Identifying Similarities, Periodicities and Bursts for Online Search Queries

Michail Vlachos †
UC Riverside
mvlachos@cs.ucr.edu

Chris Meek
Microsoft Research
meek@microsoft.com

Zografoula Vagena
UC Riverside
foula@cs.ucr.edu

Dimitrios Gunopulos
UC Riverside
dg@cs.ucr.edu

ABSTRACT

We present several methods for mining knowledge from the query logs of the MSN search engine. Using the query logs, we build a time series for each query word or phrase (e.g., 'Thanksgiving' or 'Christmas gifts') where the elements of the time series are the number of times that a query is issued on a day. All of the methods we describe use sequences of this form and can be applied to time series data generally. Our primary goal is the discovery of semantically similar queries and we do so by identifying queries with similar demand patterns. Utilizing the best Fourier coefficients and the energy of the omitted components, we improve upon the state-of-the-art in time-series similarity matching. The extracted sequence features are then organized in an efficient metric tree index structure. We also demonstrate how to efficiently and accurately discover the important periods in a time-series. Finally we propose a simple but effective method for identification of bursts (long or short-term). Using the burst information extracted from a sequence, we are able to efficiently perform 'query-by-burst' on the database of time-series. We conclude the presentation with the description of a tool that uses the described methods, and serves as an interactive exploratory data discovery tool for the MSN query database.

1. INTRODUCTION

Online search engines have become a cardinal link in the chain of everyday internet experience. By managing a structured index of web pages, modern search engines have made information acquisition significantly more efficient. Indicative measures of their popularity are the number of hits that they receive every day. For example, large search services such as Google, Yahoo, and MSN each serve results for tens of millions of queries per day. It is evident that search services, aside from their information retrieval role, can also act as a data source for identifying trends, periodicities and important events by careful analysis of the queries.

Retaining aggregate query information, such as the number of times each specific query was requested every day, is storage efficient, can accurately capture descriptive trends and finally it is privacy preserving. This information could be used by the search service to further optimize their indexing criteria, or for mining interesting news patterns, that manifest as periodicities or peaks in the query log files.

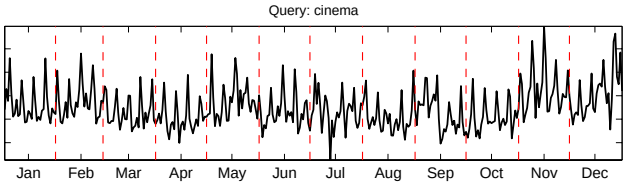

Figure 1: Query demand for the word "cinema" for every day of 2002

As an illustrative example, let's consider the query "cinema". In fig. 1 we observe the request pattern for every day of 2002. We can distinguish 52 peaks that correspond to each weekend of the year. In fact, similar trends can be noticed for specific cinemas as well, indicating a clear preference of going to the movies during Friday and Saturday. By distilling such a knowledge, the engineers of a search service can optimize the search of a certain class of queries, during the days that a higher query load is expected. Such a discussion is out of the scope of this paper, however possible ways of achieving this could be (for example) enforcing higher redundancy in their file servers for a specific class of queries.

While a significant number of queries exhibit strong weekly periodicities, some of them also depict seasonal bursts. In fig. 2 we observe the trend for the word "Easter", where a clear accumulation of the queries during the relevant months, followed by an immediate drop after Easter. On a similar note, the query "Elvis" experiences a peak on 16th Aug. every year (fig. 3), which happens to be the death anniversary of Elvis Presley.

The above examples are strong indicators about the amount of information that can be extracted by close examination of the query patterns. To summarize, we believe that past query logs can serve 3 major purposes:

1. Recommendations (the system can propose alternative

† Part of this work conducted while author was visiting Microsoft Research.

SIGMOD 2004 June 13-18, 2004, Paris, France.

