size, it is dramatically slower when the memory is limited. Note that, the available implementation of B^{2}ST does not support large memory; this is why B^{2}ST plot stops at 2GB.

We also compared the performance of ERa against the performance of WaveFront and Trellis using large memory budget. Note that Trellis needs to accomodate the entire input string in memory. Since the human genome cannot fit in 2GB memory, the plots for Trellis start at 4GB. Recall that both ERa and WaveFront access the string sequentially from disk during construction. As shown in Figure 10(a), both ERa and WaveFront outperform Trellis. Although Trellis does not pay the I/O cost of accessing the string, it has to access in random fashion the large sub-trees (total size is roughly 26 times larger than the input string) from the disk during the merging phase. Our results agree with those from [2, 7].

Furthermore, we varied the size of the input string from 2.5 to 4GBPs (DNA dataset) while setting the memory size to 1GB. The total execution time is shown in Figure 10(b). ERa is at least twice as fast as its competitors. The performance gap from WaveFront is increasing for longer strings.

Finally, we evaluated the performance of ERa and WaveFront for different alphabet sizes. Figure 11 shows the results of both methods for DNA ($|\Sigma| = 4$), Protein ($|\Sigma| = 20$), and English ($|\Sigma| = 26$) datasets. For ERa, since DNA has only 4 symbols, each symbol is encoded in 2 bits, in contrast to Protein and English that need 5 bits per symbol. Therefore, a larger part of the DNA string can fit in the memory. Also fewer symbols mean smaller branch factor, therefore less CPU cost. For these reasons using ERa, the processing of DNA is around 20% times faster than Protein and English. Moreover, the longest sequence that is repeated in the English dataset is shorter than the longest repeated sequence in the Protein dataset. The longest repeated sequence affects the depth of the tree. Therefore, ERa indexes the English dataset faster than the Protein.

WaveFront inserts the suffixes (leaf nodes) ordered as they appear in the string from left to right. Since the leaves of the suffix tree are sorted lexicographically, nodes that are logically connected, are not physically nearby in the suffix tree built by WaveFront. The more symbols, the larger

the branch factor, which leads to more random memory accesses. Consequently, WaveFront spends a lot of time navigating the tree in order to insert a new leaf node. In contrast, since ERa sorts first the leaves lexicographically, it is not affected significantly by the branch factor.

6.2 Parallel Version

We developed parallel versions of ERa and WaveFront using MPI[8]. For WaveFront we followed [6]. There is no existing parallel version of B^2ST. Moreover, such an implementation would probably be inefficient because of the costly merging phase at the end of the construction. We focus on two architectures: *(a)* shared-memory and shared-disk, which represents the modern multi-core desktop systems; and *(b)* shared-nothing architecture, such as computer clusters or cloud computing environments.

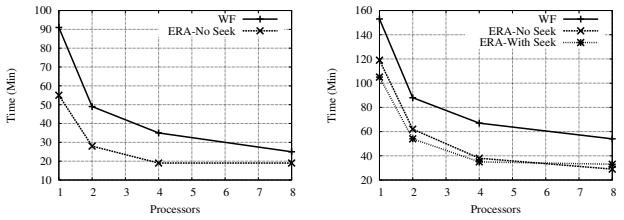
We use two metrics common to high performance computing: *(a) Strong scalability* (also known as speed-up): shows the performance for constant input size and increasing number of processors; and *(b) Weak scalability* (also known as scale-up): shows the performance when the ratio of the input size to the number of processors remains constant (e.g., when the length of the string doubles, the number of processors also doubles).

Shared-memory and -disk architecture. The scalability of the shared-memory and shared-disk architecture suffers from the interference at the memory bus and the I/O system. Here, we investigate this interference. We used the same machine as the previous section (i.e., Linux with two quad-core Intel CPUs at 2.67GHz and 24GB RAM), but we employed 1 to 8 cores and used 16GB RAM. The memory is divided equally among cores (1 core with 16GB, or 2 cores with 8GB RAM each, etc.).

For the next experiment we turn off the disk seek optimization (Section 4.4); the reason will become evident in the next paragraph. Figure 12(a) illustrates the execution time of ERa-No Seek and WaveFront for the Human Genome (i.e., strong scalability) with 16GB RAM. With 4 cores ERa-No Seek indexes the entire Human Genome in 19 minutes. ERa-No Seek scales well up to 4 cores (4GB RAM per core). In fact ERa is at least 1.5 times faster than WaveFront for up to 4 cores. However, ERa does not scale well to 8 cores. We believe the reason is that each of the 8 cores accesses only 2GB RAM, meaning that a smaller part of the tree is processed in one iteration. Therefore, each core requests data more frequently, leading to bottlenecks due to interference. In contrast, we believe WaveFront scales better to 8 cores (although in absolute time is still worse than ERa), because the CPU cost of the algorithm is higher. Therefore, it requests less frequently data from the disk, causing less interference.

To confirm our justification we run the same experiment with the larger DNA dataset (i.e., 4GBps). Now each core has more workload. Figure 12(b) illustrates that ERa-No Seek scales better to 8 cores. The same graph also shows the performance of ERa with the disk seek optimization turned on. With few cores, ERa-With Seek performs better, because it skips the parts of the input string that do not contain relevant input (see Section 4.4). For 8 cores, however, ERa-No Seek becomes better. This is due to the fact that each of the 8 cores work asynchronously on different parts of the string. The disk seek optimization is applied independently by each core and causes the disk head to swing back and forth, creating significant delays.

[8] http://www.mcs.anl.gov/research/projects/mpi



(a) Human Genome (b) 4GBps DNA

Figure 12: Shared-memory, shared-disk; strong scalability; 16GB RAM divided equally among cores

CPU	WaveFront (min)	ERa (min)	Gain	ERa speedup	ERa all speedup
1	285.2	93.4	305%	–	–
2	142.2	46.4	306%	1	0.94
4	71.2	23.4	304%	0.99	0.87
8	36.2	12.4	292%	0.94	0.73
16	19.2	7.4	259%	0.78	0.53

Table 3: Shared-nothing; strong scalability; human genome; 1GB RAM per CPU. The last column shows the speedup for the entire process. The other columns exclude the string transfer (2.3min) and the vertical partitioning phase (1.6min)

Shared-nothing architecture. For the shared-nothing architecture experiments, we used a Linux cluster consisting of 16 machines connected through a switch, each with one dual-core Intel CPU at 3.33GHz and 8GB RAM. In each machine we used only one core and limited the memory to 1GB; therefore, if all 16 nodes are used, the total memory is 16GB. Note that the results in this paragraph are not comparable with those in the Share-memory and disk paragraph for two reasons: *(a)* the cluster machines have faster individual CPUs and *(b)* the total memory of the cluster varies from 1GB to 16GB depending on the number of machines used, whereas in the previous section the memory was fixed to 16GB irrespectively of the number of cores.

Table 3 shows the strong scalability results for the Human Genome. ERa is 3 times faster than WaveFront. Also the speed-up (refer to column titled ERa speedup) is very close to the theoretical optimal (i.e., 1.0), indicating very good load balancing. Note that all, but the last, columns in the table show only the time for tree construction. In the measurements, we have not included: *(i)* The time for the initial transfer of the input string to all nodes (roughly 2.3min). The bottleneck of string transfer is the slow switch; the performance can be improved with better network equipment that supports broadcast. *(b)* The vertical partitioning phase that takes around 1.6min since that phase has not been parallelized. The last column (titled ERa-all) shows the speedup considering these overheads; the speed-up is still very good, although not as close to the optimal. If everything is added, ERa indexes the entire Human Genome in roughly 11.3 minutes on a cluster with 16 commodity machines.

Since more memory was available in our machines, we run

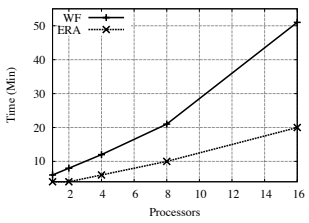


Figure 13: Shared-nothing; weak scalability; DNA dataset; size varies from 256MBps to 4096MBps

an experiment using 4GB per node. ERa indexed the Human Genome in 8.3 minutes. To the best of our knowledge in absolute time this is the fastest measurement reported so far. Note that we achieved this speed on a cluster whose total cost is roughly US$ 20,000 (February, 2011), low enough to be within the reach of many individuals.

The last experiment investigates the weak scalability (recall that in weak scalability the ratio between the size of the input S and the number of nodes is constant). We used the DNA dataset and varied the size from 256MBps for 1 node to 4096MBps for 16 nodes, with 1GB memory per node. According to the definition of weak scalability, in the optimal case the construction time should remain constant. In our case, since the size of S increases proportionally to the number of nodes, the number of sub-trees to be constructed by each node is indeed constant. However, the average construction time of each sub-tree increases proportionally to $|S|$, because each node still needs to scan S the same number of times. Therefore, optimal weak scalability cannot be achieved. Figure 13 illustrates that the construction time indeed increases linearly to the number of processors for ERa and WaveFront (note that the overheads for the string transfer is excluded). However, the rate of increase of ERa is much smaller than that of WaveFront. Therefore, as the string size increases, the performance gap widens; for example, when the string size is 4096MBps, ERa is 2.5 times faster than WaveFront. This is an important advantage of ERa, since in practice strings are expected to be very long.

