Fast and Space-Efficient Top-k Queries on Time Series

James W. McClain *

Abstract

Given a collection of time series, one might want to know which of them have the greatest values at some instant in time. In this article we present a new algorithm to answer this question: A technique which answers such queries for general collections of one-dimensional time series using an asymptotically optimal number of I/O operations. Our approach is to calculate the moments in time when the roster of top series changes, and then to store the list of those events in an efficient manner. We show theoretically and with experiments that this approach produces search indices much smaller than those produced by previously-described approaches.

1 Introduction

In January 1815 the first-, second-, and third-most expensive stocks trading on the New York Stock Exchange were Merchants Bank, Manhattan Bank, and Mechanics' and Traders' Bank at $118, $117, and $116 per share, respectively[14].

The information given above is the answer to a top-k query over historical NYSE price data with k equal to three and the instant of interest equal to "January 1815". (The instant is an entire month because these historical data are only available at that granularity.) The ability to retrieve answers to such questions is of great practical usefulness, not just in finance and economics, but also in many other areas. How to efficiently compute answers to these types of queries is the subject of this work.

1.1 Problem Statement, Notation, Definitions

Adapting the notation and nomenclature used in [11], we consider a collection of functions $f_i : \mathbb{R} \to \mathbb{R}$ (from some real-valued time instant to some real-valued magnitude) and call them *time series*. When the data set $\{f_i\}$ is composed of sample points, m_i is the number of sample points in series f_i. We say that there are n such series, so the index variable i assumes n discrete values (for example, integers between 1 and n). We are concerned with computing the k time series which have the largest value at *time instant* t. In this article, we talk about how to build a *search index*, an on-disk data structure, to facilitate answering

* james.mcclain@gmail.com

such queries. The parameter $k_{\max}$ is the upper-bound on the maximum k for which the search index can answer a query.

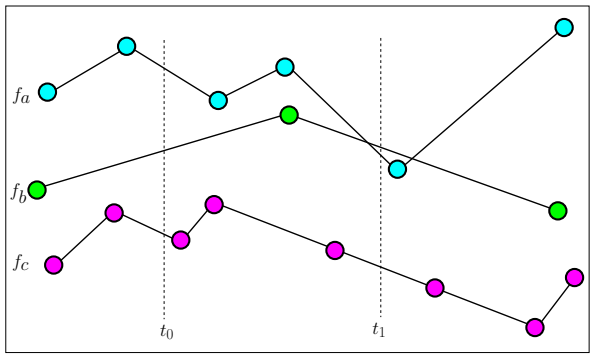


Figure 1: Notation illustration.

Please see Figure 1 for an illustration of the notation. In that picture, there are three time series: f_a, f_b, and f_c, so that $n = 3$. There, $m_a = 6$, $m_b = 3$, and $m_c = 8$. The dashed lines labeled t_0 and t_1 are instants in time. In the picture, $\text{top}_3(t_0) = (a, b, c)$, $\text{top}_3(t_1) = (b, a, c)$, and $\text{top}_1(t_1) = (b)$. A table of notation is given in Table 1.

Symbol	Meaning
$\{f_i\}$	The data set.
f_i	A time series.
m_i	The number of samples in f_i.
n	The number of time series in the data set.
t	An instant in time.
N	The number of samples in the data set, $\sum_i^n m_i$.
$k_{\max}$	The largest supported k for any top-k query.
$\text{top}_k(t)$	The top k series at time t.
M	The size of main memory.
B	The size of a disk block.
I	The size of an on-disk search index.

Table 1: Notation used in this article.

For discrete (or discontinuous) data, it is assumed that there is some given mechanism for interpolating over gaps. Many times, the method of interpolation is to draw straight segments between adjacent data points or segments. When the data set is composed of samples on the time series, we define $N := \sum_i^n m_i$ as the total size of the given data set.

Various results will be quoted in terms of the main memory size M, the size of a disk block B, the number of sample points N, and the size of an on-disk search index I.

Structure	Index Size	Construction Time	Query Time	Update Time
MVB-Tree[3, 11]	$\mathcal{O}\left(\frac{N^2}{B}\right)$	$\mathcal{O}\left(I \log \frac{N}{B}\right)$	$\mathcal{O}\left(\log_B \frac{N}{B} + \frac{k}{B}\right)$	*n/a*
R-Tree[8, 11]	$\mathcal{O}\left(\frac{N}{B}\right)$	$\mathcal{O}(I \log I)$	$\mathcal{O}\left(\frac{N}{B}\right)$	$\mathcal{O}\left(\log_B \frac{N}{B}\right)$
SEB-Tree[11]	$\mathcal{O}\left(\frac{N}{B}\alpha\left(\frac{N}{B}\right)\left(\sqrt{\log \frac{N}{B}} + \log \frac{k_{\max}}{B}\right)\right)$	$\mathcal{O}\left(I \log \frac{N}{B}\right)$	$\mathcal{O}\left(\log_B N + \frac{k}{B}\right)$	unknown
Wrld-Tree (static)	$\mathcal{O}\left(\frac{N}{B} \log_2 k_{\max}\right)$	$\mathcal{O}\left(I \log \frac{N}{B}\right)$	$\mathcal{O}\left(\log_B N + \frac{k}{B}\right)$	*n/a*
Wrld-Tree (bulk dyn.)	*do.*	*do.*	*do.*	$\mathcal{O}\left(\log \frac{N}{B} \log_2 k_{\max}\right)$
Wrld-Tree (dyn.)	*do.*	*do.*	$\mathcal{O}\left(\log_2 k_{\max}\left(\log_B N + \frac{k}{B}\right)\right)$	$\mathcal{O}\left(\log^2 \frac{N}{B} \log_2 k_{\max}\right)$

Table 2: A tabular summary of index sizes, construction times, query costs, and update costs for all known temporal top-k query algorithms. The value I in the "Construction Time" column is equal to the bound on the index size of the algorithm (the value in the cell to the left). All of the bounds shown are absolute except for the index size and construction time bounds of SEB-Tree, which are in expectation.

1.2 Contributions, Organization

In this article, we present Wrld-Tree, a new data structure indexing time series data for top-k queries. The advantages that this structure presents over previous approaches are three.