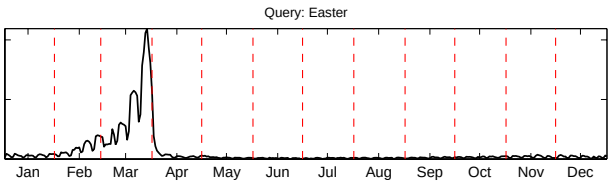

Figure 2: Search pattern for the word "easter" during 2002

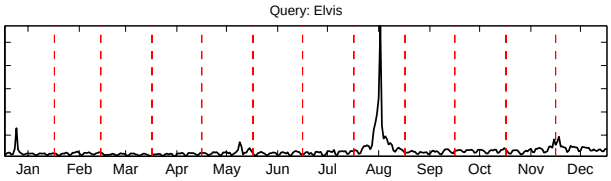

Figure 3: The demand for query "elvis" for every day of 2002

or related keywords, that depict similar request patterns)

2. Discovery of important news (burst of a specific query)
3. Optimization of the search engine (place similar queries in same server, since they are bound to be retrieved together)

In the following sections we will explain how one can extract useful information from query logs, in general, and the MSN query logs, in particular.

1.1 Contributions

This paper makes three main contributions. First we develop new compressed representations for time-series and an indexing scheme for these representations. Because the data we are dealing with tend to be highly periodic, we describe each time-series using the k best Fourier coefficients (instead of the frequently used first ones). In this manner we are able to provide the *tightest yet* lower and upper bounds on euclidean distance between time-series. We demonstrate how this representation can be indexed using a variation of a metric tree. The index is very compact in size and exhibits strong pruning power due to the bounds that we provide. The new algorithms described in the paper, are presented in terms of Fourier coefficients but can be generalized to *any* orthogonal decomposition with minimal or no effort. Our second contribution is an 'automatic' method for discovering the number of significant periods. Our final contribution is a simple yet effective way to identify bursts in time-series data. We extract burst features that can later be stored in a relational database and subsequently be used to perform 'query-by-burst' searches, that is, used to find all sequences that have a similar pattern of burst behavior.

The paper roadmap is as follows: In Section 2, we describe various tools for analyzing time-series including the Discrete Fourier Transform (DFT) and the power spectral density. In Sections 3 and 4, we describe our approach to efficiently representing, storing and indexing a large collection of time-series data. In Section 5, we develop a method for identifying significant periodicities in time-series. In Section 6, we describe a simple but effective approach to burst detection in time-series and its application to 'query-by-burst' searching of a collection of time-series. Finally, in Sections 7 and 8, we experimentally demonstrate the effectiveness of the proposed methods and discuss directions for future work.

2. SPECTRAL ANALYSIS

We provide a brief introduction to the Fourier decomposition, which we will later use for providing a compressed representation of time-series sequences.

2.1 Discrete Fourier Transform

The normalized Discrete Fourier Transform (DFT) of a sequence $x(n), n = 0, 1 \dots N-1$ is a vector of complex numbers $X(f)$:

$$X(f_{k/N}) = \frac{1}{\sqrt{N}} \sum_{n=0}^{N-1} x(n) e^{-j2\pi kn/N}, \quad k = 0, 1 \dots N-1$$

We are dealing with real signals, therefore the coefficients are symmetric around the middle one. What the Fourier transform attempts to achieve is, to represent the original signal as a linear combination of the complex sinusoids $s_f(n) = \frac{e^{j2\pi fn/N}}{\sqrt{N}}$. Therefore, the Fourier coefficients represent the amplitude of each of these sinudoids, after signal x is projected on them.