7. CONCLUSIONS

Suffix trees are essential for many practical applications that include bioinformatics, processing of financial data (e.g., time series of stock market data), document clustering, etc. The volume of such data increases rapidly; therefore it is essential to have fast suffix tree construction methods. In this paper we proposed ERa, a method that supports very long strings, large alphabets, works efficiently even if memory is very limited and is easily parallelizable. Extensive experimental evaluation with very large real datasets revealed that our method is much more efficient than existing ones in terms of speed and computational resources. ERa indexes the entire human genome in 19 minutes on an ordinary 8-core desktop computer with 16GB RAM; and in 8.3min on a low-end cluster with 16 commodity computers with 4GB RAM each. To the best of our knowledge the fastest existing method (i.e., PWaveFront) needs 15min on an IBM BlueGene/L supercomputer with 1024 CPUs and 512GB RAM.

This work is part of a large project that aims to develop an engine for storing and processing of massive strings. We are currently working on scaling our method to thousands of CPUs. We are also focusing on the parallel processing of various types of queries using the suffix tree.

8. REFERENCES

[1] A. Amir, G. M. Landau, M. Lewenstein, and D. Sokol. Dynamic text and static pattern matching. *ACM Transactions on Algorithms*, 3, Issue 2, Article 19, 2007.

[2] M. Barsky, U. Stege, A. Thomo, and C. Upton. Suffix trees for very large genomic sequences. In *Proc. of ACM CIKM*, pages 1417–1420, 2009.

[3] C. Charras and T. Lecroq. *Handbook of Exact String Matching Algorithms*. King's College London Publications, 2004.

[4] H. Chim and X. Deng. A new suffix tree similarity measure for document clustering. In *Proc. of ACM WWW*, pages 121–130, 2007.

[5] P. Ferragina, R. Giancarlo, G. Manzini, and M. Sciortino. Boosting textual compression in optimal linear time. *Journal of ACM*, 52:688–713, 2005.

[6] A. Ghoting and K. Makarychev. Indexing genomic sequences on the IBM Blue Gene. In *Proc. of Conf. on High Performance Computing Networking, Storage and Analysis (SC)*, pages 1–11, 2009.

[7] A. Ghoting and K. Makarychev. Serial and parallel methods for I/O efficient suffix tree construction. In *Proc. of ACM SIGMOD*, pages 827–840, 2009.

[8] D. Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.

[9] T. Hey, S. Tansley, and K. Tolle, editors. *The Fourth Paradigm: Data-Intensive Scientific Discovery*. Microsoft Research, 2009.

[10] E. Hunt, M. P. Atkinson, and R. W. Irving. Database indexing for large DNA and protein sequence collections. *The VLDB Journal*, 11:256–271, 2002.

[11] T. W. Lam, R. Li, A. Tam, S. C. K. Wong, E. Wu, and S.-M. Yiu. High throughput short read alignment via bi-directional BWT. In *BIBM*, pages 31–36, 2009.

[12] E. M. McCreight. A space-economical suffix tree construction algorithm. *Journal of ACM*, 23:262–272, 1976.

[13] B. Phoophakdee and M. J. Zaki. Genome-scale disk-based suffix tree indexing. In *Proc. of ACM SIGMOD*, pages 833–844, 2007.

[14] S. J. Puglisi, W. F. Smyth, and A. H. Turpin. A taxonomy of suffix array construction algorithms. *ACM Computing Surveys*, 39, 2007.

[15] F. Rasheed, M. Alshalalfa, and R. Alhajj. Efficient periodicity mining in time series databases using suffix trees. *IEEE TKDE*, 23:79–94, 2011.

[16] J. Shieh and E. J. Keogh. *i*SAX: disk-aware mining and indexing of massive time series datasets. *Data Min. Knowl. Discov.*, 19(1):24–57, 2009.

[17] S. Tata, R. A. Hankins, and J. M. Patel. Practical suffix tree construction. In *Proc. of VLDB*, pages 36–47, 2004.

[18] Y. Tian, S. Tata, R. A. Hankins, and J. M. Patel. Practical methods for constructing suffix trees. *The VLDB Journal*, 14(3):281–299, 2005.

[19] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

Hermes^{sem}: a Semantic-aware Framework for the Management and Analysis of our LifeSteps

Nikos Pelekis
Dept. of Statistics & Insurance Sc.
University of Piraeus
Piraeus, Greece
npelekis@unipi.gr

Stylianos Sideridis
Dept. of Informatics
University of Piraeus
Piraeus, Greece
siderste@yahoo.gr

Yannis Theodoridis
Dept. of Informatics
University of Piraeus
Piraeus, Greece
ytheod@unipi.gr

***Abstract* — The explosion of available positioning information associated with the inferred or user-declared semantics of the respective locations, already contributes in what is called the big data era, posing new challenges to the mobility data management and mining research community. In this paper, motivated by a series of challenges set in [11], we present a unified framework for the management and the analysis of our LifeSteps, i.e. data objects that include both (raw) trajectories and their semantic counterpart. In particular, we provide solutions for developing real-world semantic-aware Moving Object Database (MOD) and Trajectory Data Warehouse (TDW) systems and we devise respective query processing algorithms. Our experimental study on synthetic data including synchronized raw (i.e., GPS log) and semantic (i.e., diaries) information, verifies the effectiveness and efficiency of the proposed framework.**

Keywords — mobility data warehouses, semantic trajectories, pattern queries

I. INTRODUCTION

Ubiquitous positioning devices enable the generation of huge volumes of location information on a continuous basis. The trend in modern *Location-based Services* (LBS) and *Location-based Social Networking* (LBSN) applications that make use of such data, is not only to focus on raw movement information collected by those devices, but also to target at the behavioral rationale and motivation of movement, in order to provide added value and enriched services.

A semantically annotated trajectory, in short *semantic trajectory*, is an alternative representation of the (raw) motion path of a moving object as this is logged by the positioning device, where the motion is represented as a sequence of semantically meaningful episodes, either *stops* (e.g. at home, at office, for shopping) or *moves* (walking, driving, etc.), which results in detecting homogenous fractions of movement [9]. Extracting and managing semantics from (raw) trajectory data is a promising channel that leads to significant storage savings. Of course, as already declared in [11], it is not only a matter of the database size; maintaining semantic (i.e. in our case textual only) information turns out to be quite useful in terms of content-related movement analyses. Such semantic-aware mobility abstractions enable applications to better understand and exploit on human mobility: for instance, identify those locations where some activity (work, leisure, relax, etc.) takes place, infer how long does it take to get from one place of interest (POI) to another (e.g. from home to office) using a specific transportation means, conclude about the frequency of an individual's outdoor activities, calculate indices related to environmentally friendly or sustainable mobility. Given such analytics, future semantic-aware LBS/LBSN applications can be established, including location sharing and ranking, recommendation according to travel and socio-demographic similarity (e.g. for dynamic ridesharing purposes), etc.

Although conceptually strong, such a representation of mobility lacks a robust DBMS support in order to realize the above model and give life to novel applications and services based on this. This implies a unified framework that will enable efficient and effective storage, indexing mechanisms of such heterogeneous information, as well as extended query languages to support novel types of queries and advanced methods for multidimensional analysis. In this paper we present such a unified framework. More specifically, the merits and contributions of our proposal are summarized below:

- Following the successful MOD engine paradigm, we design a datatype system and its associated query language, coined Hermes^{sem}, for the representation and management of Semantic Mobility Databases (SMD) engine into an extensible DBMS architecture. Furthermore, we present a design for modeling Semantic Mobility Data Cubes (SMDC).
- We propose efficient access methods for semantic trajectories, called SemTB-tree and Sem3DR-tree (extensions of the well-known TB-tree [14] and 3DR-tree [16] designed for (raw) trajectory data), for the hybrid indexing of both the spatio-temporal and the semantic (i.e. textual) component.
- Using the proposed indices, we develop efficient query processing algorithms to support a useful query type, called *spatial-temporal-textual pattern* (ST^{2}P) query, at the SMD level as well as algorithms for efficiently feeding SMDC from SMD.

Section II presents related work. Section III provides background information. Sections IV and V present the core of the paper, namely the realization of the proposed SMD and SMDC, respectively. Section VI presents our experimental study and Section VII concludes this paper.

