Spatio-temporal Range Searching Over Compressed Kinetic Sensor Data[*]

Sorelle A. Friedler[†]
sorelle@cs.umd.edu
http://www.cs.umd.edu/~sorelle

David M. Mount[‡]
mount@cs.umd.edu
http://www.cs.umd.edu/~mount

Dept. of Computer Science, University of Maryland, College Park, MD 20742

Abstract

As sensor networks increase in size and number, efficient techniques are required to process the very large data sets that they generate. Frequently, sensor networks monitor objects in motion within their vicinity; the data associated with the movement of these objects are known as kinetic data. In an earlier paper we introduced an algorithm which, given a set of sensor observations, losslessly compresses these data to a size that is within a constant factor of the asymptotically optimal joint entropy bound. In this paper we present an efficient algorithm for answering spatio-temporal range queries. Our algorithm operates on a compressed representation of the data, without the need to decompress it. We analyze the efficiency of our algorithm in terms of a natural measure of information content, the joint entropy of the sensor outputs. We show that with space roughly equal to joint entropy, queries can be answered in time that is roughly logarithmic in joint entropy. In addition, we show experimentally that on real-world data our range searching structures use less space and have faster query times than the naive versions. These results represent the first solutions to range searching problems over compressed kinetic sensor data.

1 Introduction

Sensor networks and the data they collect have become increasingly prevalent. They are frequently employed to observe objects in motion and are used to record traffic data [16, 30], observe wildlife migration patterns [24, 33], and observe motion from many other settings [2]. In order to perform accurate statistical analyses of these data over arbitrary periods of time, the data must be faithfully recorded and stored. For example, a large sensor network observing a city's traffic patterns may generate gigabytes of data each day [16]. The vast quantities of such data necessitate compression of the sensor observations, yet analyses of these observations is desirable. Ideally, such analyses should operate over the compressed data without the need to decompress it. In order to perform statistical analyses of the data, it is often desirable that retrieval queries be supported. In this

[*]A shorter version of this paper appeared in the Proceedings of the European Symposium on Algorithms, 2010.

[†]The work of Sorelle Friedler has been supported in part by the AT&T Labs Fellowship Program and the Ann G. Wylie Dissertation Fellowship.

[‡]The work of David Mount has been supported in part by the National Science Foundation under grant CCR-0635099 and the Office of Naval Research under grant N00014-08-1-1015

paper, we present the first data structure and algorithms for answering range searching queries over compressed data streams arising from large sensor networks.

In an earlier paper [12], we presented an algorithm for losslessly compressing kinetic sensor data and a framework for analyzing its performance. (See Section 2 for a brief introduction.) We assume that we are given a set of sensors, which are at fixed locations in a space of constant dimension (our results apply generally to metric spaces of constant doubling dimension [21].) These sensors monitor the movement of a number of kinetic objects. Each sensor monitors an associated region of space, and at regular time steps it records an occupancy count of the number of objects passing through its region. Over time, each sensor produces a string of occupancy counts; the problem considered in [12] is how to compress all these strings.

Previous compression of sensor data in the literature has focused largely on approximation algorithms in the streaming model or lossy compression of the data. We consider lossless compression. This is often more appropriate in scientific contexts, where analysis is performed after the data has been collected and accurate results are required. The analysis of these results may necessarily include consideration of outliers or unusual data features that might be smoothed away by lossy compression techniques. In addition, in scientific contexts the loss of data associated with any kind of lossy compression is considered unacceptable. Lossless compression algorithms have been studied in the single-string setting [19, 27, 35, 36] but remain mostly unstudied in a sensor-based setting [12].

In order to query observed sensor data, which ranges over time and space, we need to consider both temporal and spatial queries. *Temporal range queries* are given a time interval and return an aggregation of the observations over that interval. *Spatial range queries* are given some region of space (e.g., a rectangle, sphere, or halfplane) and return an aggregation of the observations within that region. *Spatio-temporal range queries* generalize these by returning an aggregation restricted by both a temporal and a spatial range. For example, a spatio-temporal range query might return the sum of the observed object counts within a spherical range of sensors over a given time period. We assume that occupancy counts are taken from a commutative semigroup of fixed size, and the result is a semigroup sum over the range. There are many different data structures for range searching (on uncompressed data) depending on the properties of the underlying space, the nature of the ranges, properties of the semigroup, and whether approximation is allowed [1, 23]. One of the fundamental hierarchical structures, on which much other work has been based, is the quadtree, a data structure that divides the space into nested rectangular regions [29]. The net-tree is a data structure similar in spirit, but for which the fundamental unit is the ball instead of the rectangle [17].

We present data structures for storing compressed sensor data and algorithms for performing spatio-temporal range queries over these data. We analyze the quality of these range searching algorithms in terms of both time and space by considering the information content of the set of sensor outputs. There are two well-known ways in which to define the information content of a string, classical statistical (Shannon) entropy and empirical entropy. Statistical entropy [31] is defined under the assumption that the source X is drawn from a stationary, ergodic random process. The normalized statistical entropy, denoted $H(X)$, provides a lower bound on the number of bits needed to encode a character of X. In contrast, the empirical entropy [20, 22], denoted $H_k(X)$, while similar in spirit to the statistical entropy, assumes no underlying random process and relies only on the observed string and the context of the most recent k characters. These definitions and distinctions are discussed in more detail in a companion paper [13].

Bounds for Range Searching

	Temporal	Spatio-temporal
Preprocessing time	$O(\mathrm{Enc}(X))$	$O(\mathrm{Enc}(\mathbf{X}))$
Query time	$O(\log T)$	$O(((1/\varepsilon^{d-1}) + \log S) \log T)$
Space	$O(\mathrm{Enc}(X))$	$O(\mathrm{Enc}(\mathbf{X}) \log S)$

Table 1: Time and space bounds for temporal range searching and ε-approximate spatio-temporal range searching for fat convex ranges in $\mathbb{R}^d$. S is the number of sensors in the network, T is the length of the observation period, and $\mathrm{Enc}(X)$ and $\mathrm{Enc}(\mathbf{X})$ denote the sizes of the compressed representations for single sensor stream (for temporal range searching) and sensor system (for spatio-temporal range searching), respectively.

Previously, retrieval over compressed text (without relying on decompression) has been studied in the context of strings [3, 10, 11, 15] and XML files [9]. For example, Ferragina and Venturini [11] show that it is possible to retrieve a substring (indexed by start and end times) in the compressed text with query time equal to $O(1 + \frac{\ell}{\log T})$ where ℓ is the length of the substring and T is the length of the string X. Their space requirement is $T \cdot H_k(X) + o(T)$ bits. Their data structure allows substring queries, which are very different from semigroup range searching queries, which we consider here. For surveys of this area see [18, 25].

Although here we will present data structures that operate in main memory, for the large data sets generated by sensor networks it may also be useful to consider Input/Output (I/O) efficient structures. Since the data structures described here are based on simple, practical structures, modifications to I/O-efficient versions should be straightforwardly based on known I/O-efficient equivalents. Specifically, the temporal structure described could be modified to be based on an underlying I/O-efficient compressed text structure [7] and combined with a spatio-temporal structure modified to be based on an I/O-efficient kd-tree [26, 28].

1.1 Results

In this paper we present the first range query results over compressed kinetic sensor data. Specifically, we consider the problems of temporal range searching and spatio-temporal range searching for fat convex ranges (e.g. spheres, rectangles with low aspect ratio, etc. [4]).

As mentioned earlier, we analyze our algorithms in terms of the joint entropy of the sensor outputs. The preprocessing makes only one pass over the compressed data, and thus it can be performed as the data are collected. The query bounds are logarithmic in the input size. The space bounds, given in bits, match the entropy lower bound up to constant factors. Specific bounds are given in Table 1. The temporal range query data structures and associated bounds are discussed more specifically in Section 3 and the spatio-temporal results are discussed in Section 4.