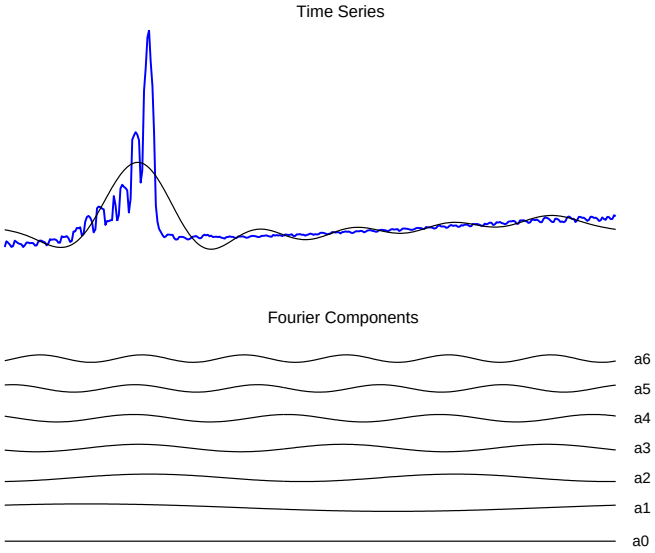

Figure 4: Decomposition of a signal into the first 7 DFT components

2.2 Power Spectral Density

In order to accurately capture the general shape of a time-series using a spartan representation, one could reconstruct the signal using just its dominant frequencies. By dominant we mean the ones that carry most of the signal energy. A popular way to to identify the power content of each frequency is by calculating the *power spectral density PSD* (or power spectrum) of a sequence which indicates the signal power at each frequency in the spectrum. A well known estimator of the PSD is the periodogram. The periodogram P is a vector comprised of the squared magnitude of the Fourier coefficients:

$$P(f_{k/N}) = \|X(f_{k/N})\|^2, \quad k = 0, 1 \ldots \lceil \frac{N-1}{2} \rceil$$

Notice that we can detect frequencies that are at most half of the maximum signal frequency, due to the Nyquist fundamental theorem. The k dominant frequencies appear as peaks in the periodogram (and correspond to the coefficients with the highest magnitude). From here on, when we refer to the best or largest coefficients, we would mean the ones that have the highest energy and correspond to the tallest peaks of the periodogram.

3. COMPRESSED REPRESENTATIONS FOR PERIODIC DATA

In this section, we describe several compressed representations for periodic data. Our primary goal is to support fast similarity searches for periodic data. To that end, we want our compressed representation to support approximate Euclidean distance computations, and, more specifically, good upper and lower bounds on the actual Euclidean distance between a compressed representation and a query point. For each of the representations, we provide algorithms to compute upper and lower bounds. Throughout the presentation we refer to Fourier coefficients because we concentrate on periodic data, however, our algorithms can be adapted to any class of orthogonal decompositions (such as wavelets, PCA, etc.) with minimal or no adjustments.

3.1 First or Best Coefficients?

The majority of the approaches that attempt to speed-up similarity search are based on the work of Agrawal et al. [1], where the authors lower bound the Euclidean distance using the first k Fourier coefficients. The authors use the name GEMINI to describe their generic framework for lower bounding the distance and utilizing a multidimensional index for candidate retrieval. Rafiei et al. [13] improve on this method by exploiting the symmetry of the Fourier coefficients and, thus, provide an even tighter lower bound (using the same number of coefficients). We will refer to this lower bound using the symmetric property as *LB-GEMINI*.

A later enhancement to this paradigm appears in [14] by Wang & Wang. The authors, in addition to the first coefficients, also record the *error* of the approximation to the original sequence. Using this extra information, the authors provide a tighter lower bound (*LB-Wang*) and an upper bound (*UB-Wang*).

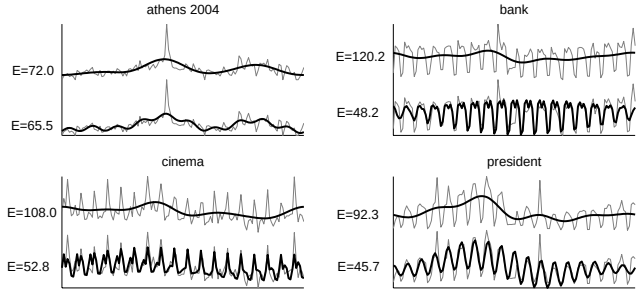

Figure 5: A comparison of using the first coefficients vs best coefficients in reconstructing the time-series for four queries. Using the best coefficients can significantly reduce the reconstruction error.

All of the above methods inherently suffer from the assumption, that the first coefficients describe adequately the decomposed signal. While this may be true for some time-series such as random walks, it is not true for many time-series that exhibit strong periodicities. In such sequences most of the power is not concentrated in the low frequency components, but is dispersed at various frequencies of the spectrum. Figure 5 depicts the reconstruction error E to the original sequence when using the 5 first Fourier coefficients against the 4 best (the explanation of space requirement for each method will be deferred until later). It is evident that using the coefficients with the most power yields superior reconstruction even when using fewer components.