978-1-4673-8273-1/15/$31.00 ©2015 IEEE

II. Related Work

Modeling, management, and knowledge discovery aspects on (raw) trajectory data have been exhaustively researched in the past two decades [13], including plenty of algorithms and systems, spreading from data management [4] to data mining [2]. On the other hand, semantic mobility data management is a relatively new entry in the research agenda. Models for semantic trajectories include [9][11], while techniques for extracting semantic trajectories from raw ones have been also proposed recently [19]. In [13], the interested reader may find a survey of relevant models and techniques.

According to the state-of-the-art model [9], a semantic trajectory is defined as a sequence of episodes, labeled either as 'Stops' or 'Moves', each associated with appropriate meta-data (tags). Technically, Stops are places (points or regions) where the object remains "static" and Moves are the parts of the object's trajectory in between two Stops, i.e., where the object is "moving". This model was extended in [11] in order to enable the management of such data to extensible database architectures, as well as to support their modeling and analysis at various scales and/or spatio-temporal granularities.

Traditionally, aggregated information from DBs is stored in a Data Warehouse (DW), in the form of data cubes [3]. Data cubes are views of a DW, used for multi-dimensional analysis, the so-called On-Line Analytical Processing (OLAP). The data cube paradigm has been extended to support spatial [5] and (raw) trajectory DWs [8][6][7], involving spatial, temporal, and thematic dimensions as well as spatial, spatio-temporal, and numerical measures. A DW model for semantically-enriched mobility data, called Mob-Warehouse, was proposed in [17] to enrich trajectory data with domain knowledge by following the so-called 5W1H model (Who, Where, When, What, Why, How). In [1], another semantic model tailored to open, linked data was proposed. In [11], a graph-based representation of mobility-aware data cubes was proposed.

Apart from the inclusion of semantics at the conceptual level, one of the challenges raised in [11] was the necessity to efficiently support the management and analysis of such data. Thus, [11] sketched the big picture of an envisioned three-tier framework, as follows: (i) at the bottom-layer, a traditional MOD lies, being in charge of the raw mobility data and supported by well-known access methods and query functionality [18][10]; (ii) at the middle-layer, it is the SMD that provides novel datatypes, indexing methods, and operators extending MOD query languages for querying and analyzing mobility data from a semantic perspective; (iii) at the top layer, the application interface provides users with querying and analysis functionality on either MOD or SMD, via simple SQL.

This paper presents a data management and analysis framework, which is a realization of that vision. To the best of our knowledge, this approach is novel and provides a valuable tool in the hands of the researchers in the field.

III. Background

Formally, the (*raw*) *trajectory* τ of a moving object is defined as a tuple (*o-id*, *traj-id*, *T*), where *o-id* (*traj-id*) is the identifier of the moving object (the specific trajectory of the moving object, respectively) and *T* is a 3D polyline consisting of a sequence of N+1 pairs (p_i, t_i), $0 \le i \le N$, where p_i is a 2D point (x_i, y_i) in the plane and t_i is a timestamp, assuming linear interpolation between two consecutive pairs (p_i, t_i) and (p_{i+1}, t_{i+1}). In turn, a (raw) trajectory defined as above can be partitioned into a sequence of (raw) sub-trajectories; formally, a (*raw*) *sub-trajectory* τ*'* of a (raw) trajectory τ is defined as a tuple (*o-id*, *traj-id*, *subtraj-id*, *T'*), where *o-id* (*traj-id*, *subtraj-id*) is the identifier of the moving object (the specific trajectory and sub-trajectory of the moving object, respectively) and *T'* is the portion of *T* between two timestamps, t_i and t_j, $t_i < t_j$. We are now able to define their semantic variants:

Definition 1 (LifeStep): a LifeStep *ls* corresponds to a sub-trajectory τ' and is defined as a tuple (*LifeStepID, LifeStepFlag, MBB, tags, T-link*), where *LifeStepID* is the identifier of the LifeStep, *LifeStepFlag* is a flag taking values from set {*'Move', 'Stop'*}, *MBB* is a tuple (*MBR*, t_{start}, t_{end}) corresponding to the 3D approximation of τ', with *MBR* being the 2D enclosing rectangle of the spatial projection of τ' in 2D plane and [t_{start}, t_{end}] being the 1D temporal projection of τ' in 1D timeline, *tags* is a set of keywords, describing the corresponding activities and semantic annotations related to this portion of movement, *T-link* is a link to τ'.

Definition 2 (Mobility Timeline): a mobility timeline τ^{sem} of a moving object is defined as a triple (*o-id*, *timeline-id*, T_{LS}), where *o-id* (*timeline-id*) is the identifier of the moving object (the mobility timeline of the moving object, respectively) and T_{LS} is a sequence of LifeSteps belonging to the same trajectory τ and being successive in time, i.e., $ls_i[t_{end}] = ls_{i+1}[t_{start}]$.

Looking at the above definitions, it is clear that the content of a MOD (raw trajectories) and the respective of a SMD (mobility timelines) do not share much in common. This means that existing MOD engines, such as Hermes [18] and Secondo [10], cannot be used as-is in order to handle a SMD. For instance, queries like *"Find people who cross the city center on their way from office back to home"* or *"Find people who drive more than 20 km on their way from home to office"* or *"Find people who spend more than 1 hour daily for bring-get activities of their children at schools"* cannot be easily supported by MOD engines since they make strong use of semantics. Nevertheless, they are typical examples of "what-if" analysis in the transportation domain and, as such, they ask for efficient support in DBMS.

The above discussion results in a categorization of queries over MOD/SMD in at least three types [11]. Q1-type queries like range, nearest-neighbor, enter, cross, etc. over raw trajectories have been extensively studied in the MOD literature. On the other hand, Q2-type queries (that involve semantic trajectories only) and Q3-type queries (that involve both raw and semantic trajectories) are innovative and they cannot be considered as straightforward variations of Q1-type ones. Moreover, Q4-type queries include advanced operations, such as pattern queries, over SMD. A typical example is the following: *"Find people who follow the home–office–*–gym pattern"*, where we search for LifeSteps including the specific sequence (with the wildcard '*' denoting 'any').

As a step forward, we introduce the notion of *Semantic Mobility Data Cube* (SMDC), where aggregated data should

not only expose interesting measures with respect to the chosen dimensions via a relational format, but they should also encapsulate the spatial topology and its intrinsic relationships. To succeed this ambitious goal, we exploit on the so-called *Semantic Mobility Network* (SMN) [11], a dynamic graph representation of the semantic mobility timelines. A nice interesting characteristic of a SMN, which we consider as a novel characteristic of our approach in the mobility management and analysis domain, is that this graph-based design is data-driven and unifies all the mobility-related dimensions (space, time, and semantics). Below, we provide formal definitions of a SMN and an aggregate SMN, while the reader is referred to [11] for several examples that illustrate the merits of this representation:

Definition 3 (Semantic Mobility Network - SMN): A semantic mobility network N, is a graph denoted by $N = (V, E, M)$, where V is a set of vertices, $E \subseteq V \times V$ is a set of edges and $M = \{M_1, M_2, \ldots, M_n\}$ is a set of measures applicable to vertices and edges, i.e. $\forall v \in V$ and $\forall e \in E$, there is a tuple $M(v)$ of v and $M(e)$ of e respectively, denoted as $M(v) = (M_1(v), M_2(v), \ldots, M_n(v))$ and $M(e) = (M_1(e), M_2(e), \ldots, M_n(e))$, where $M_i(v)$ and $M_i(e)$ is the value of v and e on i-th measure, $1 \leq i \leq n$. The set V of vertices corresponds to the union of all distinct LifeSteps that are of 'Stop' type, of all mobility timelines τ^{sem}, while the set E of edges corresponds to the union of the 'Move' type LifeSteps. The set M of measures is a set of scalars quantifying properties of vertices and edges.

Definition 4 (Aggregate Semantic Mobility Network - ASMN): Given a semantic mobility network N, a set of dimensions $D = \{D_1, D_2, \ldots, D_m\}$ with their corresponding hierarchies, an aggregation $D^a = \{D^a_1, D^a_2, \ldots, D^a_m\}$ along these hierarchies, with $D^a(v) = (D^a_1(v), D^a_2(v), \ldots, D^a_m(v))$ denoting a tuple of values $D^a_j(v)$ of v on j-th dimension, $1 \leq j \leq m$ ($D^a(e)$ is defined similarly), upon which measure M_i can be aggregated, an aggregate semantic mobility network with respect to D^a is a semantic mobility network $N^a = (V^a, E^a, M^a)$, where:

i. V^a is the set of aggregate vertices $v^a \in V^a$, each of which is constructed by a unification process $U_V([v])$ upon a nonempty equivalence class $[v]$ of V, where $[v] = \{v \mid D^a_j(v) = D^a_j(u), v, u \in V, j = 1, \ldots, m\}$,

ii. E^a is the set of aggregate edges $e^a \in E^a$, each of which is constructed by a unification process $U_E(E(v^a, u^a))$ upon a non-empty edge set (i.e. 'Move' LifeSteps), where $E(v^a, u^a) = \{(v, u) \mid v \in [v], u \in [u], (v, u) \in E\}$, and

iii. M^a is the set of aggregate measures, each of which is computed by applying an aggregate function $A(\cdot)$ on the measure values $M_i(v)$, $v \in [v]$ and $M_i(e)$, $e \in E(v^a, u^a)$, respectively, $1 \leq i \leq n$.