In addition to theoretical results, we present experimental evaluation of our temporal range searching structure. These results show that, in addition to being theoretically efficient, our data structure offers a roughly 50-fold improvement in space. These improvements increase as the data sets become larger.

Both our temporal and spatio-temporal data structures are quite practical, being based on very simple data structures (tries, binary trees, and quadtrees, in particular). The temporal range searching data structure relies on the trie created when compressing the data together with an annotated binary tree. The spatio-temporal range searching data structure relies on modifications to an existing quadtree-based data structure used for answering approximate range search queries as well as the temporal range searching solution. The use of these partition-tree based structures requires storage of an aggregated version of the encoded data at each level of the tree, and we show that this only increases the space used by a factor logarithmic in the number of sensors.

2 Framework for Kinetic Sensor Data

In an earlier paper [12] we introduced a framework and a lossless compression scheme for discrete kinetic data observed by a sensor network. This framework will be used as a basis for the results of this paper. We begin with some basic definitions about the structure of the sensor network and the associated observed data streams. Consider a static sensor network with S sensors, monitoring the motion of a collection of moving objects. Let P be a point set indicating the sensor locations. All sensors are assumed to operate over T synchronized time steps. Each sensor observes the motion of objects in some region surrounding it, and records an *occupancy count* indicating the number of objects passing within its region during the observed time step. (We think of the object motion as continuous even though our observation of it is discrete.) Our results apply more generally to any discrete statistic over a domain of constant size. No assumptions are made about the nature of the point motion nor the nature of the sensor regions (e.g., their shapes, density, disjointness, etc.).

Central to our framework is the notion that each sensor's output is statistically dependent on a relatively small number of nearby sensors. For some point $p \in P$, let $NN_m(p) \subseteq P$ be the m nearest neighbors of p. Sensors i and j with associated sensor positions $p_i, p_j \in P$ are said to be *mutually m-close* if $p_i \in NN_m(p_j)$ and $p_j \in NN_m(p_i)$. For a constant m, a sensor system is said to be *m-local* if all pairs of sensors that are not mutually m-close are statistically independent.

In [12] we introduced a compression algorithm, *PartitionCompress*, which operates on an m-local sensor system. (The algorithm is sketched in the proof of Lemma 4.1.) It compresses the sensor outputs to within a constant factor c (depending on dimension) of the optimal joint entropy bound.

Intuitively, the compression algorithm is based on the following idea. If two sensor streams are statistically independent, they may be compressed independently from each other. If not, optimal compression can only be achieved if they are compressed jointly. The algorithm works by compressing the outputs from clusters of nearest neighbor groups together, as if they were a single stream. In order to obtain the desired compression bounds, these clusters must be sufficiently well separated so that any two mutually m-close sensors are in the same cluster. *PartitionCompress* partitions the points into a constant number c (independent of m but depending on dimension) of subsets for which this is true and then compresses clusters together to take advantage of local dependencies. The compression of a single cluster may be performed using any string compression algorithm; to obtain the near optimal bound, this algorithm must compress streams to their optimal entropy bound. It is shown in a companion paper [13] that LZ78, the Lempel-Ziv dictionary compression algorithm [36], is sufficient for our purposes.

For the rest of the paper, we will use $\mathrm{Enc}_{alg}(\mathbf{X})$ to denote the length of the encoded set of sensor outputs $\mathbf{X}$, where *alg* specifies the string compressor used by the compression algorithm of [12]. Since the LZ78 algorithm will suffice for our purposes, let $\mathrm{Enc}(\mathbf{X}) = \mathrm{Enc}_{\mathrm{LZ78}}(\mathbf{X})$. In a companion paper [13], it is shown that $\mathrm{Enc}(\mathbf{X})$ is on the order of the optimal space bound when analyzed in terms of either the statistical or empirical entropy. (A slightly weaker form of independence, called δ-independence, was also considered and it was shown that the bounds hold approximately under this weaker definition.)

3 Temporal Range Searching

In this section we describe a data structure that answers temporal range searching queries over a single compressed sensor stream. Let X be a sequence of sensor counts over time period $[1, T]$, which will be compressed and preprocessed into a data structure so that given any temporal range $[t_0, t_1] \in [1, T]$, the aggregated count over that time period can be calculated efficiently. We assume that the individual sensor counts are drawn from a semigroup, and the sum is taken over this semigroup. The space used by the data structure (in bits) will be asymptotically equal to that of the compressed string, and the query time will be logarithmic in T. Here is the main result of this section. Recall that, given string X, $\mathrm{Enc}(X)$ denotes the length of the compressed encoding of X.

Theorem 3.1. *There exists a temporal range searching data structure, which given string X over a time period of length T, can be built in time $O(\mathrm{Enc}(X))$, achieves query time $O(\log T)$, and uses space $O(\mathrm{Enc}(X))$ bits.*

The remainder of this section is devoted to proving this theorem. In Section 3.1 we consider the simpler special case where the semigroup is in fact a group, which means that both addition and subtraction of weights are allowed. In Section 3.2, we consider the general semigroup case, where only addition is allowed.

3.1 Group Setting

We begin by describing the preprocessing for our data structure in the group context, where subtraction of counts is allowed. First, the given sequence X is compressed using the LZ78 compression algorithm and the standard accompanying trie (also known as a *dictionary*) containing nodes that represent *words* is created [36]. We begin with a short overview of this algorithm. LZ78 scans over the input, putting characters into a trie so that each edge in the trie represents a single character. As the string is scanned from beginning to end, the prefix is looked up in the trie and the most recent character is added to that path in the trie. The resulting *word* is added to the compressed version of the string by simply storing a pointer to the bottom most node of the path in the dictionary. Let d be the number of words in the dictionary. Each word in the dictionary (possibly excepting the last) is used in the compressed version of the string exactly once. In addition, each word in the dictionary was generated only after all prefixes had previously been added, so the trie is *prefix-complete* [10]. We will make use of the fact, proved in a companion paper [13], that $d \log d = \mathrm{Enc}(X)$.

Let us now discuss our preprocessing of the stream X. It involves two phases. The first takes place during the single scan through the input. The data are compressed using LZ78 compression, the associated trie is created, and pointers to word endings (called *anchor points*) are stored. Additionally, the aggregated value of each word (e.g. the sum of its component counts, or the *word sum*) is added to the associated node in the dictionary. This value can be found by adding the count at the current node to its parent's stored aggregated value as each letter is added to an existing word in the trie. This phase takes time $O(T)$ and we will refer to the result of this phase as

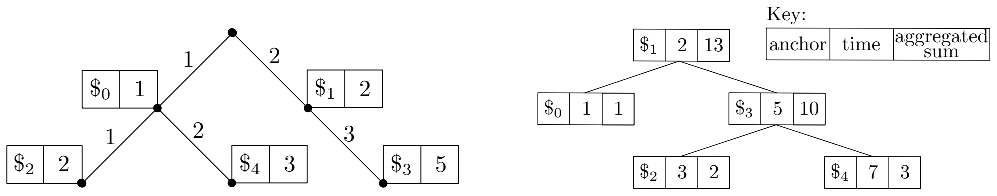

Figure 1: Left: LZ78 trie annotated with associated anchor points and word sums for a single sensor with observation string "12112312". Considered inline, the string with anchor points as breaks between the words becomes 1 $\$_0$ 2 $\$_1$ 11 $\$_2$ 23 $\$_3$ 12 $\$_4$. Right: The corresponding binary search tree based on word start times that also contains aggregated sums for the words contained in each node's subtree. The anchor points are also sorted by the word start times (in the order of their indices) and the start times are stored separately with the anchors so that $\$_0$ is associated with start time 1, $\$_1$ is associated with start time 2, $\$_2$ is associated with start time 3, and so on until $\$_4$ is associated with start time 7. Thus, a temporal range $[3, 7]$ would include parts of the words with anchor points $\$_2$, $\$_3$, and $\$_4$.

the *compressed form* of the input. Here, we briefly present an example of this initial preprocessing step for $X =$ "12112312," with aggregation type a sum of counts. An accompanying trie is shown in Figure 1. The first word found while scanning through the input and entered into the trie as part of the LZ78 compression algorithm is "1" which has associated anchor point $\$_0$ and stored word count sum of 1. Similarly, the next word discovered is "2" with associated anchor point $\$_1$ and word sum 2. The next word is "11" with anchor point $\$_2$ and word sum $1 + 1 = 2$, and so on until the word "12" ending at anchor point $\$_4$ with word sum $2 + 3 = 5$. LZ78 outputs a space-efficient encoding of this trie, but for our purposes it is not necessary to understand the actual encoding. See Figure 1 for an example.