The first advantage is the small size of the index that the algorithm creates. Here, the size of the on-disk index is bounded above absolutely by $\mathcal{O}\left(\frac{N}{B} \log k_{\max}\right)$ disk blocks. That is smaller than the $\mathcal{O}\left(\frac{N}{B}\alpha\left(\frac{N}{B}\right)\left(\sqrt{\log \frac{N}{B}} + \log \frac{k_{\max}}{B}\right)\right)$ expected disk blocks required by the best previously-described approach, SEB-Tree[11]. We will see in Section 4 that on the synthetic and real data sets that we tested, Wrld-Tree produces a much smaller on-disk search structure.

Wrld-Tree also provides deterministic upper bounds on the construction cost of the index, the index size, and query cost (the last of which is optimal for static and bulk-dynamic versions of Wrld-Tree). The deterministic bounds given in this work can be contrasted to the bounds-in-expectation for SEB-Tree. Please see Table 2 for a tabular comparison of the index size, index construction time, query cost, and update cost of Wrld-Tree, SEB-Tree, and two other approaches.

The third contribution is that our methodology for doing top-k queries. The approach that we take is more "general" than any previously-presented. It can be used with time series represented in any way (piecewise constant, piecewise linear, piecewise continuous, symbolic, and so on), which is in contrast to MVB-Tree[3, 11] which requires that the series be piecewise constant, and is in contrast to the piecewise linear representation required by R-Tree[8, 11] and SEB-Tree[11].

Other previous and related works are discussed in Section 2. A description and discussion of Wrld-Tree, including analysis, is provided in Section 3. Experimental results are discussed in 4. We conclude the article in Section 5.

2 Previous and Related Work

The present work and SEB-Tree[11] are the only two algorithms of which we are aware that are targeted specifically at the problem of answering top-k queries on time series. The discussion of previous work in [11, §2] provides an extensive listing and discussion of works which address questions similar- or related-to our question. We recommend that section if a more extensive survey desired.

There were are only two solutions to the temporal top-k problem prior to SEB-Tree, both of which were adaptations of pre-existing data structures.

It is possible to use R-Trees[8] to do top-k queries on piecewise linear approximations of time series by viewing the segments of the data as ranges which can be stored in a range tree. It is also possible use MVB-Trees[3] to do queries on piecewise constant approximations of series by viewing the series as scalars which change over time (have multiple versions). The first of these solutions has potentially linear query times, while the second one could require an index of quadratic size.

By way of other related work, the Threshold Algorithm ("TA")[7] is a methodology that can be used to widen the applicability of the ideas that we share in this paper. TA allows multiple top-k databases to be combined together optimally so that new top-k lists, sorted by different aggregation functions, can be generated. TA requires that the primitive top-k lists are provided in sorted order, which the algorithm in Section 3.1.1 does. Additional recent work along these lines can be found in [9], where the authors improve the performance of larger aggregate queries.

3 Algorithms and Analysis

In this section, we present two algorithms for constructing static top-k search structures. Those discussions are found in Sections 3.1.1 and 3.1.2, the first is an initial step and the second is more efficient. A query algorithm that can be used with either of those structures is presented and analyzed in Section 3.1.3. A dynamic search structure and its accompanying query algorithm are presented in Section 3.2.

3.1 Static Structure

The approach that we take to answering top-k queries over the data set $\{f_i\}$ is to view $\text{top}_k(\cdot)$ as a tuple-valued function of one real parameter, then to try to store and evaluate this function as efficiently as possible.

If there are three time series f_a, f_b, and f_c, then the value of $\text{top}_3(t)$, for any value of the time parameter t, is some permutation of (a, b, c). Based on that, the simplest possible way to store the $\text{top}_3(\cdot)$ function associated with those three series would be to simply compute all of the contiguous intervals in time on which the function value does not change, *isovalued intervals*, and store those intervals in a B-Tree. The algorithm presented in the following subsection is based on that idea and almost that simple.

3.1.1 The First Static Algorithm

Instead of computing every contiguous interval on which the value of $\text{top}_k(\cdot)$ is constant and storing the tuples explicitly, the approach that we take in the first algorithm is to store some of the function values in compressed form by referring to the values immediately preceding them in time.

If the rankings of three time series are (a, b, c) and (b, a, c) on two adjacent isovalued intervals, the first interval ending and the second beginning at time t_0, then the time t_0 coincides with an instant at which f_a and f_b intersect each other (or cross each other if the series are not continuous).

That means that it is possible to record a change in the value of $\text{top}_k(\cdot)$ by noting the causal intersection and its time, say (a, b, t_0), instead of the entire length-k tuple (which would restate a great deal of information already present in the previous tuple).

Since the intersections can be noted using only constant space (assuming that the time values are of fixed-precision), and since any report of $\text{top}_k(t)$ must require at least $\mathcal{O}\left(\frac{k}{B}\right)$ disk operations, it makes sense to only explicitly store the tuple value associated with every $(k+1)^{\text{st}}$ isovalued interval and store only the *intersections* that occasion the next k intervals. The tuple and the following intersections are then packaged together as a single value and stored in a B-Tree. The key associated with that entry is the start time of the interval associated with the tuple. Please see Figure 2.

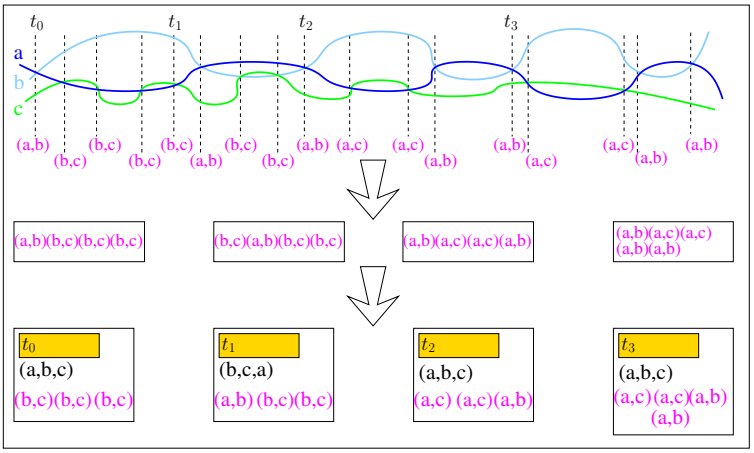


Figure 2: An illustration of the data structure. In this example, $k_{\max} = n = 3$. In the upper third, The data set with magenta intersections is shown (see Algorithm 1 line 1). In the middle third, the list of intersections is partitioned into chunks of size $k+1$ (see Algorithm 2 line 2 and let $2^j = k$). In the bottom third, the B-Tree entries, with their keys shown in gold, are created (see Algorithm 2 line 7).