Note that the set of measures may be different for vertices and for edges (e.g. average stop vs. move duration) depending on the application, while A(•) aggregate function may differ from measure to measure (e.g. count(•) or average(•)). Of course, spatial or spatio-temporal measures may require more sophisticated aggregate functions (e.g. the 'mean' LifeStep), which are out of the scope of the current work. The key issue for the aggregate function is to avoid being holistic, because in that case, super-aggregates cannot be computed from sub-aggregates, even if we employ auxiliary measures [3]. Note also that the unification process U_V (U_E) operates on the spatio-temporal and semantic properties of the 'Stop' ('Move') LifeSteps, respectively.

Definition 5 (Semantic Mobility Data Cube - SMDC): Given a semantic mobility network $N = (V, E, M)$ and a set of dimensions $D = \{D_1, D_2, \ldots, D_m\}$ with their corresponding hierarchies, the semantic mobility cube is the lattice of the aggregate semantic mobility networks produced by all possible aggregations in D.

The above definition implies that given a SMN N, each aggregation D^a of D, called a *semantic mobility cuboid*, is itself a graph. To the best of our knowledge, this modeling approach is novel for the mobility domain, since it has been studied only for non-spatial, vertex-specific multidimensional networks of traditional datatypes [20].

It is interesting to note a few example operations that can be applied to this framework for progressive analysis purposes. For instance, after we *"extract the aggregate semantic mobility network A of user Bob during a period of time"*, we could *"restrict it at a particular region of interest"*, by using a range query. Assuming this network is at the base cuboid level, we could then join it with *"the aggregated (over a period of time) network B of a set of users (e.g. Bob's friends and co-workers according to a social network)"*, as such ASMN B resides at a higher level in the lattice than ASMN A. The join result (which obviously is a novel cross-SMN operation) could identify Bob's mobility network wherein he performs similar activities at similar places following similar routes with his friends.

Given the above discussion, and following the typology of query types proposed in [11], Q5- and Q6-type queries involve SMN, perhaps with the aid of cross-over operations that link MOD and SMD. Clearly, both types of queries are innovative and have not been addressed in the related work on semantic trajectories [9].

IV. Modeling Mobility Timelines and Semantic Mobility Networks

This section presents our proposal for modeling the concepts defined in Section III, namely Mobility Timelines (Section IV.A) as well as SMN and SMDC (Section IV.B).

A. Modeling Mobility Timelines

Towards the realization of the concepts of LifeSteps and Mobility Timelines presented in Section III, we follow the object-relational (OR) approach and extend the type system of Hermes MOD engine [10] and its associated query language.

In detail, we follow the Abstract Data Type (ADT) paradigm and define the so-called LifeStep and Timeline datatypes (the former being subtype of the latter) that support Definitions 1 and 2, respectively (see Section III). Upon these datatypes, we register a rich palette of object methods. A few indicative examples appear in Table 1. More advanced methods include confinement of a Timeline in spatial-temporal-textual cube, calculating the distance between two LifeSteps or between two Timelines in each spatial-temporal-textual dimension, and more.

TABLE I. METHODS OVER LIFESTEP AND TIMELINE DATATYPES

Method name	Description
sem_episode.duration return number	Returns the lifespan of a LifeStep
sem_trajectory.getMBB return MBB	Returns the MBB of a Timeline
sem_trajectory.num_of_stops return integer	Returns the number of STOP LifeSteps contained in a Timeline
sem_trajectory.num_of_moves return integer	Returns the number of MOVE LifeSteps contained in a Timeline
sem_trajectory.episodes_with (tag varchar2) return sem_episode_tab	Given a string 'tag' that is a concatenation of tags, returns the set of LifeSteps of a Timeline that match with the content of 'tag'

The resulted query language, i.e., SQL extended with methods and operators over the new datatypes, is appropriate for such complex (spatial-temporal-textual) objects, which can actually be considered as synchronized GPS traces along with diary information. Although interesting, due to space limitations, the presentation of the query language is not included in this paper. Nevertheless, in order to provide the general idea we present three examples below:

```
Q1) Select episode from SMD
    where SDO_ANYINTERACT(t.geom,
          sdo_geometry(3008, 2100, null,
          sdo_elem_info_array(1, 1007, 3),
          sdo_ordinate_array(450000, 4210000,
          timestamp(2013,11,10,7,0,0).toSpatial,
          480000, 4230000,
          timestamp(2013,11,10,17,0,0).toSpatial)))
          = 'TRUE'
    and defining_tag = 'STOP'
    and episode_tag = 'UNIVERSITY'
    and activity_tag = 'STUDYING';

Q2) Select std.range_episodes(sem_episode(
          'STOP', 'UNIVERSITY', 'STUDYING',
          sem_mbb(sdo_geometry(2003, 2100, null,
          sdo_elem_info_array(1, 1003, 3),
          sdo_ordinate_array(450000, 4210000,
          480000, 4230000)),
          timeperiod(timestamp(2013,11,10,7,23,18),
          timestamp(2013,11,10,9,24,12))),null),
           SemTB-tree) from SMD;

Q3) Select std.patterns(sem_episode_tab(
          sem_episode('STOP', 'HOME', 'RELAXING',
          sem_mbb(sem_st_point(473600, 4200700,
          timestamp(2013,11,10,3,0,0)),
          sem_st_point(473700, 4200800,
          timestamp(2013,11,10,6,0,0))), null),
          sem_episode('STOP', 'GYM', 'SPORTING',
          null, null)), varchar_ntab(null, '*'), 'SMD');
```

Q1 returns all LifeSteps of the SMD that match (in all spatial, temporal and textual dimensions) a given LifeStep, namely a Stop at a university for studying, located in MBR (450000, 4210000, 480000, 4230000), between 7am and 5pm on Nov. 10th, 2013. Q1 exploits on the built-in R-tree (hence, it uses Sem3DR-tree). The same result, this time using the SemTB-tree index, is achieved by delivering Q2.

On the other hand, Q3 implements the ST^2P query (to be formally defined in the section that follows); in particular, it searches for Timelines in SMD including LifeSteps that follow a given pattern (starting from home/relaxing and ending at gym/sporting, where the starting LifeStep is spatially and temporally constrained within an MBR and period, respectively).

B. Designing Efficient SMDC over SMN for OLAP Purposes

Defining aggregations over mobility networks is a problem related to the graph cube problem [20]. Recalling the idea of trajectory DW (TDW) [6], we propose a SMN to be a constellation scheme consisting of five dimensions (namely, space, time, user-profile, STOP-type-activity, MOVE-type-activity) and two fact tables (namely, STOPS-fact and MOVES-fact). Intuitively, this approach allows the support of several kinds of analysis:

- STOPS-Fact-table: who made a stop? when and where? what was the activity during the stop?
- MOVES-Fact-table: who made a movement? when and from/to where? How was the movement made and what was the activity during the movement?

Fig. 1 provides a relational scheme of an effective modeling of SMN, where the measures of the fact tables correspond to the weight vectors of a SMN. These measures are similar with the ones used for TDW [6], but here they are trivially customized for 'Stops' and 'Moves'. Also note that these measures are subject to the *distinct count problem* when computing super-aggregates by sub-aggregates. This issue is tackled with a similar manner as in [6].

The adoption of the relational model is in order to be fully compatible with our approach of extending a real MOD with semantic functionality. This actually allows us to use the extended query language when feeding the SMDC during the ETL process. Deriving a SMN from a SMD is a computational challenging task. For instance, the SMN is built at a very refined spatial granularity (namely, at the level of the POIs and not at the region level, as in the case of TDW [6]). In the following section, we provide alternative approaches for feeding a SMDC.

V. INDEXING AND QUERY PROCESSING OVER SMD AND SMDC

The realization of the above-described framework raises a natural question: how would a SMD be developed to provide efficiency in storage and querying, as well as effectiveness in analytics? In this direction, we first propose access methods for indexing mobility Timelines (Section IV.A). Then, we provide algorithms for the efficient processing of the so-called spatial-temporal-textual pattern query (Section IV.B) and the efficient feeding of SMDC (Section IV.C).