The second phase, which is the one we will analyze for its additional non-compression related time, consists of creating a binary search tree over the anchor points and initializing auxiliary data structures. Building a binary search tree over the anchor points (stored already sorted by word start time) requires $O(d)$ time, since there are d words and each has one associated anchor point. Additionally, we create an aggregation tree over the aggregate word values, so that aggregate values of consecutive words can be easily found when considering substrings. This takes time $O(d)$ when created as an annotation to the existing binary search tree. Finally, we will later need access to a level ancestor data structure, which can be built in $O(d)$ time [5].

Lemma 3.1. *Assuming that the input is given in compressed form, temporal range searching takes preprocessing time $O(d) = O(\mathrm{Enc}(X))$.*

Next we describe query processing. (Here, we will give some examples. The more precise explanation can be found in the proof of Lemma 3.2.) Each temporal query can be categorized as either *internal* or *overlapping* depending on whether the query interval overlaps one word or multiple words, respectively. Internal queries implicitly divide a word into a prefix, query region, and suffix. For example, in Figure 1, consider the query $[5, 5]$, which effectively asks for the value of the fifth character of the string. This query overlaps the first character of the substring "23", and therefore it is an internal query. In this case, the prefix is null (since there are not characters of the substring preceding the query), the query region consists of "2", and the suffix consists of the remainder of the string, namely “3”. Overlapping queries consist of a suffix, one or more complete words, and a prefix of the trailing word. For example, in Figure 1, the overlapping query $[4, 7]$, corresponding to the substring “1231,” consists of the suffix “1” of the word “11,” all of the word “23,” and the prefix “1” of word “12.”

Since the trie is prefix-complete, all prefix aggregations are stored in our annotated trie and can be retrieved in $O(1)$ time using these annotations and the level-ancestor data structure. Entire word aggregate values can be retrieved as a group using the annotated binary search tree created over the aggregate word values. For example, in Figure 1, the aggregate sum of the substring “112312,” with temporal range $[3, 8]$ and consisting of the words with anchor points $\$_2$, $\$_3$, and $\$_4$, can be easily found to be 10 by a look-up in the binary search tree. Finally, suffix values can be retrieved by finding the complementary prefix value and subtracting from the total associated with the entire word. For example, the suffix sum of 1 indicated by the temporal range $[4, 4]$ in Figure 1 can be retrieved by subtracting the value 1 associated with $\$_0$ from the sum of 2 associated with $\$_2$. Using these basic retrieval systems, internal queries can be found by subtracting the prefix and suffix values from the word total and overlapping queries can be determined by adding the suffix, complete word sums, and prefix values.

Lemma 3.2. *The query time for temporal range searching in the group setting is $O(\log d) = O(\log T)$.*

Proof. The compressed text is modified to be of the form $W_1\$W_2\$\dots W_d\$$ where $\{W_1, ..., W_d\}$ are the words in the dictionary and \$ is a character not in the original alphabet. Each \$ is associated with the W_i preceding it, and the location of that \$, or *anchor point*, is added as an annotation to each dictionary word. (These manipulations were introduced by Ferragina and Manzini [10], and though pointers to the beginning of words would suffice for our application, we use the insertion of \$'s for notational convenience.) Since each dictionary word appears exactly once in the compressed text, each word has a single associated anchor point.

When given a temporal range $[t_0, t_1]$, the first step is to locate the anchor points $\$_0$ and $\$_1$ such that $\$_0 \leq t_0$ and $\$_1 \geq t_1$, and there are no other $\$'_0$ or $\$'_1$ such that $\$_0 < \$'_0 \leq t_0$ and $\$_1 > \$'_1 \geq t_1$. This is performed by a binary search through the sorted list of anchor points, which takes time $O(\log d)$. We say that the result is *overlapping*, if there exists some anchor between $\$_0$ and $\$_1$ in the compressed text, and otherwise it is *internal.* We handle these as separate cases.

Overlapping Case: First sum the counts for all words that are completely contained within the given temporal range. There can be no more than d of these, so this summation takes at most d time. Next, the count of the suffix of the requested range, which is the prefix of the word that starts just after t_1 and ends at $\$_1$, is added to the sum. By prefix-completeness, this prefix is stored on its own in the trie. The prefix count can be efficiently retrieved in $O(1)$ time, given a pointer to the leaf node associated with $\$_1$, the length of the prefix, and the data structure of [5] for answering level-ancestor queries.

Finally, the prefix of the requested range, which is the suffix of the word w_0 beginning at $\$_0$, is added to complete the sum. The suffix count is calculated by first looking up the prefix of w_0 that ends just before t_0, and subtracting its count from w_0's total count. This prefix count is computed exactly as in the previous paragraph.

Internal Case: The dictionary word is subdivided into three non-overlapping sections based on the range query; the prefix, the query region, and the suffix. Due to prefix-completeness, the count for the prefix is recorded in the annotated dictionary. It can be retrieved in $O(1)$ time, as above, using the level ancestor algorithm [5]. Similarly, the count for the word resulting from the concatenation of the prefix and the query region is also in the dictionary and can be retrieved in $O(1)$ time. Subtracting this count from the total word count results in the count for the suffix. Subtracting the suffix and prefix counts from the total word count gives the count for the query region, as desired.

The query time, once given a specific temporal range, is $O(d + \log d)$. In order to reduce the query time, we supplement the data structure for the overlapping case so that d words are never summed individually, but rather are looked up in an aggregation tree (of size $O(d)$) from which we use the largest component subtrees. The aggregation tree is a binary tree containing word sums as leaf nodes and aggregate values of all words in the associated subtree for each internal node. Using this data structure, the number of summed subtrees is $O(\log d)$. With this modification, the running time is dominated by the $O(\log d)$ time needed to lookup which word(s) overlap the given temporal query using a binary search over the sorted anchor points, and the $O(\log d)$ complete words that might be summed using the aggregation tree for overlapping queries. $\square$

Finally, we consider the total number of bits of space used in this process. The storage of the anchor points requires space d and the annotated dictionary takes space d. Under our assumption that the group is of fixed size, the largest sum that can be achieved during this process is $O(T)$. These sums annotate dictionary words, so the modified dictionary takes space at most $O(d \log T)$, which is $O(d \log d)$ since $T = O(d^2)$. In addition, we make use of an auxiliary data structure to solve the level ancestor problem [5]. This data structure requires storage only of the tree, $O(d)$ pointers to nodes in the tree, and a table of $O(d)$ encoded subtrees that each take $O(\log d)$ space. Thus, the total size required by this auxiliary data structure is also $O(d \log d)$.

Lemma 3.3. *The total space in bits required for our temporal range structure in the group setting is* $O(d \log d) = O(\mathrm{Enc}(X))$.