The observation that it is best to represent a signal using the largest coefficients of a DFT (or other orthogonal decomposition) is not novel. For instance, Wu et al. [15] note that choosing the best coefficients can be a fast and powerful alternative to SVD for searching images. It is also useful to note that in addition to providing a tighter distance approximation, the use of the best coefficients, has the advantage of offering an immediate overview of the periodic components of the data.

3.2 Notation

We will present our algorithms for lower bounding the Euclidean distance, using the best coefficients to approximate a sequence. We begin with some notation first.

We denote a time-series by $t = \{t_1, t_2, \ldots, t_n\}$ and the Fourier transformation of t by the capital letter T. The vector describing the positions of the k largest coefficients of T is denoted as p^+, while the positions of the remaining ones as p^- (that is $p^+, p^- \subset [1, \ldots, n]$). For any sequence T, we will store in the database the vector $T(p^+)$ or equivalently T^+. Now if Q is a query in the transformed domain, then $Q(p^+)$ (or Q^+) describes a sequence holding the equivalent coefficients as the vector $T(p^+)$. Similarly, $Q(p^-) \equiv Q^-$ is the vector holding the analogous elements of $T(p^-) \equiv T^-$.

Example: Suppose $T = \{(1+2i), (2+2i), (1+i), (5+i)\}$ and $Q = \{(2+2i), (1+i), (3+i), (1+2i)\}$. The magnitude vector of T is: $abs(T) = \{2.23, 2.82, 1.41, 5.09\}$. Then, $p^+ = \{2, 4\}$, $T(p^+) = \{(2+2i), (5+i)\}$ and $Q(p^+) = \{(1+i), (1+2i)\}$.

3.3 Algorithm BestMin

In order to speedup similarity search, we compute a lower bound of the Euclidean distance between the compressed representation $T(p^+)$ and the full query data $Q = \{Q(p^+), Q(p^-)\}$. The squared Euclidean distance is defined as:

$$\begin{aligned} D(Q,T)^2 &= D(Q(p^+), T(p^+))^2 + D(Q(p^-), T(p^-))^2 \\ &= \|Q(p^+) - T(p^+)\|^2 + \|Q(p^-) - T(p^-)\|^2 \end{aligned} \quad (1)$$

The computation of the first part of the distance is trivial since we have all the required data. For the second part we are missing the term $T(p^-)$, the discarded coefficients. Because we select the best coefficients, we know that the magnitude of each of the coefficients in $T(p^-)$ is less than the smallest magnitude in $T(p^+)$. We use $minPower = \|T^+_{min}\|$ to denote the magnitude of the smallest coefficient in $T(p^+)$.

FACT 1. ***[minProperty]*** *The magnitude of all coefficients in $T(p^-)$ is less than the minimum magnitude of any coefficient in $T(p^+)$ (by construction). That is: $\|T^+_{min}\| \geq \|T^-_i\|$.*

We can use this fact to lower bound $D(Q^-, T^-)^2$.

$$\|Q^- - T^-\|^2 = \sum_{i \in p^-} \|Q_i^- - T_i^-\|^2 \qquad (2)$$

Every element of the sum will be replaced by something of smaller (or equal) value.

$$\|Q_i^- - T_i^-\| \geq \|Q_i^-\| - \|T_i^-\| \quad \text{(triangle inequality)}$$
$$\|Q_i^- - T_i^-\| + \|T_{min}^+\| \geq \|Q_i^-\| - \|T_i^-\| + \|T_i^-\| \quad \text{(minProperty)}$$
$$\|Q_i^- - T_i^-\| \geq \|Q_i^-\| - \|T_{min}^+\|$$

The distance *LB_BestMin*, named for its use of the best (largest) coefficients and the minProperty, is a lower bound of the Euclidean distance and defined as follows:

$$LB\_BestMin(Q,T)^2 =$$
$$\sum \begin{cases} \|Q_i - T_i\|^2 & \text{if } i \in p^+ \\ (\|Q_i\| - \|T_{min}^+\|)^2 & \text{if } i \in p^- \text{ and } \|Q_i\| \geq \|T_{min}^+\| \\ 0 & \text{if } i \in p^- \text{ and } \|Q_i\| < \|T_{min}^+\| \end{cases}$$