The approach just described assumes a fixed value of k. The idea can be extended to work with any $k \leq k_{\max}$ by building a logarithmic number of these structures, where the value k is made to be 2^j and j ranges from 0 to $\lfloor \log_2 k_{\max} \rfloor$. When this structure is queried for a particular k, a ranking of length $\mathcal{O}(k)$ is returned.

The construction algorithm can be seen in pseudocode form in Algorithm 1. Please note that a few mild assumptions are made in the pseudocode for the purpose of simplicity, such as that the total number of intersections is divisible by $2^{\lfloor \log_2 k_{\max} \rfloor}$.

Algorithm 1 First construction algorithm.

Require: $\{f_i\}$, $k_{\max}$
1: E $\leftarrow$ all intersections in $\{f_i\}$, sorted by time.
2: **for all** $e \in E$ **do**
3: $t_e \leftarrow$ the time at which e occurs
4: **for all** $0 \leq j \leq \lfloor \log_2 k_{\max} \rfloor$ **do**
5: **if** $\text{top}_{2^j}(t_e - \varepsilon) \neq \text{top}_{2^j}(t_e + \varepsilon)$ **then**
6: STORE(j, e)
7: **end if**
8: **end for**
9: **end for**

On line 4 a comparison between the values of $\text{top}_{2^j}(\cdot)$ immediately before and after an intersection is performed. If the two sorted lists are identical, that means that the intersection does not effect $\text{top}_{2^j}(\cdot)$ and so it does not need to be considered for that value of j. However, if the intersection does effect the top-k for $k = 2^j$, then it is stored in the search tree associated with that j value.

Line 5 is where the storage of relevant intersections takes place. The storage subroutine called on that line is shown in pseudocode-form in Algorithm 2. The subroutine works by maintaining an (initially empty) list of 2^j intersections for each j value. When the subroutine is called and the list associated with j has 2^j or more entries, the subroutine creates a new entry in the appropriate B-TREE consisting of a tuple (of length 2^j) and 2^j intersection events (each of constant size), then empties the list.

Algorithm 2 The common storage subroutine.

1: **function** STORE(j,e)
2: **if** $|\texttt{list}_j| \geq 2^j$ **then**
3: $\texttt{tree}_j \leftarrow$ the $\text{top}_{2^j}(\cdot)$ B-TREE
4: $\texttt{key} \leftarrow$ the earliest instant of time in $\texttt{list}_j$
5: $\texttt{topk}_j \leftarrow \text{top}_{2^j}(\texttt{key})$
6: $\texttt{value} \leftarrow$ [$\texttt{topk}_j$, DROP$(1, \texttt{list}_j)$]
7: INSERT$(\texttt{tree}_j, \texttt{key}, \texttt{value})$ ▷ Store in B-TREE.
8: $\texttt{list}_j \leftarrow \emptyset$ ▷ Empty the list.
9: **end if**
10: $\texttt{list}_j \leftarrow$ CONJ$(\texttt{list}_j, e)$ ▷ Add e to the end of list.
11: **end function**

We can now establish a few facts about this algorithm.

Lemma 1. *If E is defined to be number of times that the sorted list $\text{top}_{k_{\max}}(t)$ changes as t ranges over $(-\infty, \infty)$, then Algorithm 1 produces an index of size $\mathcal{O}\left(\frac{E}{B}\right)$ disk blocks (in expectation).*

Proof. A B-Tree containing m items of constant size requires $\mathcal{O}(m)$ space. Algorithm 1 produces $\mathcal{O}(\log k_{\max})$ B-Trees, the largest one containing $\mathcal{O}(E)$ items. Those trees have sizes distributed as powers of 2^j, with one tree of each size. Based on those facts, the total size consumed by the search structure is $\mathcal{O}(E)$. □

Restating that in terms of N gives the following result.

Lemma 2. *If the time series are presented as sample data, and if the method of interpolation between samples is to connect adjacent samples by curves of degree $d \in \mathcal{O}(1)$, then Algorithm 1 produces an index of size $\mathcal{O}\left(\frac{N}{B} k_{\max}\right)$ disk blocks.*

Proof. There are $\mathcal{O}(N)$ curved segments which arise from interpolating between adjacent samples. We say that a segment "generates" an intersection if the segment's starting time is earlier than that of the other segment that participates in the intersection (we are assuming that no more than two segments intersect at any given point in space and time, which can be made true by arbitrary but consistent tie-breaking). Because each f_i is a function and each segment is of bounded degree, no segment can generate more than $\mathcal{O}(d^2 k_{\max}) = \mathcal{O}(k_{\max})$ intersections.

Since there are only $\mathcal{O}(N)$ segments there can be only $\mathcal{O}(N k_{\max})$ intersections, so the total storage cost is bounded above by $\mathcal{O}\left(\frac{N}{B} k_{\max}\right)$ disk blocks. □

From now on, we will use the setting established in the hypothesis of Lemma 2. That is, we will assume that the data set is composed of sampled time series with adjacent samples connected by curves of bounded degree.

We will now talk about the number of disk operations required to run Algorithm 1.

Lemma 3. *If we define $I := \mathcal{O}\left(\frac{N}{B} k_{\max}\right)$, then Algorithm 1 requires $\mathcal{O}(I \log I)$ disk operations.*

Proof. Since I is the bound on the number of intersections, in order to prove the statement we need only show that the intersections in the data set can be computed in time less than or equal to this. In [6] it is shown that the set of intersections can be computed in $\mathcal{O}\left(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B} + \frac{E}{B}\right)$ where M is the size of main memory and E is the number of intersections in the data set. Since $\frac{N}{B} \in \mathcal{O}(I)$ and $E \in \mathcal{O}(I)$, we conclude that the number of disk operations is bounded by $\mathcal{O}(I \log I + I) = \mathcal{O}(I \log I)$. □

In the next subsection, we describe a more space- and time-efficient data structure.

3.1.2 The Final Static Algorithm

The construction algorithm presented in the previous subsection creates a search structure which, when it is queried, gives a $\mathcal{O}(k)$-length sorted list of the top-k series. However, following the precedent established by SEB-Tree[11], we can make the search structure even more efficient by instead returning a list of $\mathcal{O}(k)$ series, unsorted, in which the top-k are guaranteed to be present.[1] That can be done my making one change: instead of comparing to see if the lists of rankings are the same, on line 5 of Algorithm 3 we test to see whether the top-2^j rankings are the same when viewed as unordered sets.