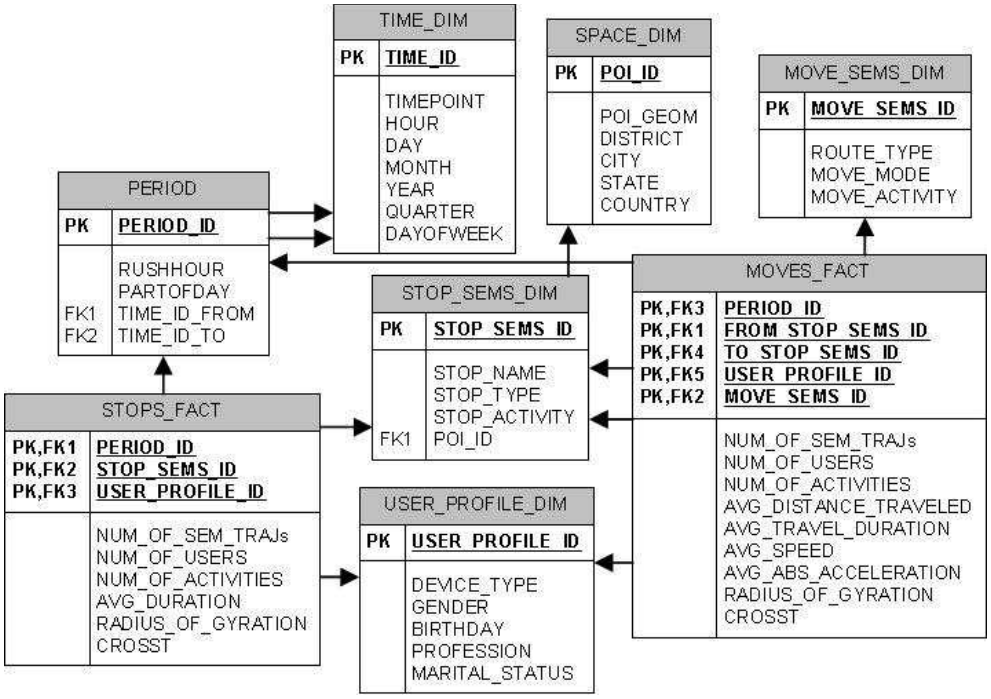


Fig. 1. A data cube for SMN; a constellation scheme consisting of two fact- and five dimensional- tables.

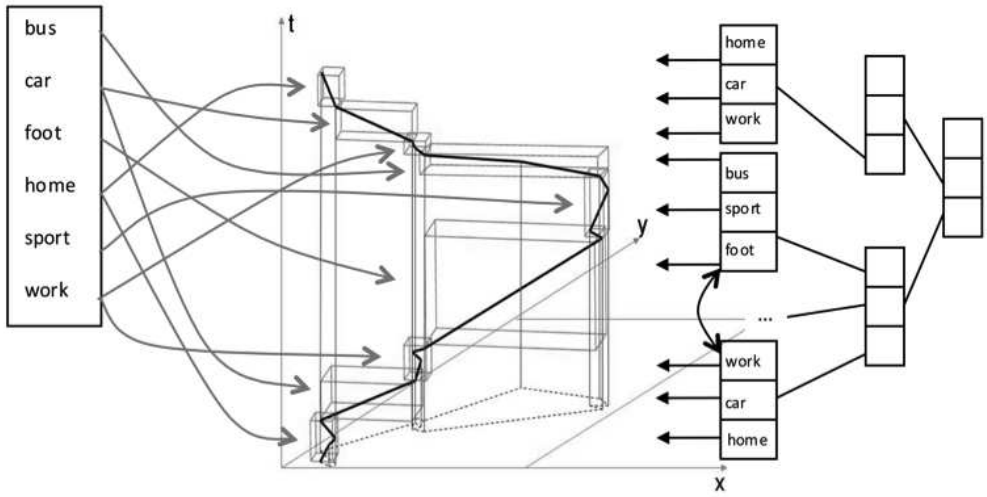


Fig. 2. Hybrid indexing of spatial-temporal-textual information combining 3-dimensional indices for (raw) trajecotries and inverted files.

A. Indexing Mobility Timelines

As usual in ADT, the introduced query operators ask for efficient index support. Motivated by solutions already proposed in the field of geographic information retrieval [18], we propose hybrid access methods that extend the well-known TB-tree [14] and 3DR-tree [16] access methods, proposed for (raw) trajectory data, by combining them with an appropriate text index (inverted file). The architecture of our hybrid indexing scheme is illustrated in Fig. 2.

In particular, Fig. 2 exhibits how the LifeSteps of a single Timeline are indexed. Let us consider a user's Timeline consisting of nine LifeSteps (the 3D MBBs of which are illustrated in the middle of Fig. 2). The right part of Fig. 2 illustrates a 3-dimensional index for (raw) trajectories, could be a TB-tree or a 3DR-tree, constructed by these MBBs, enhanced at the leaf level with the textual information assigned to each LifeStep. Thus, the entries of the leaves contain the tags of the LifeSteps, as well as the MBBs of the LifeSteps' sub-trajectories (in Fig. 2 the latter are pointed by the arrows). This choice allows for queries with combined spatio-temporal and textual constraints. At the left part of the architecture, we adopt a typical inverted file organizing the tags that appear at the LifeSteps. We call the overall scheme SemTB-tree or Sem3DR-tree, depending on the respective trajectory index adopted at the right part.

Whichever is the choice, note that such a spatio-temporal index is different from the one we would have, if we decided to index the initial trajectories. This is due to the segmentation of

trajectories to sub-trajectories that produces a more effective partitioning of the space-time with less dead space, as 'Stop' sub-trajectories (intuitively, they correspond to long MBBs in the time dimension with small spatial footprint) are not mixed with 'Move' sub-trajectories (intuitively, they have large spatial extent). This mixing is a source of inclusion of dead space in the structure and state-of-the-art indexing methods have been proposed to tackle this issue. Here, we implicitly tackle it via the prior segmentation of the initial raw trajectories with respect to their semantics.

B. Querying Mobility Timelines

In a SMD, an interesting operation is *searching for mobility timelines that follow a specific sequence pattern*. Of course, the challenge that arises is that LifeSteps composing Timelines impose textual as well as spatio-temporal constraints. Thus, the query "*Find people who follow the home–office–*–gym pattern*" implies that the user may add spatio-temporal constraints at each of the textual constraints (i.e. search for "home" in region *r* during temporal period *p*). A *Spatial-Temporal-Textual Pattern* (ST^2P) query in a SMD is essentially a (simplified) regular expression consisting of LifeStep objects. In particular, it is defined as a sequence of LifeSteps that forms a search pattern in a SMD. Formally:

$$Q := <p^* \mid p \text{ is either a LifeStep } ls_i \text{ or a wildcard } w \in \{>, *\}> \quad (1)$$

For ease of exposition, Q is represented as an array of LifeSteps L along with an array of wildcards W. The array of wildcards consists of either '>' or '*' (or the empty symbol ∅), which may be placed in-between consecutive LifeSteps. The existence of '>' between two LifeSteps ls_i and ls_{i+1} implies that ls_i is immediately followed by ls_{i+1}, while '*' means that there may exist an arbitrary number of other LifeSteps in between ls_i and ls_{i+1}. Thus, the array of wildcards W is consistent with the array of LifeSteps L. For instance, for $L=[ls_1, ls_2, ls_3, ls_4]$ and $W=[\emptyset, >, *, >]$, the pattern

$$Q(L, W) = [ls_1 > ls_2 * ls_3 > ls_4]$$

conforms to Timelines that start from ls_1, immediately followed by ls_2, then followed by an arbitrary number of LifeSteps of any type, then followed by ls_3, immediately followed by ls_4, which is the ending LifeStep of the Timeline.

Algorithm ST^2P provides the pseudocode for processing ST^2P queries using the hybrid indexing scheme (SemTB-tree or Sem3DR-tree) proposed in the previous subsection. Before providing the details of the algorithm, we should note that pattern_tags() searches the index starting from the inverted file (Fig. 2, left), while pattern_mbb() searches the index starting from the spatio-temporal index (Fig. 2, right).

The algorithm iterates through the LifeSteps of the query pattern (lines 3-9) and finds candidate solutions that satisfy textual constraints (using the pattern_tags() function), which are used to prune the spatio-temporal space (using the pattern_mbb() function). In each of the following iterations, the algorithm moves on to the next input LifeStep and retrieves a corresponding set of LifeSteps from SMD. During each iteration, candidate solutions must also constitute a 'continuation' with the solutions found in the previous step.