Similarly, we can define an *upper bound* to the Euclidean distance as follows:

$$UB\_BestMin(Q,T)^2 = \sum \begin{cases} \|Q_i - T_i\|^2 & \text{if } i \in p^+ \\ (\|Q_i\| + \|T_{min}^+\|)^2 & \text{if } i \in p^- \end{cases}$$

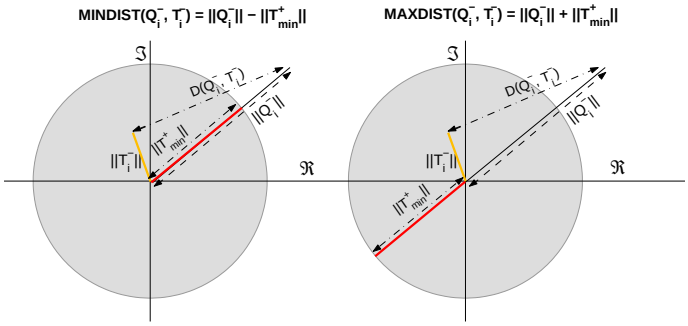

Figure 6: BestMin Explanation. All points T_i^- lie within the circle of radius $\|T_{min}^+\|$. *Left:* The minimum possible distance between any point Q_i^- and T_i^- happens when the 2 vectors are aligned and then their distance simply is $\|Q_i^-\| - \|T_i^-\|$. *Right:* The maximum distance is $\|Q_i^-\| + \|T_i^-\|$.

3.4 Algorithm BestError

In this section, we describe the BestError algorithm. This algorithm typically provides a looser lower bound than the BestMin algorithm and is presented to facilitate the understanding of the BestMinError algorithm described in the next section. In this algorithm we utilise an additional quantity $T.err = \|T^-\|^2$ the sum of squares of the omitted coefficients. This quantity represents the error in the compressed representation, or, equivalently, the amount of energy in the coefficients not represented. For this algorithm we only use knowledge of $T.err$ and ignore any additional constraints such as the minProperty, thus the algorithm could be applied when coefficients other than the best coefficients are chosen.

```
1 [LB, UB] = BestMin(Q,T)
2  {
3     LB = 0; // lower bound
4     UB = 0; // upper bound
5     DistSq = 0; // distance of best coefficients
6     minPower = min(abs(T)); //smallest best coeff
7
8     for i = 1 to length(Q)
9     {
10        if T[i] exists
11        // i is a coefficient used
12        // in the compressed representation
13        DistSq += abs(Q[i] - T[i])^2;
14        else
15        {
16            // lower bound
17            if (abs(Q[i]) > minPower)
18                LB += (abs(Q[i]) - minPower)^2;
19
20            //upper bound
21            UB += (abs(Q[i]) + minPower)^2;
22        }
23    }
24    LB = sqrt(DistSq + LB);
25    UB = sqrt(DistSq + UB);
26 }
```

Figure 7: Algorithm BestMin

To obtain the lower bound for the quantity of interest we use the inequality $\|Q^- - T^-\| \geq \|Q^-\| - \|T^-\|$ and for the upper bound we use the inequality $\|Q^- - T^-\| \leq \|Q^-\| + \|T^-\|$. These inequalities yield the following upper and lower bounds when using the best coefficients and the approximation error.

$$\begin{cases} LB\_BestError(Q,T)^2 = \|(Q^+ - T^+)\|^2 + (\|Q^-\| - \|T^-\|)^2 \\ UB\_BestError(Q,T)^2 = \|(Q^+ - T^+)\|^2 + (\|Q^-\| + \|T^-\|)^2 \end{cases}$$