Algorithm 3 Final construction algorithm.

Require: $\{f_i\}$, $k_{\max}$
1: E $\leftarrow$ all intersections in $\{f_i\}$, sorted by time.
2: **for all** $e \in E$ **do**
3: $\quad t_e \leftarrow$ the time at which e occurs
4: $\quad$ **for all** $0 \le j \le \lfloor \log k_{\max} \rfloor$ **do**
5: $\quad\quad$ **if** $\text{top}_{2^j}(t_e - \varepsilon) \mathbin{\triangle} \text{top}_{2^j}(t_e + \varepsilon) \neq \emptyset$ **then**
6: $\quad\quad\quad$ Store(j, e)
7: $\quad\quad$ **end if**
8: $\quad$ **end for**
9: **end for**

Lemma 4. *Algorithm 3 produces a search index whose size in disk blocks is bounded by* $\mathcal{O}\big(\frac{N}{B} \log_2 k_{\max}\big)$.

Proof. As discussed in the proof of Lemma 2, each intersection is generated by one particular curved segment between adjacent sample points.

Let us examine one particular segment belonging to time series f_i. Whereas all of the segments are of bounded degree, the collection of intersections generated by any segment cannot cross the boundary between the group of top-2^j ranked series and the group of top-2^{j+1} ranked series more than $\mathcal{O}(1)$ times. That implies that one segment can generate no more than $\mathcal{O}(\log_2 k_{\max})$ entries in the index produced by Algorithm 3 because there are only that many ranking groups under consideration.

There are $\mathcal{O}(N)$ segments, so there are only $\mathcal{O}(N \log_2 k_{\max})$ unique intersections in the data structure.

No intersection appears in more than two of the B-Trees built by Algorithm 3, so the total amount of space consumed is $\mathcal{O}\big(\frac{N}{B} \log_2 k_{\max}\big)$. □

Now we will consider the number of disk operations required to build the search structure.

Lemma 5. *If* $I := \mathcal{O}\big(\frac{N}{B} \log k_{\max}\big)$*, then Algorithm 3 requires* $\mathcal{O}\big(I \log \frac{N}{B}\big)$ *disk operations.*

[1] This carries an implicit assumption that $k \le k_{\max}$ is small enough to allow the returned items to be sorted in-core; if that is not the case then an extra $\mathcal{O}(k \log k)$ disk operations are required.

Proof. As the pseudocode is written, it appears that the computation requires at least $\mathcal{O}\left(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B}+\frac{E}{B}\right)$ disk operations because the complete list intersections is computed on line 1 of Algorithm 3.

However, the computation can be made more efficient by only computing the intersections that cause an actual change in the constitution of $\text{top}_{2^j}(\cdot)$, when the value of that function is viewed as an unordered set.

The collection of approximately $\log_2 k_{\max}$ segments that are respectively at the bottom of each of the $\text{top}_{2^j}(-\infty)$ rankings can be found using a segment tree. After that, the list of intersections that define the bottom of each $\text{top}_{2^j}(\cdot)$ from time $-\infty$ to ∞ can be found: the successor in time to any bottom-defining-intersection of a ranking can be computed in $\mathcal{O}\big(\log\frac{N}{B}\big)$ operations by using a segment tree to find the next segment that intersects the present segment. Such a segment tree would require $\mathcal{O}\big(\frac{N}{B}\log\frac{N}{B}\big)$ operations to build.

The total cost of computing all of the interesting intersections is therefore $\mathcal{O}\big(N\log k_{\max}\log\frac{N}{B}\big)$ disk operations because there are $I := \mathcal{O}(N\log k_{\max})$ intersections of interest (as shown in Lemma 4) and each one requires $\mathcal{O}\big(\log\frac{N}{B}\big)$ disk operations to compute.

The less-expensive methodology for computing the relevant intersections also prevents any undue expense from computing the symmetric difference of the two sets shown on line 5. That computation actually never has to be done because an intersection is considered if and only if that difference is non-empty. □

3.1.3 Querying

The query algorithm, Algorithm 4, can correctly query an index created by either Algorithm 1 or 3.

When a request for the top-k series at time t arrives, the query algorithm starts by deciding which of its B-Trees to search. For any $k \le k_{\max}$, querying the j B-Tree where $j = \lceil \log k \rceil$ returns an item of size of $\mathcal{O}(k)$ that is sufficient to compute the desired ranking. The pertinent entry in that tree is found on line 3 by looking for the value whose key is the largest that is less-than-or-equal-to the query time.

The initial top-k tuple and the list of intersections are retrieved from the value returned by the B-Tree query. The algorithm iterates over the time-sorted list of intersections, updating the tuple as it goes (line 7), until an intersection is found whose time is after t, whereupon the algorithm breaks out of the loop and returns the modified tuple. The tuple is the value of $\text{top}_{2^j}(\cdot)$ on the isovalued interval that contains the query time t. In other words, its is $\text{top}_{2^j}(t)$.

3.2 Dynamic Structure

There are two styles of updates that we would like to support. The user might want to have the ability to expand or contract the range of time covered by the data set, and she might also want to be able to add or remove time series from $\{f_i\}$. We refer to those two types of updates as *horizontal* updates and *vertical* updates, respectively. Given the fact that the

Algorithm 4 Query algorithm.

Require: k,t
1: $j \leftarrow 2^{\lceil \log_2 k \rceil}$
2: $\texttt{tree}_j \leftarrow$ the $\text{top}_{2^j}(\cdot)$ tree
3: [`topk` , `list`] $\leftarrow$ FIND-LEQ($\texttt{tree}_j, t$)
4: **for all** $e \in$ `list` **do**
5: $t_e \leftarrow$ the time at which e occurs
6: **if** $t_e \leq t$ **then**
7: UPDATE(`topk`, e)
8: **else**
9: break out of loop
10: **end if**
11: **end for**
12: **return** `topk`

WRLD-TREE is built on top of a standard dynamic B-TREE, support for horizontal updates is automatic. Vertical update support requires some additional work.

Fortunately, the static data structures can be adapted to support vertical updates by using the well-known technique described in [12, Chapter 7].