Algorithm ST^2P
Input: a pattern query Q as an array of LifeSteps *L* along with an array of wildcards *W*, the *root* of the index
Output: the IDs of the timelines that conform to pattern Q=(*L*,*W*)
1. **begin**
2. *curr_sol* = ∅
3. **for each** ls_i in *L*
4. *curr_sol* = *pattern_tags*(*L*(*i*), *W*(*i*), *curr_sol*, root)
5. **if** curr_sol = ∅ **then**
6. **break**
7. **end if**
8. *curr_sol* = *pattern_mbbs*(*L*(*i*), *W*(*i*), ∅, *curr_sol*, *root*)
9. **end for**
10. **return** timeline IDs from *curr_sol*
11. **end**

Algorithm pattern_tags
Input: a LifeStep *ls*, a wildcard *w*, a pointer to the *curr_sol* relation, the *root* of the index
Output: a subset of the *curr_sol* relation that satisfy *ls.tags*
1. **begin**
2. **if** *w* = ∅ **then**
3. *solutions* = *get_LifeSteps*(*ls.tags*, *root*)
4. **else**
5. *new_sol* = *get_LifeSteps*(*ls.tags*, *root*)
6. *solutions* = *combine*(*curr_sol*, *new_sol*, *w*, *root*)
7. **end if**
8. **return** *solutions*
9. **end**

Algorithm pattern_mbb
Input: a LifeStep *ls*, a wildcard *w*, a pointer to the *prev_sol* relation, a pointer to the *curr_sol* relation, the *root* of the index
Output: a subset of the candidate solutions that satisfy *ls.mbb*
1. **begin**
2. **if** *ls*.mbb = ∅ **then**
3. solutions= *curr_sol*
4. **else**
5. solutions= *get_LifeSteps*(*ls.mbb*, *curr_sol*, *root*)
6. **end if**
7. **if** *w* = ∅ **or** prev_sol =∅ **then**
8. **return** solutions
9. **else**
10. **return** *combine*(*prev_sol*, solutions, *w*, *root*)
11. **end if**
12. **end**

In this case, 'continuation' means that LifeSteps retrieved by searching the textual and the spatio-temporal space must belong to the same timeline and that the second LifeStep comes after the first with respect to time and according to the wildchar between them. This process guarantees that at the end *curr_sol* holds LifeSteps of timelines following the whole pattern (line 10).

In detail, Algorithm pattern_tags examines the wildcard *w* and either retrieves solutions (i.e. LifeSteps) from the index satisfying the textual constraints of the input LifeStep, or combines this result with current solutions found from a previous step. The combine function ensures the continuation of previous step's solutions with the currently found from the index, as we described earlier.

More specifically, given two sets of solutions and a wildcard (i.e. '>' or '*'), the *combine* function joins the two sets on *o_id*, *timeline_id* fields and then based on the wildcard, it imposes the continuation by using the *node_id*, *entry_id* and *numOfEntries* fields. A LifeStep of the second set is the

continuation of a LifeStep of the first set if, in case having a '>' wildcard, both belong to the same leaf and their entries differing by one, or they belong to different leaves (neighboring leaves of the same timeline) and the first LifeStep is the last entry in its leaf, while the second LifeStep is the first entry in its leaf. In case having a '*' wildcard, both should belong to the same leaf and their entries differing by at least one or they should belong to different leaves (of the same timeline). Obviously, the resulting solutions come from the second set, so the algorithm is progressing one step further.

Candidate solutions found to satisfy textual constraints help in pruning the search space in pattern_mbb function. The pattern_mbb algorithm in line 2 checks the existence of input LifeStep's mbb. If not, then solutions found by pattern_tags (i.e. *curr_sol*) are the only candidate solutions so far. Otherwise, solutions are found by retrieving the LifeSteps from the index by traversing it with respect to the spatio-temporal constraints of the *mbb* of the LifeStep. Note that when we reach a leaf this may be pruned, by checking its existence within the solutions found from pattern_tags, thus saving searching its entries. This is represented by the function get_LifeSteps (line 5). Solutions found are returned to main algorithm in line 8 as the combine for current iteration is already executed in pattern_tags (that is the reason why the main algorithm passes an empty set for parameter *prev_sol*).

C. Feeding Semantic Mobility Networks

Feeding data cubes from databases is not a straightforward approach due to the dimensionality and the eventual complexity of the measures in the data cube as well as the size of the database. Therefore, there have been proposed appropriate Extract-Transform-Load (ETL) operations for this task. In the case of SMDC proposed in Section IV, while loading data into the dimension tables is straightforward, feeding the fact tables is not so. In this subsection, we provide three alternative approaches:

(i) The so-called *LifeStep-based* approach scans the SMD sequentially without taking advantage of the index; this decision is based on the rationale that using access methods for sequential data may not be efficient, as it was experimentally shown for the case of TDW [6].

(ii) The so-called *cell-based* approach does make use of the index by following a cell-oriented methodology: for each (spatial-temporal-textual) cell of the data cube, it searches for the LifeSteps that partially match the cell by applying spatial-temporal-textual queries, then calculates measures over these LifeSteps.

(iii) The so-called *text-based* approach is based on the intuition that the semantics (i.e. the textual domain) is usually more selective than the spatio-temporal domain, so again an index-based search is followed but, this time, the index is propagated according to the textual constraints and then the result is refined according to the spatio-temporal constraints.

In order to materialize the above approaches for feeding the Stops-Fact table in SMDC, we propose the respective algorithms, called LifeStepStopsLoad, CellStopsLoad, and TextStopsLoad, respectively.

Rather than calculating measures for each cell of the SMDC, Algorithm LifeStepStopsLoad first finds valid cells for each LifeStep of the SMD and then calculates measures for these cells only. Valid cells are those SMDC cells that spatially-temporally-textually match a LifeStep. In detail, for each LifeStep in the SMD (line 2), valid periods are found by checking the matching of the LifeStep's lifespan with the Period_Dim (line 4). Then, for each period found (line 4), stop_sems set is found by spatio-textually matching the LifeStep's MBR and tags with the Stop_Sems_Dim. Then, for these valid cells, measures in Stops-Fact table are calculated (lines 6-7).

On the other hand, Algorithm CellStopsLoad and Algorithm TextStopsLoad take advantage of the SMD indexing scheme. Algorithm CellStopsLoad applies a spatio-temporal range query (line 4) that returns LifeSteps falling inside cells (i.e. query ranges), which are constructed by combining the Period_Dim and Stop_Sems_Dim dimensions (lines 2-4). Textual constraints are imposed at the leaf level of the index. Again, for each LifeStep, measures in Stops-Fact table are calculated (lines 5-6). In turn, Algorithm TextStopsLoad utilizes the pattern_tags algorithm (i.e. inverted file) to find candidate solutions based on the textual constraints of each value in Stop_Sems_Dim (line 2). Then, these candidate solutions are passed to a spatio-temporal range query (line 5) to prune the search space. The remaining steps are the same as in the CellStopsLoad algorithm.

In the same fashion, in order to materialize the above approaches for feeding the Moves-Fact table, we propose the respective algorithms, called LifeStepMovesLoad, CellMovesLoad, and TextMovesLoad, respectively.

Algorithm LifeStepMovesLoad scans the SMD for each 'Move' LifeStep and finds its tags (line 3) and time periods (line 4). Then, for each time period, it tries to find matching cells in the Stop_Sems_Dim dimension with its previous 'Stop' LifeStep (line 6). Similarly, it tries to find matching cells with its next 'Stop' LifeStep (line 8). Finally, for each returned cell, measures in Moves-Fact table are calculated (lines 9-10).

In turn, Algorithm CellMovesLoad in its core calls an ST^2P query, restricting the result to the 'Move' LifeSteps that follow query pattern Q := [from_ls > via_ls > to_ls] consisting of three LifeSteps (lines 9-10). These LifeSteps are constructed by all triplet combinations of values from the dimensions, which textually match 'Move' LifeSteps created form the Move_Sems_Dim (lines 7-8) and also match spatially-temporally-textually their previous and next 'Stop' LifeSteps, created from the Stop_Sems_Dim and Period_Dim dimensions (lines 2-6). Thus, *Q* is a Stop-Move-Stop- like pattern. For each 'Move' LifeStep found, measures in Moves-Fact table are calculated (line 11-12).

Finally, Algorithm TextMovesLoad tries to find in a step-by-step fashion 'Move' LifeSteps that conform to values of the Move_Sems_Dim (note that this dimension includes only textual values), also having the respective previous and next 'Stop' LifeSteps textually conforming to values in Stop_Sems_Dim (namely, we ignore the spatial part of this spatio-textual dimension). Thus, the algorithm first uses the inverted file to deal with textual constraints (lines 2-8).