3.2 Semigroup Setting

The results from the previous section hold only for group operations. Specifically, they do not hold for queries such as “max” and “min.” In this section we generalize these results to the semigroup setting. In order to handle semigroup operations, for a substring in a given temporal range we need to be able to return the aggregated result in $O(\log d)$ time without relying on subtraction. The additions explained here are only used to handle the remaining suffix needed for the overlapping case or for lookup in the internal case. In other words, we will only be using this auxiliary data structure when considering queries over time periods within a single word.

We base our auxiliary data structure on the Sleator and Tarjan link-cut tree [32]. They annotate edges along the tree to be either solid or dashed so that any path from the root to a leaf node has $O(\log d)$ dashed edges and any solid path may have as many as $O(d)$ edges. Each solid path is additionally annotated with a binary tree, so that any node may still be reached through a path of $O(\log d)$ edges. This binary tree’s nodes represent edges and are annotated with their edge’s cost. We augment the link-cut tree to additionally include the aggregated cost of the subpath represented by the node for each node in the binary trees associated with solid paths (see Figure 3.2). With these additions, the data structure still takes space $O(d \log d)$.

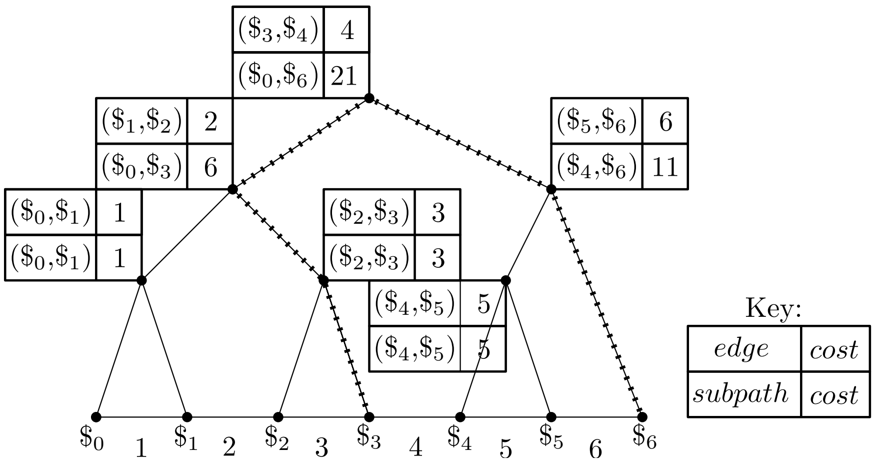

Figure 2: A solid path between anchors $\$_0$ and $\$_6$ along sensor output string "123456" with associated binary tree. Each node of the binary tree is annotated with its associated edge cost and its associated subpath cost. The search paths when finding the cost of the subpath between anchors $\$_3$ and $\$_6$ is shown with a dotted line. The cost of this subpath is found to be 0 for the path from $\$_3$ plus 11 for the path from $\$_6$ plus 4 for least common ancestor, for a total cost of 15.

To retrieve an aggregated value when given pointers to string endpoints that are fully contained within a solid tree path, start at each endpoint's corresponding leaf node and traverse the path to their least common ancestor. While traversing from the left endpoint, at each parent node that is not the least common ancestor of the endpoints, if the path up the tree goes from a left child to the parent, add the subpath cost stored at the parent to the running total. If the path up the tree goes from a right child to the parent, no cost is added in that step. Proceed symmetrically for the right endpoint. Sum the resulting paths and the least common ancestor's edge cost. This step takes time $O(\log d)$, or the depth of a solid path's binary tree. For an example, see Figure 3.2.

In order to combine solid paths with dashed paths, recall that since we only need to handle queries within a single word in this section, both endpoints of the substring must be on a single path to the root. We traverse from the given bottom most pointer up the tree until we reach the corresponding ending pointer. The cost of all dashed edges on this path are added to the sum, while solid path segments are handled as described above and the resulting sum is added to the total. There are at most $\lfloor \log d \rfloor$ dashed edges on the path from any vertex to the root if we make solid vs. dashed edge choices based on the number of nodes in vertex subtrees [32], and traversing any solid path segment takes time in the depth of the binary tree, so this step takes time $O(\log d)$.

Finally, note that the endpoint nodes can be identified in the tree by considering $t_0 - \$_0$ and $\$_1 - t_1$ since the queries are along a single path that is indicated by $\$_0$ and $\$_1$. Annotate the leaf nodes in the solid path binary trees with their numeral positions in the path. Combining the navigation through these trees with following the dashed edges along the identified path, the endpoints of the substring can be found in $O(\log d)$ time. In total, modifying the query procedures to handle semigroup operations maintains the query time of $O(\log d)$.

Lemma 3.4. *The query time for temporal range searching in the semigroup setting is $O(\log d) = O(\log T)$.*

4 Spatio-temporal Range Searching

In this section we consider how to extend the results of the previous section on temporal range searching on a single string to range searching for a sensor system, in which queries include both the spatial and temporal components of the data. We assume that we are given an m-local sensor system with S sensors. Each sensor is identified with its location p_i in space and a stream X_i of occupancy counts over some common time interval $[1, T]$. We assume that the sensors reside in real d-dimensional space, $\mathbb{R}^d$, where d is a constant. Our approach can be generalized to metric spaces with constant doubling dimension. We model each sensor's location as a point, and the answer to a range query consists of the sensors whose associated point lies within the query region. Let P and $\mathbf{X}$ denote the sets of sensor locations and observation streams, respectively.

Define a *spatio-temporal range query* to be a pair $(Q, [t_0, t_1])$ consisting of a geometric query range Q from some space $\mathcal{Q}$ of allowable ranges (e.g., rectangles, balls, or halfspaces) and a time interval $[t_0, t_1] \subseteq [1, T]$. The problem is to compute the sum of the occupancy counts of the sensors whose locations lie within the range, that is, $P \cap Q$, over the given time interval. In general, the occupancy counts are assumed to be drawn from a commutative semigroup, and the sum is taken over this semigroup. The remainder of this section is devoted to proving the following theorem, which shows that approximate spherical spatio-temporal range queries can be answered efficiently. In fact, these techniques hold for all fat convex ranges, but for simplicity of presentation we will limit ourselves to the spherical case here.

Theorem 4.1. *There exists a data structure for answering ε-approximate spatio-temporal spherical range queries for an S-element m-local sensor system $\mathbf{X}$ in $\mathbb{R}^d$ for all sufficiently long time intervals T with preprocessing time $O(\mathrm{Enc}(\mathbf{X}))$, query time $O(((1/\varepsilon^{d-1}) + \log S) \log T)$, and space $O(\mathrm{Enc}(\mathbf{X}) \log S)$ bits.*

Rather than considering a particular range searching problem, we will show that the above problem can be reduced to a generalization of classical range searching. To motivate this reduction, we recall that the compression algorithm presented in [12] groups sensors into clusters, and the sensor outputs within each cluster are then compressed jointly. In order to answer range queries efficiently, it will be necessary to classify each such cluster as lying entirely inside the range, outside the range, or overlapping the range's boundary. In the last case, we will need to further investigate the cluster's internal structure. Efficiency therefore is dependent on the number of clusters that overlap the range's boundary. We will exploit spatial properties of the clusters as defined in [12] to achieve this efficiency. To encapsulate this notion abstractly, we introduce the problem of *range searching over clumps*, in which the points are replaced by balls having certain separation properties. Eventually, we will show how to adapt the BBD-tree structure [4] to answer approximate range queries in this context.