In brief, that technique can be described in the following way. One creates $\mathcal{O}(\log_2 N)$ static data structures, each with a rough size that is a power of two, and no more than one of each rough size. For example, to store a data set that contains $n = 33$ time series (all with roughly the same number of segments), one would create one static search index containing $2^5 = 32$ series and another containing $2^0 = 1$ series. The creation and deletion of the static sub-structures in response to an insertion follows a pattern analogous to that seen when $n + 1$ is computed in binary arithmetic: when circumstances result in two trees of size 2^j being present in the collection, both of those trees are destroyed and their contents put into a single tree of size 2^{j+1}.

That technique results in a data structure supporting dynamic insertion that requires the same amount of disk space (asymptotically) as the static version.

Because there are $\mathcal{O}(\log_2 N)$ trees, a query of this dynamic structure requires roughly a factor of $\log_2 N$ more disk operations than does a query of a static structure containing the same data.

On an amortized basis, insertions into the data structure require a number of operations proportional to the number required to build a static structure, divided by $\frac{N}{B}$ and again multiplied by $\log \frac{N}{B}$, which results in an amortized insertion cost of $\mathcal{O}\big(\log^2 \frac{N}{B} \log_2 k_{\max}\big)$.

Deletions can be handled in $\mathcal{O}\big(\log^2 \frac{N}{B} \log_2 k_{\max}\big)$ amortized operations by insertion of anti-series. If the series f_i is to be deleted from the data set, that can be implemented by inserting a series f_i^{-1} that is identical to f_i but which instructs the system not to report f_i if both are returned by a query (of course f_i^{-1} is never reported). This will not cause the size of the on-disk data structure to grow out of control: when half of the series contained in the structure are found to be deletions, the entire structure can be scrapped and rebuilt

with no penalty (that is, no penalty in amortized, asymptotic terms). Since up to half of the series found by any query my be anti-series, supporting deletion means that if the top-k series at some time are desired, then a query for the top-$2k$ series (and anti-series) must be done (but this is at no asymptotic penalty).

The query and update costs just given are pessimistic in that they account for a pattern of piecemeal updates. If one makes the possibly more realistic assumption that all additions add $\Omega(N)$ new segments and all deletions remove $\Omega(N)$ segments, then the query cost for the dynamic structure is the same as that of the static one (because there will never be more than a constant number of static sub-structures in the overall dynamic structure). Under that scenario, the amortized update cost is the construction cost divided by $\frac{N}{B}$, or $\mathcal{O}(\log N \log_2 k_{\max})$, which is a logarithmic factor less than the bound in the piecemeal case.

4 Experiments

We performed a number of experiments to validate the algorithms presented in this article. In design, the experiments are largely meant to replicate those shown in [11]. The reason is that the experiments in [11] showed that the SEB-Tree data structure is comprehensively superior to any method that came before it. Our aim here is to not only allow the reader to evaluate the relative merits of Wrld-Tree and SEB-Tree, but also allow her to compare Wrld-Tree to the R-Tree- and MVB-Tree-based approaches.

To that end, we have made our Wrld-Tree implementation freely available and included detailed instructions on how to replicate the experiments reported in this article.[2]

All of the experiments were performed on a computer with an Intel Core 2 Duo CPU, four gigabytes of RAM, and a one terabyte disk which was running Ubuntu 12.04. The Wrld-Tree prototype was written in the Clojure language, version 1.7.0-alpha2, running on OpenJDK version 7. The SEB-Tree code[3] was compiled with gcc 4.6.3.

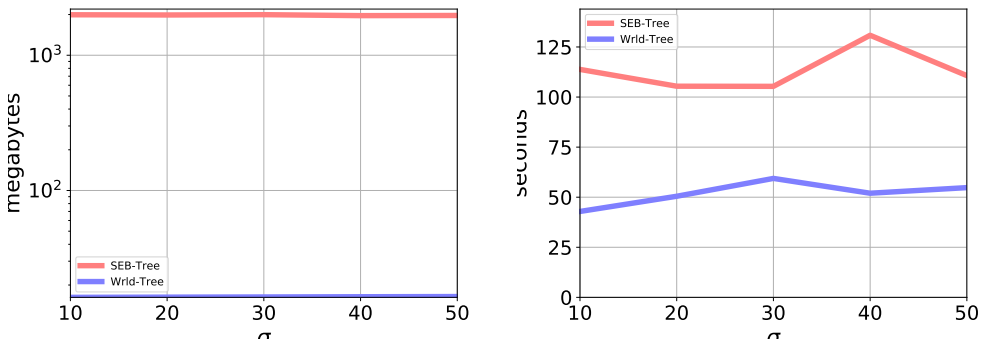
The experiments were performed on a number of synthetic data sets which were randomly-generated, as well as on two real data sets. The synthetic data sets, for compatibility of comparison, were generated to the same rough specifications as those reported in [11]: with some given n (number of series), m (segments per series), and σ (standard deviation of the random magnitudes). The two real data sets, *Mallat* and *Lightcurve*, are both part of the Keogh iSAX corpus[15].[4]

[2]The code and replication instructions are available from `https://github.com/jamesmcclain/WorldTree`.

[3]Available from `http://www.cs.utah.edu/~lifeifei/rankt/`. To allow the code to work with all data sets, we had to make a few minor changes to the SEB-Tree code (mostly to do with increasing the precision of certain numeric comparisons). We have verified that those changes do not decrease the speed of the program to any appreciable degree.

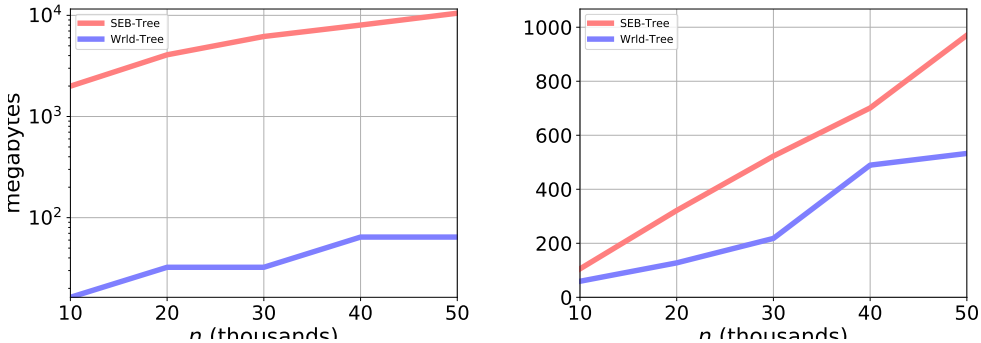
[4]Both data sets are linked-to from the iSAX webpage, `http://www.cs.ucr.edu/~eamonn/iSAX/iSAX.html`. At the time of writing, *Mallat* is found within the CD-Rom zip file and *Lightcurves* is found in its own file linked from that page.