Algorithm LifeStepStopsLoad
Input: a SMD
Output: updated Stops_Fact table
1. **begin**
2. **for each** 'Stop' ls_k in SMD
3. periods=*get_Periods*(ls_k)
4. **for each** p_j in periods
5. stop_sems=*get_Stop_sems*(ls_k)
6. **for each** s_s_i in stop_sems
7. *CalcMeasures*(ls_k, s_s_i, p_j)
8. **end for**
9. **end for**
10. **end for**
11. **end**

Algorithm CellStopsLoad
Input: the root of the index
Output: updated Stops_Fact table
1. **begin**
2. **for each** s_s_i in Stop_Sems_Dim
3. **for each** p_j in Period_Dim
4. LifeSteps = *get_Stop_LifeSteps(*root, s_s_i, p_j)
5. **for each** ls_k in LifeSteps
6. *CalcMeasures*(ls_k, s_s_i, p_j)
7. **end for**
8. **end for**
9. **end for**
10. **end**

Algorithm TextStopsLoad
Input: the *root* of the index
Output: updated Stops_Fact table
1. **begin**
2. **for each** s_s_i in Stop_Sems_Dim
3. *curr_sol =pattern_tags*(*LifeStep(*s_s_i*)*, ∅, ∅, root)
4. **for each** p_j in Period_Dim
5. LifeSteps = *get_Stop_LifeSteps(*root, s_s_i, p_j, *curr_sol*)
6. **for each** ls_k in LifeSteps
7. *CalcMeasures*(ls_k, s_s_i, p_j)
8. **end for**
9. **end for**
10. **end for**
11. **end**

Algorithm LifeStepMovesLoad
Input: a SMD
Output: updated Moves_Fact table
1. **begin**
2. **for each** 'Move' ls_k in SMD
3. m_sem=*get_Move_sems*(ls_k)
4. periods=*get_Periods*(ls_k)
5. **for each** p_j in periods
6. from_stop_sems=*get_Stop_sems*(ls_{k-1})
7. **for each** from_s_s_f in from_stop_sems
8. to_stop_sems=*get_Stop_sems*(ls_{k+1})
9. **for each** to_s_s_t in to_stop_sems
10. *CalcMeasures*(ls_k, from_s_s_f, to_s_s_t, m_sem, p_j))
11. **end for**
12. **end for**
13. **end for**
14. **end for**
15. **end**

Algorithm CellMovesLoad
Input: the *root* of the index
Output: updated Moves_Fact table
1. **begin**
2. **for each** p_j in Period_Dim
3. **for each** from_s_s_f in Stop_Sems_Dim
4. from_ls=LifeStep('Stop', from_s_s_f, p_j)
5. **for each** to_s_s_t in Stop_Sems_Dim
6. to_ls=LifeStep('Stop', to_s_s_t, p_j)
7. **for each** m_s_m in Move_Sems_Dim
8. via_ls=LifeStep('Move', m_s_m)
9. *L*=[from_ls, via_ls, to_ls]; *W*=[∅, >, >]; *Q*=(*L*, *W*)
10. LifeSteps = ST^2P (*Q*)
11. **for each** ls_k in LifeSteps
12. *CalcMeasures*(ls_k, from_s_s_f, to_s_s_t, m_s_m, p_j))
13. **end for**
14. **end for**
15. **end for**
16. **end for**
17. **end for**
18. **end**

Algorithm TextMovesLoad
Input: the *root* of the index
Output: updated Moves_Fact table
1. **begin**
2. **for each** from_s_s_f in Stop_Sems_Dim
3. from_sol= *pattern_tags*(LifeStep(from_s_s_f), ∅, ∅, root)
4. **for each** m_s_m in Move_Sems_Dim
5. move_sol= *pattern_tags*(LifeStep(m_s_m), '>', from_sol, root)
6. **for each** to_s_s_t in Stop_Sems_Dim
7. to_sol= *pattern_tags*(LifeStep(to_s_s_t), '>', move_sol, root)
8. candidate_moves=*get_Moves*(move_sol, to_sol)
9. **for each** p_j in Period_Dim
10. from_sol= *pattern_mbb*(LifeStep(from_s_s_f, p_j), ∅, ∅, from_sol, root)
11. to_sol= *pattern_mbb*(LifeStep(to_s_s_t, p_j), '>', candidate_moves, to_sol, root)
12. result_moves=*get_Previous_Moves*(to_sol)
13. **for each** m_g in result_moves
14. *CalcMeasures*(m_g, from_s_s_f, to_s_s_t, m_s_m, p_j))
15. **end for**
16. **end for**
17. **end for**
18. **end for**
19. **end for**
20. **end**

Note the successive application of pattern_tags algorithm, where at each step we combine with the results found before the step to prune the search space. Then, the algorithm proceeds to apply spatio-temporal constraints to the (already found) previous and next 'Stop' LifeSteps, with respect to the candidate 'Move' LifeSteps found by the textual filtering (lines 9-12). For each qualifying 'Move' LifeStep measures in Moves-Fact table are calculated (lines 13-14).

VI. EXPERIMENTAL STUDY

We have developed the proposed *Hermes*^{sem} framework extending Hermes MOD engine [10] and we present preliminary experimental results. Although many aspects can be experimentally studied, due to space limitations we focused on processing time.

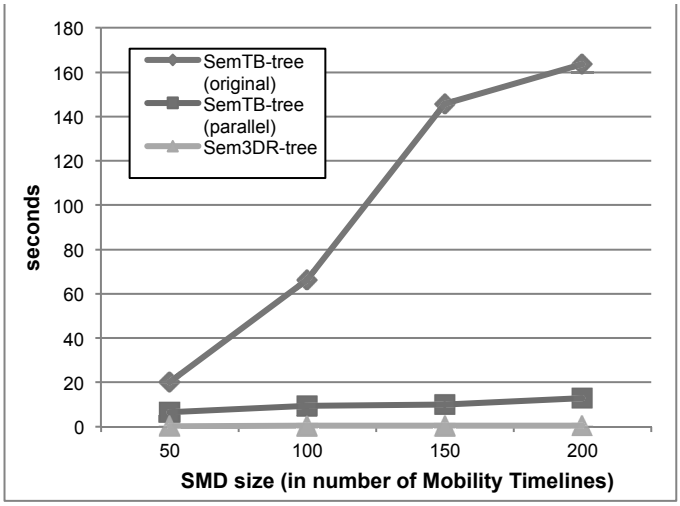


Fig. 3. Sem-TB-tree vs. Sem-3DR-tree construction time

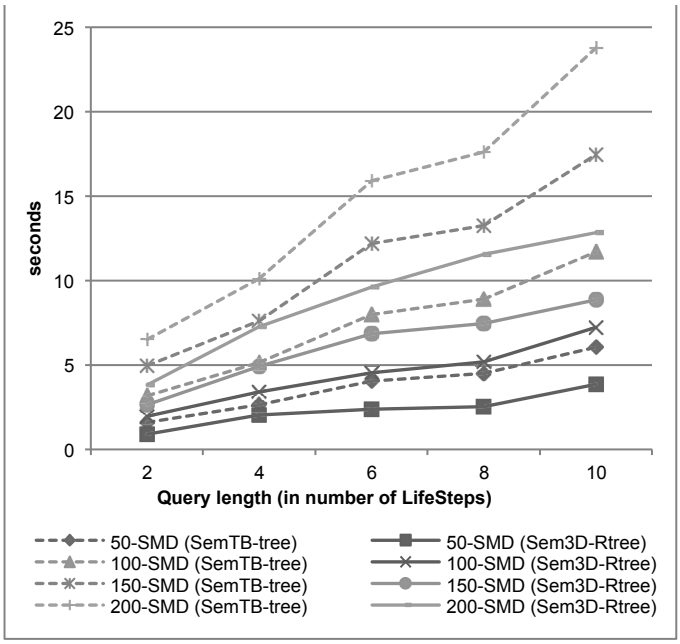


Fig. 4. ST^{2}P query processing time

All experiments were performed in a PC of i7 CPU with 4 cores at 1.73 GHz and of 8Gb RAM. Source code and used datasets is available at: http://infolab.cs.unipi.gr/hermes/. Input SMDs were simulations of mobility scenarios by the Hermoupolis semantic trajectory generator [12]. We simulated two mobility scenarios: the first scenario is a 7-days movement in the city of Athens for 4 different profiles (movement behaviors); the second scenario is a 1-day movement in the city of Athens, again for 4 different profiles. For both scenarios we created 4 SMDs consisting of 50, 100, 150 and 200 timelines. The average number of LifeSteps for each scenario is 58 and 10, respectively. We used the first scenario for ST^{2}P query experiments (i.e. in order to evaluate the performance of the algorithm proposed in Section V.B) and the second for SMDC feeding experiments (i.e. in order to evaluate the performance of the algorithms proposed in Section V.C).