Note that these measures are analogous to what had been proposed in [14] but for the case of best coefficients.

```
1 [LB, UB] = BestError(Q,T)
2  {
3     // In this approach we store the sum of squares
4     // of the coefficients not in the compressed
5     // representation for T in T.err
6     Q.err = 0   // used to store unused energy of Q
7     DistSq = 0; // distance of best coefficients
8     for i = 1 to length(Q)
9     {
10        if T[i] exists
11        {
12           // i is a coefficient used
13           // in the compressed representation
14           DistSq += abs(Q[i] - T[i])^2;
15        }
16        else
17        {
18           Q.err +=  abs(Q[i])^2;
19        }
20    }
21
22    LB = sqrt(DistSq +(sqrt(Q.err)-sqrt(T.err))^2;
23    UB = sqrt(DistSq +(sqrt(Q.err)+sqrt(T.err))^2;
24 }
```

Figure 8: Algorithm BestError

3.5 Algorithm BestMinError

Our last algorithm is the BestMinError algorithm that uses the best coefficients and both the minProperty and

$T.err$ to obtain a tighter lower bound. The algorithm is described in Figure 9. The algorithm is somewhat more complicated than the previous algorithms and we provide some intuitions to aid the reader. The basic idea is to compute a lower and an upper bound of this quantity iteratively. For each coefficient not in the compressed representation we consider two cases:

Case 1: When $Q[i] > minPower$ we use the $minProperty$. We are certain that we can increment the distance by $(abs(Q[i]) - minPower)^2$ for this coefficient (line 24). For the lower bound, the most optimistic case is when this is precisely this distance and "use" precisely $minPower$ energy (line 26). Note that using this metaphor we would also say that we have used all of the energy in $Q[i]$. For the upper bound, the worst case is that we have use none of the energy from $T.err$.

Case 2: When $Q[i] \leq minPower$ the $minProperty$ does not apply. In this case we increment the count of unused energy from Q by the size of $Q[i]$ (line 30).

Roughly, we compute the lower and upper bound by using the upper and lower bound from the BestError case with the unused energies. More specifically, for the lower bound, we add the distance computed for the known coefficients, the distance computed in case 1 and the best-case distance using the overestimate of the unused energy from the missing coefficients of T. Similarly for the upper bound, we combine these quantities but use an underestimate of the amount of energy used namely $T.err$.

```
1 [LB, UB] = BestMinError(Q,T)
2  {
3     // In this approach we store the sum of squares
4     // of the coefficients not in the compressed
5     // representation for T in T.err
6     LB = 0;          // lower bound
7     DistSq = 0;      // distance of best coefficients
8     Q.nused = 0;     // energy of unused coeffs for Q
9     T.nused = T.err; // energy left for lower bound
10
11    minPower = min(abs(T)); //smallest best coeff
12
13    for i = 1 to length(Q)
14    {
15        if T[i] exists
16        // i is a coefficient used
17        // in the compressed representation
18        DistSq += abs(Q[i] - T[i])^2;
19        else
20        {
21            // lower bound
22            if (abs(Q[i]) > minPower)
23            {
24                LB += (abs(Q[i]) - minPower)^2;
25                // at most minPower used
26                T.nused -= minPower^2;
27            }
28            else
29            // this energy wasn't used
30                Q.nused += abs(Q[i])^2;
31        }
32    }
33
34    // have we used more energy than we had?
35    if (T.nused < 0) T.nused = 0;
36
37    UB=sqrt(DistSq+LB+(sqrt(Q.nused)+sqrt(T.err))^2);
38    LB=sqrt(DistSq+LB+(sqrt(Q.nused)-sqrt(T.nused))^2);
39 }
```

Figure 9: Algorithm BestMinError

4. INDEX STRUCTURE

All of the algorithms proposed in section 3, use a different set of coefficients to approximate each object, thus making difficult the use of traditional multidimensional indices such as the R*-tree [2]. Moreover, in our algorithms we use *all* the query coefficients in the new projected orthogonal space, making the adaptation of traditional space partition indices almost impossible.

We overcome these challenges by utilizing a *metric* tree as the basis of our index structure. Metric trees do not cluster objects based on the values of the selected features but on relative object distances. The choice of reference objects, from which all object distances will be calculated, can vary in different approaches. Examples of metric trees include the Vp-tree [16], M-tree [7] and GNAT [4]. All variations of such trees, exploit the distances to the reference points in conjunction with the triangle inequality to prune parts of the tree, where no closer matches (to the ones already discovered) can be found.