4.1 Construction



(a) Index sizes. (b) Build times.

Figure 3: Index sizes and build times for random series as σ varies.



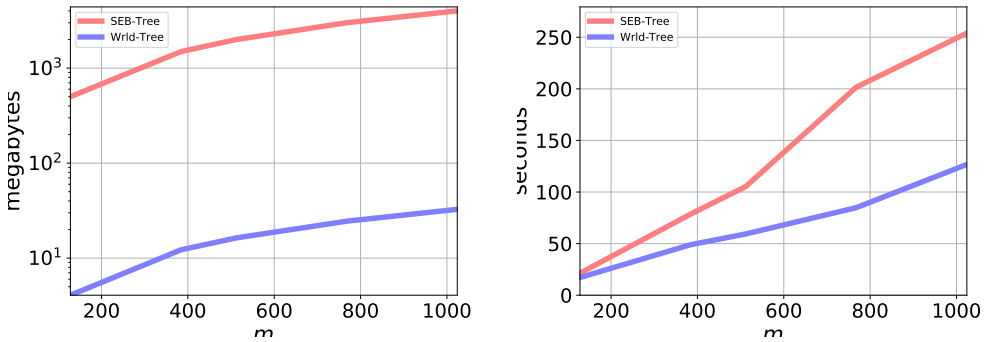
(a) Index sizes. (b) Build times.

Figure 4: Index sizes and build times as n varies.

We performed a number of experiments to evaluate the size of the search indices created by Wrld-Tree and by SEB-Tree, as well as the time required to build them. All of the indices were generated with $k_{\max} = n$.

Figure 3 (cf. [11, Fig. 8]) shows the index sizes and index build times required by various random data sets with σ ranging from 10 to 50, with $n = 10000$ and $m = 512$ fixed. The average size of an index created by SEB-Tree is about 1.9 gigabytes compared to around 17 megabytes for Wrld-Tree.

The index sizes for SEB-Tree are larger than those reported in [11], but we have verified that that discrepancy is due to our building indices with $k_{\max} = n$ instead of $k_{\max} = 200$, as is done in [11]. Setting $k_{\max} = 200$ reduces the sizes of the SEB-Trees by a factor of ten, but doing that also reduces the sizes of the Wrld-Trees by a similar amount.



(a) Index sizes.

(b) Build times.

Figure 5: Index sizes and build times as m varies.

It was shown in [11] that SEB-Tree produces smaller indices, and makes them faster, than previous approaches as σ varies. This experiment shows that the Wrld-Tree construction process is faster and produces more compact output than any of those approaches on comparable input.

Figure 4 (cf. [11, Fig. 9]) shows index sizes and build times on synthetic data sets as n ranges from 10,000 to 50,000, with $\sigma = 30$ and $m = 512$ fixed. The largest index created by SEB-Tree is 10.5 gigabytes, compared to 64 megabytes for Wrld-Tree. The longest construction time was 970 seconds for SEB-Tree compared to 532 seconds for Wrld-Tree.

SEB-Tree was shown in [11] to be better in space and speed than R-Tree or MVB-Tree as n varies, so our experiment on comparable input shows that the same is true of Wrld-Tree.

Figure 5 (cf. [11, Fig. 10]) shows index sizes and build times as m ranges from 128 to 1,024, with $n = 10000$ and $\sigma = 30$ fixed. The largest index created by SEB-Tree was 4 gigabytes (253 seconds), compared to 32 megabytes (126 seconds) for Wrld-Tree.

On the *Mallat* data set, SEB-Tree took 23 seconds to build a 617 megabyte search structure, while Wrld-Tree took 78 seconds to build a 9 megabyte search structure. For the *Lightcurve* data set, SEB-Tree took 49 seconds to produce a 1.1 gigabyte structure and Wrld-Tree took 96 seconds to build a 17 megabyte structure. Here we did not preprocess the data with the SWAB algorithm[10] as was done in [11]. These results can be compared to those shown in [11, Fig. 11].

4.2 Queries

We performed a number of experiments to assess the query performance of Wrld-Tree. In the first three of those experiments, the query parameter $k = 128$ is fixed and σ, n, and m are respectively allowed to vary. In the next three experiments, k is allowed to vary as random queries are performed in the synthetic data sets, in the *Mallat* data set, and in *Lightcurve*. For each combination of parameters, 10,000 random queries were performed and the query costs in kilobytes and milliseconds were averaged.

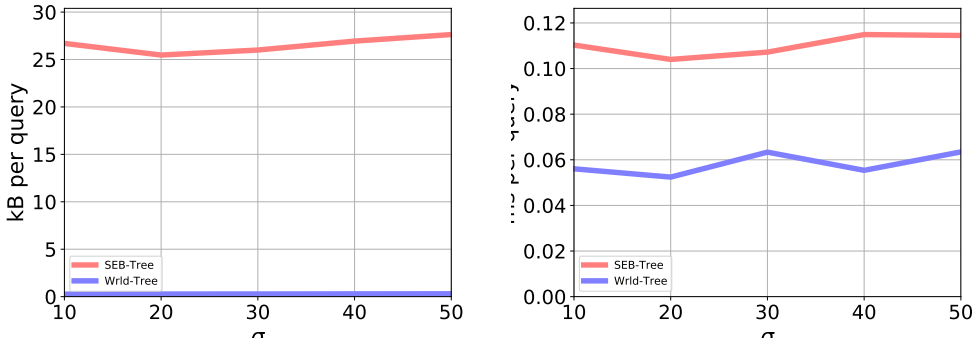


Figure 6: Query performance as σ varies from 10 to 50 for $n = 10000$, $m = 512$, and $k = 128$.