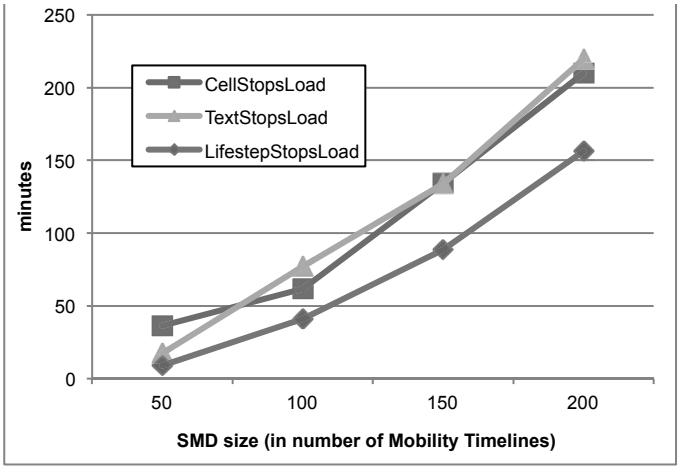


Fig. 5. SMDC feeding processing time (Stops-Fact table)

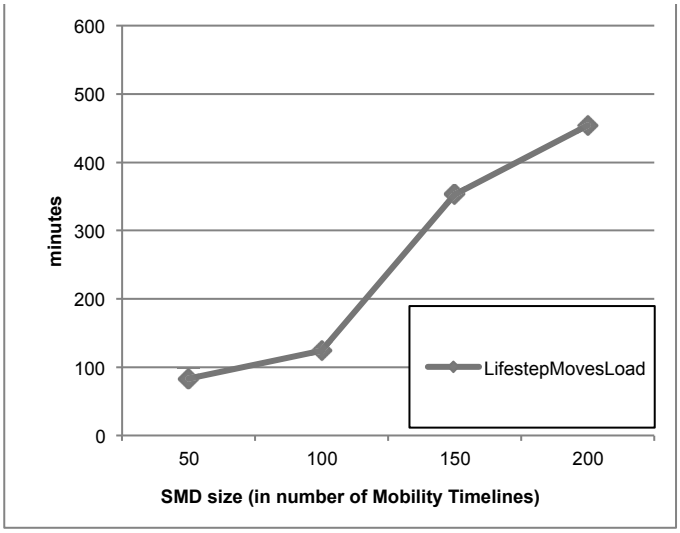


Fig. 6. SMDC feeding processing time (Moves-Fact table)

We start our performance study by calculating the time required to build the two alternative indexing structures. Fig. 6 illustrates the results for either one-by-one insertion or bulk loading of the SMD content into the index. The significant gain when performing bulk loading on Sem-3DR-tree is due to the fact that the loading task can be parallelized by partitioning the SMD timelines. (In our implementation that is found behind the numbers in Fig. 3, we used 4 parallel processes.) On the other hand, we provide a single series for Sem-3DR-tree since it exploits on the DBMS built-in R-tree index. To measure processing time on ST^{2}P queries we set up queries of 2, 4, 6, 8 and 10 input LifeSteps with various wildcards in between them. Fig. 4 depicts the performance of our algorithm in all datasets using both types of indices, i.e. Sem3D-Rtree and SemTB-tree. We note that increased query complexity does not increase significantly processing time, while in all cases the Sem3D-Rtree performs better than SemTB-tree. This is rather expected, as the ST^{2}P query can be considered as a sequence of coordinate-based (i.e. range) queries, which have been shown in the literature to be supported more efficiently using 3DR-trees rather than TB-trees [14].

Next, we provide the results for feeding the SMDC Stops-Fact table and Moves-Fact table, in Fig. 5 and Fig. 6, respectively. Regarding Stops (Fig. 5), the three approaches do not appear to have significantly different performance (linear with respect to the number of Timelines). On the other hand, Moves (Fig. 6) are much more expensive to process; in this case, the LifeStep-based approach appears to be the only one that performs in acceptable time (therefore, the other two are omitted from the chart).

VII. SUMMARY - FUTURE CHALLENGES

In this paper, motivated by related work on semantic trajectories [9] and an envisioned framework for their realization [11], we presented the *Hermes*^{sem} integrated MOD / SMD / SMDC engine. In this line, we proposed efficient access methods for the hybrid indexing of the spatial-temporal - textual component of this type of data.

We also devised efficient query processing algorithms to support a very interesting query type, called Spatial-Temporal-Textual Pattern (ST^2P) query that receives as input a sequence of LifeSteps (to be considered as a simplified regular expression) and outputs the Timelines that obey to the constraints of this sequence. To the best of our knowledge, it is the first time that such pattern queries are discussed in the spatial-temporal-textual domain. Moreover, we presented a data constellation schema for modeling SMDC and devised algorithms for the efficient feeding of its fact tables.

We should note that the proposed development approach is only a first SMD and SMDC implementation and it could only be used as a baseline in future works. For instance, the storage layer could be physically organized according to any data management paradigm (centralized, distributed, map-reduce, etc.), the hybrid indexing of the spatial-temporal-textual information as well as the ST^2P query processing and optimization challenges could find more attractive solutions, and so on. Our goal in this work was to design a unified framework that solves many (though not all) raised issues better than legacy approaches can. Thus, we believe that each of the tackled issues could be a research challenge per se. This is our plan for the near future.

ACKNOWLEDGMENT

This work has been supported by the European Union's FP7-DATASIM (grant 270833) and IRSES-SEEK (grant 295179) projects. Nikos Pelekis' research has been co-financed by the European Union (European Social Fund - ESF) and Greek national funds through the Operational Program "Education and Lifelong Learning" of the National Strategic Reference Framework (NSRF) - Research Funding Program: Thales.

REFERENCES

[1] R. Fileto, A. Raffaetà, A. Roncato, J. A. P. Sacenti, C. May, D. Klein, "A Semantic Model for Movement Data Warehouses," DOLAP, 2014.

[2] F. Giannotti, M. Nanni, D. Pedreschi, F. Pinelli, C. Renso, S. Rinzivillo, R. Trasarti, "Unveiling the complexity of human mobility by querying and mining massive trajectory data," VLDB Journal, 20(5): 695-719, 2011.

[3] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, H. Pirahesh, "Data cube: A relational aggregation operator generalizing groub-by, cross-tab and sub-totals," Data Mining and Knowledge Discovery, 1(1):29-54, 1997.

[4] R. H. Güting, T. Behr, C. Düntgen, "SECONDO: A Platform for Moving Objects Database Research and for Publishing and Integrating Research Implementation," IEEE Data Engineering Bulletin, 33(2):56-63, 2010.

[5] J. Han, N. Stefanovic, K. Koperski, "Selective materialization: An efficient method for spatial data cube construction," PAKDD, 1998.

[6] A. Raffaetà, L. Leonardi, G. Marketos, G. Andrienko, N. Andrienko, E. Frentzos, N. Giatrakos, S. Orlando, N. Pelekis, A. Roncato, C. Silvestri, "Visual Mobility Analysis using T-Warehouse," International Journal of Data Warehousing & Mining, 7(1), 2011.

[7] L. Leonardi, S. Orlando, A. Raffaetà, A. Roncato, C. Silvestri, G.L. Andrienko, N.V. Andrienko, "A general framework for trajectory data warehousing and visual OLAP," GeoInformatica 18(2): 273-312, 2014.

[8] S. Orlando, R. Orsini, A. Raffaetà, A. Roncato, C. Silvestri, "Spatio-temporal aggregations in trajectory data warehouses," DaWaK, 2007.

[9] C. Parent, S. Spaccapietra, C. Renso, G. Andrienko, N. Andrienko, V. Bogorny, M.L., Damiani, A. Gkoulalas-Divanis, J.A. Macedo, N. Pelekis, Y. Theodoridis, Z. Yan, "Semantic trajectories modeling and analysis," ACM Computing Surveys, 45(4), 2013.

[10] N. Pelekis, E. Frentzos, N. Giatrakos, Y. Theodoridis, "HERMES: A Trajectory DB Engine for Mobility-Centric Applications", Int'l Journal of Knowledge-based Organizations (IJKBO), 5(2), 19-41, 2015.

[11] N. Pelekis, Y. Theodoridis, D. Janssens, "On the Management and Analysis of Our LifeSteps," SIGKDD Explorations, 15(1), 23-32, 2013.

[12] N. Pelekis, S. Sideridis, P. Tampakis, Y. Theodoridis "Hermoupolis: A Semantic Trajectory Generator in the Data Science era", The SIGSPATIAL Special Newsletter of the ACM,. 7(1), 2015.

[13] N. Pelekis, Y. Theodoridis, "Mobility Data Management and Exploration," Springer, New York, 2014.

[14] D. Pfoser, C.S., Jensen, Y. Theodoridis, "Novel Approaches to the Indexing of Moving Object Trajectories," VLDB, 2000.

[15] G. Salton, C. Buckley, "Term-weighting approaches in automatic text retrieval," Information Processing and Management: an International Journal, 24(5): 513–523, 1988.

[16] Y. Theodoridis, M. Vazirgiannis, T. Sellis, "Spatio-temporal indexing for large multimedia applications," ICMCS, 1996.

[17] R. Wagner, A. Raffaeta, A. Roncato, J. A. de Macedo, R. Trasarti, C. Renso, "Mob-Warehouse: a semantic approach for mobility analysis with a trajectory data warehouse," SECOGIS, 2013.

[18] D. Wu, G., Cong, C.S. Jensen, "A framework for efficient spatial web object retrieval," The VLDB Journal, 21:797–822, 2012.

[19] Z. Yan, D. Chakraborty, C. Parent, S. Spaccapietra, K. Aberer, "SeMiTri: a framework for semantic annotation of heterogeneous trajectories," EDBT, 2011.

[20] P. Zhao, X. Li, D. Xin, J. Han, "Graph cube: on warehousing and OLAP multidimensional networks," SIGMOD, 2011.