Given any metric space of constant dimension, a *set of clumps* is defined to be a finite set C of balls that satisfies the following *packing property* for some constant γ (depending possibly on dimension): Given any metric ball b of radius r, the number of clumps of C of radius r' that have a nonempty intersection with b is at most $O((1 + (r/r'))^{\gamma})$. Given a range shape Q, a clump may either lie entirely within Q, entirely outside Q, or may intersect the boundary of Q. In the last case, we say that the clump is *stabbed* by Q. See Figure 3 for examples of these cases.

The relevance of the notion of clumps to our setting is established in the following lemma. The lemma states that the clusters of sensors within a single partition created by the *PartitionCompress*

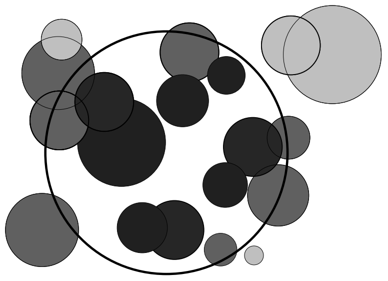

Figure 3: A set of clumps and a spherical range. The query range is indicated by the larger disc. The clumps are indicated by smaller balls and are shaded based on their membership in one of three groups; clumps that are outside of the range are light grey, clumps that are stabbed by the range are grey, and clumps that are entirely included in the given range are dark grey.

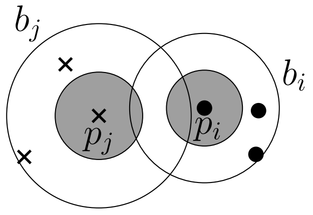

Figure 4: Two clumps b_i and b_j from a partition P' generated by *PartitionCompress*. Points represent sensor locations. The clump b_i centered at p_i contains cluster P'_i represented by the solid points. The clump b_j centered at p_j contains cluster P'_j represented by the points marked with a cross. The shaded disc centered at p_i is $\frac{1}{2}b_i$ and the shaded disc centered at p_j is $\frac{1}{2}b_j$.

algorithm of [12], when associated with a bounding ball, form a set of clumps. The *PartitionCompress* algorithm partitions the sensor point set P into a constant number of *groups*, $P_1, \ldots, P_c$ (where c depends only on the dimension of the space). Each group P_i is further partitioned into subsets, called *clusters*, such that if two sensors are in different clusters then their outputs are independent of each other. Given a ball b and real $\varphi > 0$, let $\varphi\, b$ denote the ball concentric with b whose radius is a factor of φ times the radius of b. The proof of the following lemma relies on the observation that $\frac{1}{2}b_1, ..., \frac{1}{2}b_h$ are pairwise disjoint. This is established based on the geometric properties of the repetitive partitioning process of the *PartitionCompress* algorithm. See Figure 4 for an accompanying diagram.

Lemma 4.1. *Given a point set P, let $P' \subseteq P$ be any of the groups generated by the* PartitionCompress *algorithm, and let $P'_1, \ldots, P'_h$ denote the associated set of clusters for this group. Then there exists a set of balls $C = \{b_1, \ldots, b_h\}$ that form a set of clumps such that $P'_i \subseteq b_i$.*

Proof. For the sake of completeness, let us first recall how the set P' is formed by the *PartitionCompress* algorithm. Initially all the points of P are unmarked. The algorithm repeatedly selects the unmarked point $p_i \in P$ that has the smallest m-nearest neighbor ball (with respect to the entire point set P). Let b_i denote this ball and let $P'_i = P \cap b_i$. These points are removed from P, and all the points of P lying within $3b_i$ of p_i are marked. This process is repeated until no unmarked point of P remains. Let h denote the number of iterations until termination, and let C denote the resulting set of balls. Let $P' = P'_1 \cup \ldots \cup P'_h$. (This produces one group. To form the next group, the process is then applied recursively to the points of P that were removed. This is all repeated until every point of P has been assigned to some group. See [12] for further details.)

We assert that, for $1 \leq i \leq h$, the balls $\frac{1}{2}b_1, \ldots, \frac{1}{2}b_h$ are pairwise disjoint. Consider any pair i, j, where $1 \leq i < j \leq h$. Let r_i and r_j denote the radii of b_i and b_j, and let p_i and p_j denote their respective centers. Since when p_i is being processed, all the points lying within distance $3r_i$ are marked, and since only unmarked points are chosen as centers of the balls, we have $\|p_i p_j\| \geq 3r_i$. Also, since the ball of radius $\|p_i p_j\| + r_i$ centered at p_j contains the m-nearest neighbor ball of p_i, it follows that this ball contains strictly more than m points, from which we conclude that $r_j \leq \|p_i p_j\| + r_i$. Combining these observations we have

$$\begin{aligned} \frac{r_i}{2} + \frac{r_j}{2} &\leq \frac{1}{2}(r_i + (\|p_i p_j\| + r_i)) \;\leq\; \frac{1}{2}(\|p_i p_j\| + 2r_i) \\ &\leq \frac{1}{2}\left(\|p_i p_j\| + \frac{2}{3}\|p_i p_j\|\right) \;<\; \|p_i p_j\|. \end{aligned}$$

Because the sum of their radii is less than the distance between their centers, it follows that the balls $\frac{1}{2}b_i$ and $\frac{1}{2}b_j$ are pairwise disjoint, and this completes the proof of the assertion.

To see that C is a set of clumps, consider any positive real r and r'. Let b be any ball of radius r, and let C' denote the subset of balls of C whose radius is at least r'. By the above assertion, the centers of any two balls of C' must be at distance at least r' from each other. By basic properties of doubling spaces, it follows that b can be covered by $O((1 + r/(r'/2))^d)$ balls of radius $r'/2$. Clearly, each ball of this set can contain the center of at most one ball of C'. Therefore, $|C'| = O((1 + (2r/r'))^d) = O((1 + (r/r'))^{d+1})$. Setting $\gamma = d + 1$ completes the proof. $\square$

We define the problem of *range searching among clumps* as follows: Given a space $\mathcal{Q}$ of allowable ranges and a set C of clumps, each of which is associated with a numeric weight from some commutative semigroup, preprocess the clumps into a collection of subsets, called *generators*, such that given any query range $Q \in \mathcal{Q}$, it is possible to report (1) a subset of these generators that form a disjoint cover of the clumps lying wholly within Q and (2) the subset of clumps that Q stabs. The total space requirements of a data structure for the range searching problem over clumps is the sum of space needed to represent the generators and the clumps, together with the space needed for storing the index structure needed to answer queries. The query time includes number of generators and stabbed clumps returned, plus the time to compute them.

Many data structures used in range searching are based on partition trees [1]. In such data structures, space is recursively subdivided into regions and the points are partitioned among these regions, until each region contains a single point. Each node of the tree is associated with a generator corresponding to the elements of the point set that lie in the leaves descended from this node. Our main result shows that, given a partition-tree based solution to the problem of range-searching among clumps, we can use such a structure to answer spatio-temporal range queries. This is done by adding an auxiliary data structure to each of the nodes of the tree to answer the temporal queries.

Lemma 4.2. *Suppose that we have a partition-tree based data structure that, given a set C of n clumps, can answer range queries over a query space $\mathcal{Q}$ with preprocessing time $pp(n)$, query time $qt(n)$, space $sp(n)$ bits, and has height $h(n)$. Then there exists a data structure that can answer spatio-temporal range queries for an m-local sensor system $\mathbf{X}$ of size S over a range space $\mathcal{Q}$ and time interval of length T with preprocessing time $O(h(S) \cdot pp(S) + \mathrm{Enc}(\mathbf{X}))$, query time $O(qt(S) \cdot \log T)$, and space $O(sp(S) + h(S) \cdot \mathrm{Enc}(\mathbf{X}))$ bits.*

Proof. We first run the *PartitionCompress* algorithm on the point set P of the S sensor locations. Recall that this partitions P into $O(1)$ groups, which by Lemma 4.1 can each be represented by a collection of clumps. We build a range searching structures for each of the resulting set of clumps. We will answer each query by invoking the range search separately on each of the individual structures, and then summing the results. Henceforth, we consider just the processing of a single group, which we will denote by P'.