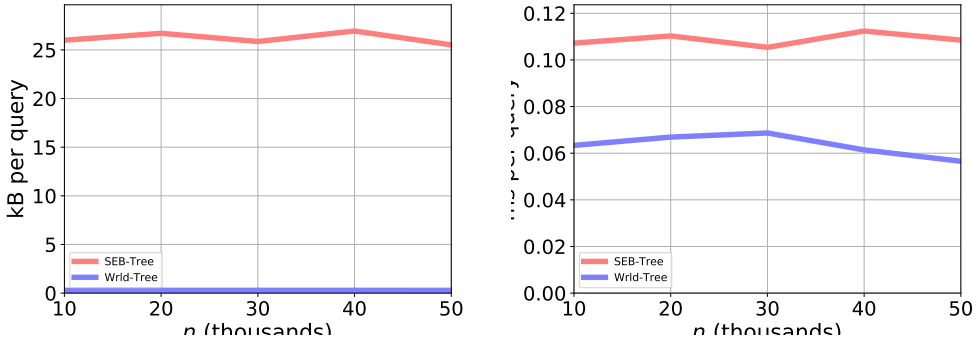


Figure 7: Query performance as n varies from 10000 to 50000 for $\sigma = 30$, $m = 512$, and $k = 128$.

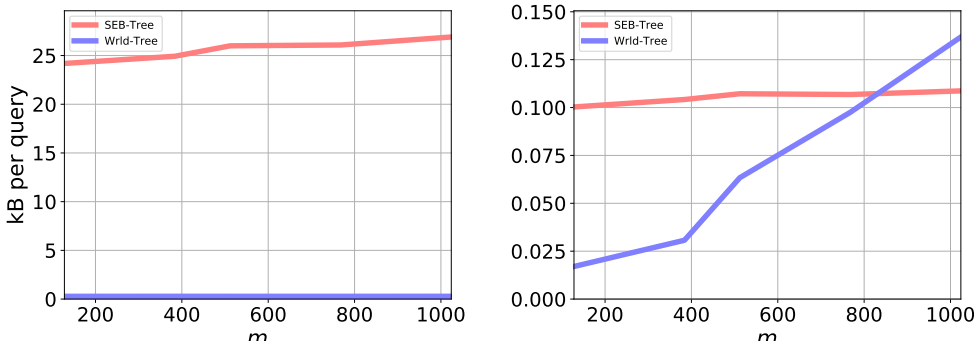


Figure 8: Query performance as m varies from 2 to 1024 for $n = 10000$, $\sigma = 30$, and $k = 128$.

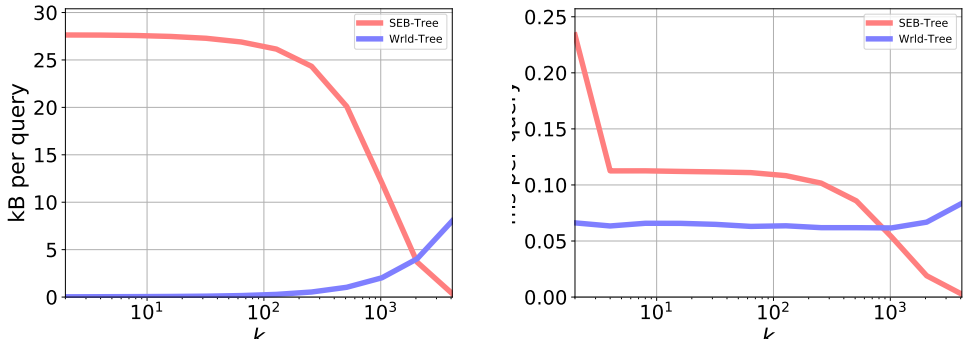


Figure 9: Query performance as k ranges from 2 to 4096 on the randomly-generated datasets.

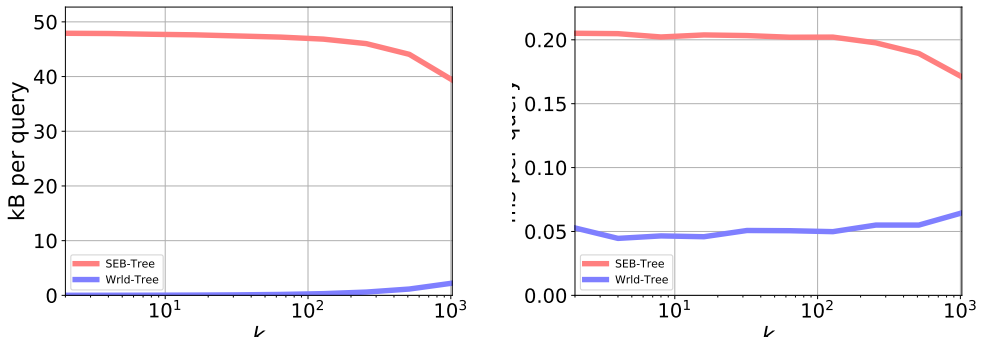


Figure 10: Query performance as k ranges from 2 to 1024 on the Mallat dataset.

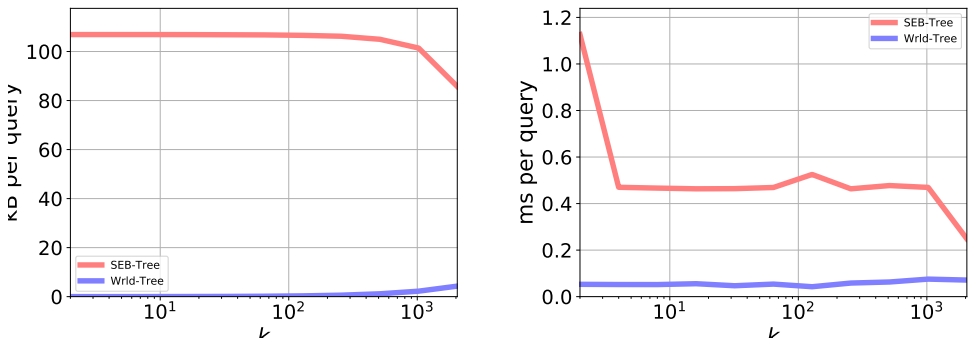


Figure 11: Query performance as k ranges from 2 to 2048 on the Lightcurves dataset.

Figure 6 (cf. [11, Fig. 12(a)]) shows the query performance of WRLD-TREE and SEB-TREE on random data sets for fixed $k = 128$ as σ varies. The typical SEB-TREE query needed about 26 kilobytes and took a little more than 0.1 milliseconds, whereas the typical WRLD-TREE query read about 300 bytes and took about 0.06 milliseconds.

Figure 7 (cf. [11, Fig. 13(a)]) shows the relative performance of the two algorithms as n ranges from 10,000 to 50,000 on synthetic data sets. The requirements in terms of bytes and time were similar in this experiment to those found in the previous one.