We augment the clump range-search structure for P' by building one temporal range search structure for each of the individual clumps of P' as well as for each of the generators, that is, for each of the internal nodes of the associated partition tree. First, recall that each clump consists of the sensor streams for some number $m' \leq m$ of sensors. For each clump we treat the data from this clump as a time stream whose elements are m'-element vectors, where the ith element of the vector is the count of the ith sensor. We compute a temporal range search data structure for the associated stream of vectors (where the semigroup sum is extended to the semigroup sum over vectors). Next, for each node u of the tree, let g_u denote the associated generator consisting of the points $\{p_1, \ldots, p_f\}$ stored in the leaves that are descended from u. Let $\{X_1, \ldots, X_f\}$ denote the corresponding set of the sensor streams. Let X_u be the aggregated stream $\sum_{i=1}^{f} X_i$, formed by taking the componentwise sum of the observations from all f streams. (Unlike the clump case, we collapse all the sensors counts into a single sum, rather than creating an f-element vector. This is because f may generally be as large as the total number of sensors.) We build a temporal range searching structure for X_u, and associate this auxiliary tree with u.

Next, let us consider how to answer a given spatio-temporal query $\langle Q, [t_0, t_1] \rangle$ over P'. We first apply the range searching data structure for Q over the set of clumps associated with P'. Recall that this returns (1) a subset of generators lying within Q and (2) the clumps that are stabbed by Q. The former set may be assumed to be associated with a set of internal nodes of the tree and the latter with a set of leaf nodes of the tree. For each node u of (1), we invoke the corresponding auxiliary temporal data structure over the aggregated stream X_u and the time interval $[t_0, t_1]$, and include the resulting semigroup sum in the final total. For each each leaf node of (2), we invoke the associated auxiliary temporal range search structure for time interval $[t_0, t_1]$ to determine the semigroup vector sum over this interval. For each sensor of the clump we determine whether it lies within Q, and if so, we include its component of the vector sum in the final total.

The space used by the data structure is equal to the total space $sp(S)$ for the range searching structure over clumps, plus the space needed for the temporal range search structures for each of the clumps and each of the generators. To bound this quantity, consider the $h(S)$ levels of the tree. Each of the nodes of this level is associated with a generator, such that each sensor stream contributes to at most one node of the level. It can be shown by basic properties of entropy that the entropy of the componentwise sum of the stream is not greater than the sum of entropies of the sensor streams at the leaf level, which is at most $\mathrm{Enc}(\mathbf{X})$ bits. Summing over $h(S)$ levels yields the desired space bound. Similarly, the preprocessing time of the data structure is just the preprocessing time needed to build the range searching structure over clumps, plus the time needed to construct the individual auxiliary temporal range search structures.

To bound the total query time, observe that the query time is dominated by the time $O(qt(S))$ to compute the set of nodes whose associated clumps and generators form the answer to the query, together with the $O(\log T)$ time from Lemma 3.1 to access each auxiliary data structure to answer the temporal range queries. This completes the proof. □

We claim that many standard partition-tree-based methods for approximate range searching can be adapted to perform range searching among clumps. Observe that we can generalize the notion of ε-approximate range searching to approximate range searching over clumps. To do so we define two ranges Q^- and Q^+, representing the inner and outer approximate ranges. For example, in the case of spherical range searching, given a query ball Q, we define $Q^- = Q$ and Q^+ to be the ball concentric with Q but whose radius is scaled relative to Q's radius by a factor of $(1+\varepsilon)$. (See Arya and Mount [4] for further details.) If a generator lies entirely within Q^+ its points may be counted as lying within the approximate range, if it lies entirely outside of Q^-, its points may be considered to lie outside the approximate range. A clump is classified as being stabbed by Q if and only if it has a nonempty intersection with both Q^- and the complement of Q^+. It is easy to show that such a clump has diameter $\Omega(\varepsilon \cdot \mathrm{diam}(Q))$ [4]. By the packing property of clumps, the number of such clumps is $O(1/\varepsilon^\gamma)$, where the parameter γ depends only on the dimension of the space. (It may seem odd to consider approximate range searching in light of our emphasis on lossless compression. However, the data structures that we will describe below have the property that the value of ε is specified at query time, and it may even be that $\varepsilon = 0$. Thus, the data represented by the sensors may be extracted to any desired degree of precision.) We conclude by remarking that it is relatively easy to generalize many standard approximate range searching data structures based on hierarchical partitioning to answer range searching over clumps. We present one example based on the BBD-tree data structure of [4].

Many data structures, such as the range searching algorithm appearing in [4] for approximate spherical range searching, exploit packing properties to achieve efficiency. Our next result shows that, through a straightforward adaptation of the algorithm presented there, it is possible to answer such queries in the context of clumps.

Lemma 4.3. *There exists a data structure for answering ε-approximate spherical range searching queries over a set C of n clumps in $\mathbb{R}^d$ with preprocessing time $O(n \log n)$, query time $O((1/\varepsilon^{d-1}) + \log n)$, and space $O(n \cdot (prec(C) + \log n))$ bits, where $prec(C)$ denotes the maximum number of bits of precision in the geometric coordinates used to define C.*

Proof. Recall from [4] that a BBD-tree for a set of n points is type of balanced and compressed quadtree decomposition, in which the tree has size $O(n)$ and height $O(\log n)$. In order to guarantee that the tree has logarithmic depth, in addition to the standard quadtree splitting operations there is a decomposition operation, called *centroid shrinking.* Define a *quadtree box* to be any axis-parallel hypercube that can be formed by starting with the unit hypercube, and repeatedly splitting it into 2^d congruent subcubes, by passing d axis-parallel hyperplanes through the center of the cell. Given a quadtree box b, this operation computes a nested quadtree box $b' \subseteq b$ such that a constant fraction of the points of b lie within b'. Corresponding to this operation there is a special node of the tree, called a *shrink node*, whose two child nodes are associated with b' and $b \setminus b'$, respectively. Each node of the tree is naturally associated with a region of space, called its *cell.* The cell associated with each node of the BBD tree is either a quadtree box or the set-theoretic difference of two quadtree boxes, one nested within the other. (See [4] for further details.)

The BBD-tree data structure is generalized to process range searching over clumps as follows. First, we assume that the sensor points have been scaled so they lie within a unit hypercube. The center points of the clumps are inserted into the BBD-tree, just as in [4], with the following exception. Let u denote a node of the clump-based BBD-tree, let q denote the cell associated with u, and let s denote b's maximum side length (also called its size). Each clump whose center lies within q and whose radius lies between $s/2$ and s is stored in a special leaf node which is made a child of u. It is easy to establish the invariant that the descendants of any node whose cell size is s are clumps of radius at most s. By Lemma 4.1, the number of such leaves per node is $O(1)$. Otherwise, the preprocessing is identical to that of the BBD-tree. The preprocessing time is essentially the same as that of the BBD-tree, which is $O(n \log n)$. The space is equal to the total space needed for the point coordinates, which is $O(n \cdot \mathrm{prec}(C))$ bits, and the total space needed for the tree and its pointers, which is $O(n \log n)$ bits.