Figure 8 (cf. [11, Fig. 14(a)]) shows average query performance as m ranges from 128 to 1,024. The number of bytes requested is similar to those just quoted, but there is a greater range of time requirements for WRLD-TREE than seen in either of the two previously-discussed experiments. This wider range of timings can probably be attributed to a less efficient B-TREE implementation being used in WRLD-TREE than was used in SEB-TREE.

Figures 9, 10, and 11 show the measured query performance of both WRLD-TREE and SEB-TREE as k ranges from 2 to roughly $\frac{n}{2}$ on various data sets. (The behavior of WRLD-TREE can be made symmetric about $\frac{n}{2}$ by storing either the top- or the bottom-k.) On the synthetic data sets, WRLD-TREE does better than SEB-TREE for smaller values of k, whereas SEB-TREE does better when k is larger. On the real data sets, WRLD-TREE does better for all of values of k tested. Please compare Figure 11 to [11, Fig. 15].

4.3 Updates

We performed experiments to measure the update performance of WRLD-TREE. On randomly-generated data sets with $\sigma = 30$ and $m = 128$, we measured the amortized time required to bulk-insert n series into a data structure already containing n series (to double the number of series to $2n$). Here, n ranged from 5,000 to 25,000. The amortized time required ranged from 3.15 milliseconds per series added down to 2.52 milliseconds per series.

On similar randomly-generated data sets, we measured the cost of bulk-deleting n series from a data structure initially containing $2n$ series. As before n was allowed to range from 5,000 to 25,000. The amortized deletion time per series, for each n value, ranged from 1.87 milliseconds on the high-end down to 1.21 milliseconds. Please see Figure 12.

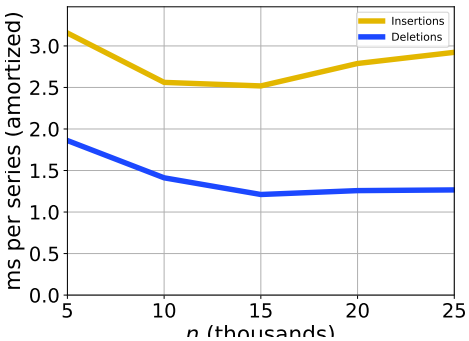


Figure 12: Amortized update performance.

5 Conclusions

In this work, we presented WRLD-TREE, a family of algorithms for answering instant top-k queries on time series. The static data structure presented in Section 3.1.1 returns a sorted list of the top-ranked k time series at time t when it is queried. In Section 3.1.2 we present an algorithm which produces a much smaller data structure than the first one, but which does not return a sorted list. The algorithm given in Section 3.2 produces a data structure that supports dynamic extension, truncation, insertion and deletion of series.

In Section 3, we gave proofs for deterministic bounds on the space requirements, construction times, and query costs of all of the data structures and algorithms presented in this article. Amortized bounds on the costs of insertions and deletions were also given for the dynamic structure. The storage requirements of the indices given in Sections 3.1.2 and 3.2 are both better than any previously published results of which we are aware.

In Section 4 we gave empirical evidence of the algorithms' performance to compliment the theoretical analysis given in the section before that. On both synthetic and real data sets, we showed that the size of the WRLD-TREE data structure is much smaller than that of any previously-published approach, while requiring similar construction time. The query performance of WRLD-TREE was shown to be better than that of the R-TREE- or MVB-TREE-based approaches, and similar to that of SEB-TREE (better on real data sets, better or worse on synthetic data).

6 Future Work

A generalization of the instant top-k query is to perform the query over intervals of time rather than instants. An area of possible future inquiry is the application of the ideas presented in this article to that question. More broadly, the general theme of viewing data structures as functions, and trying from that perspective to precompute and store them efficiently, is one that we are interested in applying to other databases-related questions.

7 Acknowledgments

For this project we made use of data sets provided by Keogh and contributors. For our experimental comparisons, we made use of the SEB-TREE implementation provided by Li and others. We would also like to thank Mark Pugner for interesting conversation and valuable thoughts.

References

[1] P. K. Agarwal and J. Erickson. Geometric range searching and its relatives. *Contemporary Mathematics*, 223:1–56, 1999.

[2] I. J. Balaban. An optimal algorithm for finding segments intersections. In *Proceedings of the eleventh annual symposium on Computational geometry*, pages 211–219. ACM, 1995.

[3] B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer. An asymptotically optimal multiversion B-tree. *The VLDB Journal—The International Journal on Very Large Data Bases*, 5(4):264–275, 1996.

[4] T. M. Chan. Random sampling, halfspace range reporting, and construction of $\leq k$-levels in three dimensions. *SIAM Journal on Computing*, 30(2):561–575, 2000.

[5] B. Chazelle and H. Edelsbrunner. An optimal algorithm for intersecting line segments in the plane. *Journal of the ACM (JACM)*, 39(1):1–54, 1992.

[6] A. Crauser, P. Ferragina, K. Mehlhorn, U. Meyer, and E. A. Ramos. Randomized external-memory algorithms for line segment intersection and other geometric problems. *International Journal of Computational Geometry & Applications*, 11(03):305–337, 2001.

[7] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. *Journal of Computer and System Sciences*, 66(4):614–656, 2003.

[8] A. Guttman. *R-trees: a dynamic index structure for spatial searching*, volume 14. ACM, 1984.

[9] W. Jin and J. M. Patel. Efficient and generic evaluation of ranked queries. In *Proceedings of the 2011 international conference on Management of data*, pages 601–612. ACM, 2011.

[10] E. Keogh, S. Chu, D. Hart, and M. Pazzani. An online algorithm for segmenting time series. In *Data Mining, 2001. ICDM 2001, Proceedings IEEE International Conference on*, pages 289–296. IEEE, 2001.

[11] F. Li, K. Yi, and W. Le. Top-k queries on temporal data. *The VLDB Journal*, 19(5):715–733, 2010.

[12] K. Mehlhorn. Multidimensional searching and computational geometry, volume 3 of data structures and algorithms, 1984.

[13] J. I. Munro and H. Suwanda. Implicit data structures for fast search and update. *Journal of Computer and System Sciences*, 21(2):236–250, 1980.

[14] R. Razaghian. Establishing financial credibility in the United States, 1789-1860: The impact of institutions, 2001.

[15] J. Shieh and E. Keogh. iSAX: indexing and mining terabyte sized time series. In *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 623–631. ACM, 2008.