In order to answer a query, we follow essentially the same searching procedure given in [4], but with a few differences. Recall that the algorithm recursively descends the tree starting at the root. On its arrival at some leaf node u, we test whether the associated clump lies within Q^+ (in which case we include it in the set of generators lying within Q^+) or is stabbed by the annulus $Q^+ \setminus Q^-$ (in which case we include it among the stabbed clumps). On arrival at an internal node u, let q_u denote the associated cell and let s_u denote its size. Since the clumps lying within u have radius at most s_u, we check whether q_u dilated by s_u lies entirely within Q^+, and if so we include the associated generator among those lying within the range query. On the other hand, if the dilation of q_u lies outside of Q^-, we ignore the associated generator. If neither of these cases holds, then we recursively apply the search to the children of u.

By a straightforward adaptation of the packing arguments given in [4], it follows that the number of nodes visited by this algorithm is $O((1/\varepsilon^{d-1}) + \log n)$. $\square$

By applying Lemma 4.2 to the above data structure, it follows that we can answer ε-approximate spherical range searching queries for a sensor system of size S over a time period of length T with preprocessing time $O((S \log^2 S) + \mathrm{Enc}(\mathbf{X}))$, query time $O(((1/\varepsilon^{d-1}) + \log S) \log T)$, and space $O((S \cdot (\mathrm{prec}(C) + \log S) + \mathrm{Enc}(\mathbf{X}) \log S)$ bits, where $\mathrm{prec}(C)$ denotes the maximum number of bits of precision in the geometric coordinates used to define C. Under the assumption that T is sufficiently large that the encoding space dominates over time-invariant quantities, this completes the proof of Theorem 4.1.

Because the above result relies only on basic packing properties of the BBD-tree data structure, it is easy to see that this result can be applied to other data structures for range searching that rely only on such packing properties. Examples include the BAR-trees [DGK01], the quadtree-based data structure of Chan [6], quadtree-based solutions to absolute range searching [8], and methods based on net-tree decompositions in metric spaces of constant doubling dimension [14, 17].

5 Experimental Results

In addition to the theoretical analysis of the range searching results presented here, we evaluated the temporal range searching structure experimentally. Using a data set provided by the Mitsubishi Electronic Research Laboratory (MERL) [34] consisting of activation times for sensors located in the hallways of their building, we analyzed three aspects of our data structure's performance; locality, space, and time. In short, we found that our assumption that sensors closer to each other are more likely to have similar outputs was correct and that our data structure was able to use less space than a naive structure while providing faster query times. In the rest of this section, we describe the data set and our experimental methods in greater detail.

The MERL data set consists of activation times, representing people moving, for 213 sensors. These activation times are given with epoch start and end times. Using these start times, and

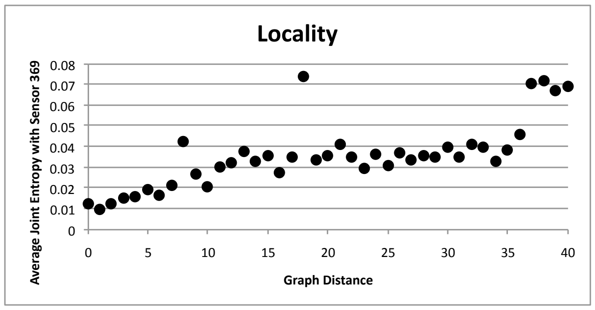

Figure 5: Locality shown via a comparison of the graph distance between sensor 369 and other sensors and the average joint entropy between all such pairs of sensors. As the distance between the sensors increases so does the joint entropy, showing that closer sensors are more likely to have related outputs.

noting that each activation lasted approximately one second, we translated these activation times into streams of data for each sensor in the form described earlier. Each activation is represented by a count of one and seconds in which no activations were reported are represented by a count of zero. These streams are associated with sensors whose locations are known and relationships are shown in a map of the building. Using this map, we create a graph representation in which neighboring sensors are connected by an edge with weight one. Sensors observing adjacent areas of the hallway or hallway areas connected by doors to observed rooms or lounges are considered to be neighboring.

In order to evaluate our assumption that nearby sensors are more likely to have related outputs, we compared the distance between a single sensor (sensor 369, a sensor in the middle of a hallway) to the pairwise joint entropy of that sensor and all other sensors' outputs. Distance was computed as the shortest path distance within the neighborhood graph described earlier, and the joint entropy considered was an empirical generalization of joint entropy [13] with a window size of 10 seconds. The average joint entropy for each distance is shown in Figure 5. The graph shows that those sensors in the neighborhood near sensor 369, those less than distance five away, have outputs with lower joint entropy. After this local neighborhood, the joint entropy raises to a relatively constant threshold for the majority of distances, and finally raises again for far away sensors. The outlying points at distances 8 and 18 represent comparisons with sensors in the unusual areas near the elevators and lunch room, respectively.

We considered the storage space per sensor data stream for the raw data (consisting of one value per second) versus the temporal range structure (specifically the annotated trie) written to a file. The average size taken over all sensors by each of these methods as it varies based on the number of days (in increments of 10) of data can be seen in Figure 5. We call the ratio between the space used by the raw data and the space used by the temporal range structure the *improvement ratio*. A graph showing the number of days of data considered versus this improvement ratio is given in Figure 5. The improvement ratio increases as the amount of data increases, ranging from a 14-fold improvement for 1 day to a 66-fold improvement for 80 days of data. This increase is likely caused by the observation of repeated patterns; the first observation must be stored in the annotated trie while later observations can simply extend existing patterns, taking less space.

Query time is considered for varying query interval lengths for 150 days of data (at 1 day intervals). We compare our temporal range searching method to the naive method that aggregates by linearly adding each count. Query times do not include the time to read in the file or, in the case of the temporal range structure, the one-time preprocessing cost. A graph showing the interval length versus the query time for each of these methods is given in Figure 7. Each query time depicted on the graph represents the average of 100 randomly chosen queries of the given interval

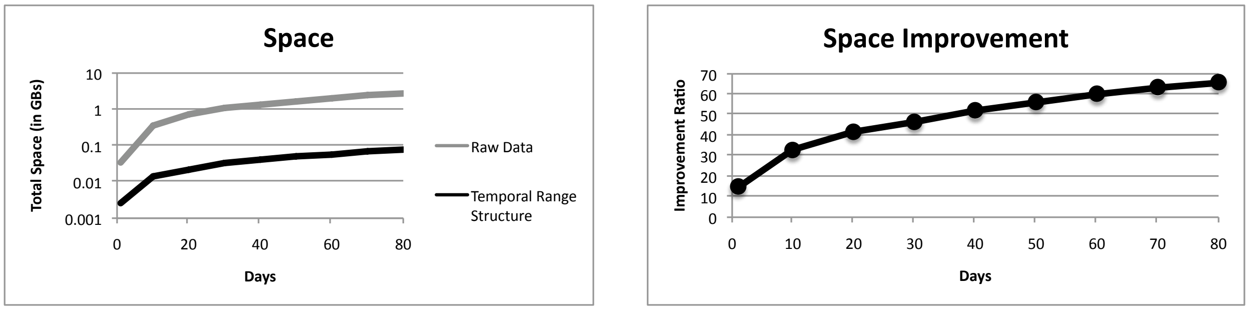

Figure 6: *Left:* The space used by the raw data in comparison to that used by the temporal range structure shown for varying numbers of days of data. Note that the size is shown in a logarithmic scale. *Right*: Space savings shown via a comparison of the number of days of data versus the ratio between the raw data and the temporal data structure sizes. As the number of days increase, the space saving increases as well.

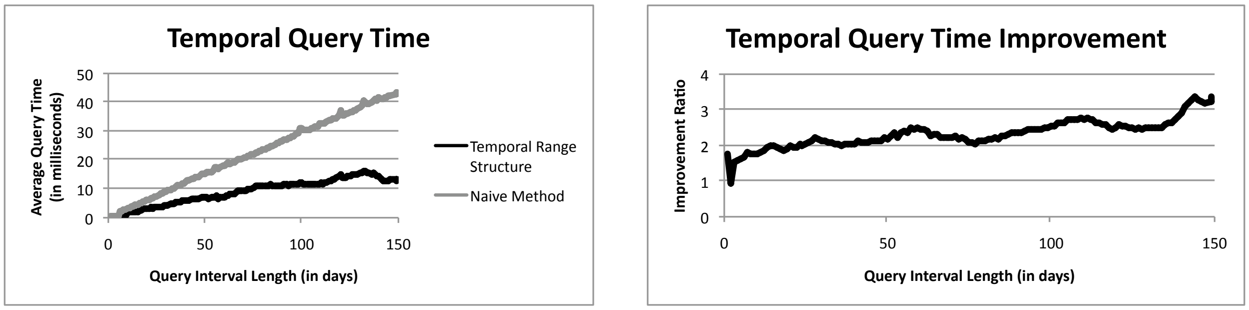

Figure 7: *Left*: Average temporal query times for 100 randomly chosen queries for each interval length. Query times using the temporal structure for aggregation over this query interval and using a naive method that simply aggregates one by one are shown. *Right*: The improvement ratio of the temporal query times, the naive time divided by the temporal range structure query time, is shown for each query interval from the corresponding temporal query times graph.

length. As the interval length increases, the temporal range structure's improvement over the naive method increases as well. The improvement ratio (the ratio of the naive query time to the temporal range structure time) shows that for most query intervals, the temporal range structure is twice as fast as the naive method and for larger intervals this improvement increases to fifteen times faster (see Figure 7). The improvement increase is likely due to the large aggregation possible for larger interval lengths.

6 Conclusion

In this paper we presented the first spatio-temporal range searching structure for kinetic sensor data. This structure operates over the compressed version of the data without the need to decompress it. Preprocessing time and the data structure's space, measured in bits, were shown to be on the order of the encoding size. The query time was shown to be logarithmic in the observed length of time. Experimental results showed that the space used was at least an order of magnitude better than the space used by the raw data and that the query time was less than that of a naive method.

The theoretical and experimental results presented here were analyzed under main memory assumptions. While we expect much of this analysis to transfer directly to an I/O-efficient model, new data structures may also be required. For future work, it would be interesting to extend these analyses to an I/O-efficient model and conduct experiments using these modified data structures.

References

[1] P. K. Agarwal and J. Erickson. Geometric range searching and its relatives. In B. Chazelle, J. Goodman, and R. Pollack, editors, *Advances in Discrete and Computational Geometry*, pages 1–56. American Mathematical Society, 1998.

[2] I. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. Wireless sensor networks: A survey. In *Computer Networks*, pages 393–422, 2002.

[3] A. Amir, G. Benson, and M. Farach-Colton. Let sleeping files lie: Pattern matching in Z-compressed files. *J. Comput. Syst. Sci.*, 52(2):299–307, April 1996.

[4] S. Arya and D. M. Mount. Approximate range searching. *Computational Geometry: Theory and Applications*, 17:135–152, 2000.

[5] M. A. Bender and M. Farach-Colton. The level ancestor problem simplified. *Theoretical Computer Science*, 321:5–12, 2004.

[6] T. Chan. Approximate nearest neighbor queries revisited. *Discrete & Computational Geometry*, 20:359–373, 1998.

[7] S.-Y. Chiu, W.-K. Hon, R. Shah, and J. S. Vitter. I/O-efficient compressed text indexes: From theory to practice. In *Data Compression Conference*, pages 426–434, 2010.

[8] G. D. da Fonseca and D. M. Mount. Approximate range searching: The absolute model. *Computational Geometry: Theory and Applications*, 43:434–444, 2010.

[9] P. Ferragina, F. Luccio, G. Manzini, and S. Muthukrishnan. Compressing and searching XML data via two zips. In *Proc. of the 15th International Conference on World Wide Web*, pages 751–760, 2006.

[10] P. Ferragina and G. Manzini. Indexing compressed text. *Journal of the ACM*, 52(4):552–581, July 2005.

[11] P. Ferragina and R. Venturini. A simple storage scheme for strings achieving entropy bounds. *Theoretical Computer Science*, 372(1):115–121, March 2007.

[12] S. A. Friedler and D. M. Mount. Compressing kinetic data from sensor networks. In *Proc. of the Fifth Intl. Workshop on Algorithmic Aspects of Wireless Sensor Networks (AlgoSensors)*, pages 191 – 202, 2009.

[13] S. A. Friedler and D. M. Mount. Realistic compression of kinetic sensor data. Technical Report CS-TR-4959, University of Maryland, College Park, 2010.

[14] J. Gao, L. J. Guibas, and A. Nguyen. Deformable spanners and applications. *Computational Geometry: Theory and Applications*, 35:2–19, 2006.

[15] R. González and G. Navarro. Statistical encoding of succinct data structures. *Combinatorial Pattern Matching*, pages 294–305, 2006.

[16] A. Guitton, N. Trigoni, and S. Helmer. Fault-tolerant compression algorithms for sensor networks with unreliable links. Technical Report BBKCS-08-01, Birkbeck, University of London, 2008.

[17] S. Har-Peled and M. Mendel. Fast construction of nets in low dimensional metrics and their applications. *SIAM J. Comput.*, 35(5):1148–1184, 2006.

[18] W.-K. Hon, R. Shah, and J. S. Vitter. Compression, indexing, and retrieval for massive string data. In *Proc. of the 19th Ann. Conf. on Combinatorial Pattern Matching*, 2010.

[19] D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40, Sept. 1952.

[20] R. S. Kosaraju and G. Manzini. Compression of low entropy strings with Lempel–Ziv algorithms. *SIAM J. Comput.*, 29(3):893–911, 1999.

[21] R. Krauthgamer and J. R. Lee. Navigating nets: Simple algorithms for proximity search. In *Symposium on Discrete Algorithms*, 2004.

[22] G. Manzini. An analysis of the Burrows–Wheeler transform. *J. ACM*, 48(3):407–430, May 2001.

[23] J. Matousek. Geometric range searching. *Comp. Surveys*, 26(4):422–461, 1994.

[24] MIT Media Lab. The Owl project. `http://owlproject.media.mit.edu/`.

[25] G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1), 2007.

[26] O. Procopiuc, P. K. Agarwal, L. Arge, and J. S. Vitter. A dynamic scalable kd-tree. In *Proc. International Symposium on Spatial and Temporal Databases*, volume LNCS 2750, 2003.

[27] J. Rissanen. Generalized Kraft inequality and arithmetic coding. *IBM Journal of Research and Development*, 20, 1976.

[28] J. Robinson. The K-D-B tree: A search structure for large multidimensional dynamic indexes. In *Proc. SIGMOD International Conference on Management of Data*, pages 10–18, 1981.

[29] H. Samet. The quadtree and related hierarchical data structures. *Computing Surveys*, 16(2), June 1984.

[30] N. Saunier and T. Sayed. Automated analysis of road safety with video data. In *Transportation Research Record*, pages 57–64, 2007.

[31] C. E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27:379–423, 623–656, July, October 1948.

[32] D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. *J. Comput. Syst. Sci.*, 26(3):362–391, 1983.

[33] B. J. M. Stutchbury, S. A. Tarof, T. Done, E. Gow, P. M. Kramer, J. Tautin, J. W. Fox, and V. Afanasyev. Tracking long-distance songbird migration by using geolocators. *Science*, page 896, February 2009.

[34] C. R. Wren, Y. A. Ivanov, D. Leigh, and J. Westbues. The MERL motion detector dataset: 2007 workshop on massive datasets. Technical Report TR2007-069, Mitsubishi Electric Research Laboratories, Cambridge, MA, USA, August 2007.

[35] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, IT-23(3), May 1977.

[36] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5):530–536, 1978.