In this work we will use a customized version of the VP-tree (vantage point tree). The superiority of the VP-tree against the R*-tree and the M-tree, in terms of pruning power and disk accesses, was clearly demonstrated in [5]. Here, for simplicity, we describe the modifications in the search algorithms for the structure of the traditional static binary VP-tree. However all possible extensions to the VP-tree, such as the usage of multiple vantage points [3] or accommodation of insertion and deletion procedures [5] can be implemented on top of the proposed search mechanisms. We provide a brief introduction to the Vp-tree structure, and we direct the interested reader to [6], for a more thorough description.

4.1 Vantage Point Tree

In this section, we adapt the notation of [6]. At every node in a VP-tree there is a vantage point (reference point). The vantage point is used to divide all of the points associated with that node of the tree into two equal-sized sets. More specifically, the distances of points associated with the vantage point are sorted and the points with distance less than the median distance μ are placed in the left subtree (subset $S_{\leq}$), and the remaining ones in the right subtree (subset $S_{>}$).

To construct a VP-tree for a dataset one simply needs to choose a method for selecting a vantage point from a set of points and to use this method recursively to construct a tree. We use a heuristic method to pick vantage point in constructing VP-trees. In particular, we choose the point that has the highest deviation of distances to the remaining objects. This can be viewed as an analogue of the largest eigenvector in SVD decomposition.

Suppose that one is searching for the 1-Nearest-Neighbor (1NN) to the query Q and its distance to a vantage point is $D_{Q,VP}$. If the best-so-far match is σ we only need to to examine both subtrees if $\mu - \sigma < D_{Q,VP} < \mu + \sigma$. In any other case we only have to examine one of the subtrees.

Typically VP-trees are built using the original data, that is, the vantage points are not compressed representations. In our approach, we adapt the VP-tree to work with a compressed representation. By doing so we significantly reduce the amount of space needed for the index. Of course, the downside of using a compressed representation as the vantage points is that the distance computation is no longer

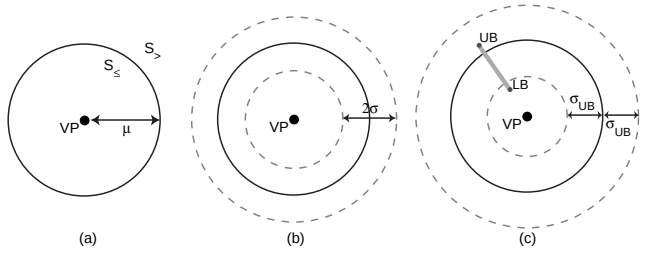

Figure 10: (a) Separation into two subsets according to median distance, (b) Pruning in VP-tree using exact distances, (c) Pruning in VP-tree using upper and lower bounds

exact and we must resort to using bounds. Our approach is to construct the VP-tree using an uncompressed representation and, then, after it is constructed, convert the vantage points to the appropriate compressed representation and ID of the original object (time series). By doing so, we obtain exact distances during the construction process. One can optimize this process slightly by compressing a point immediately after it is selected to be a vantage point.

We will modify the knn search algorithm to utilize the lower and upper bounds of the uncompressed query Q to a compressed object (whether this is vantage point or leaf object). So, suppose for the best-so-far match we have a lower bound σ_{LB} and a upper bound σ_{UB}. We will only visit the left subtree if the upper bound distance between Q and the current vantage point VP is: $UB_{Q,VP} < \mu - \sigma_{UB}$. This happens since for any $R \in S_>$ the distance between R and Q is:

$$\begin{aligned} D(Q,R) &\geq |D(R,VP) - D(Q,VP)| \quad \text{(triangle inequality)} \\ &\geq |\mu - D(Q,VP)| \quad \text{(by construction)} \\ &\geq |\mu - UB(Q,VP)| \\ &> |\mu - (\mu - \sigma_{UB})| \quad \text{(assumption)} \\ &= \sigma_{UB} \end{aligned}$$

and since our best match is less than σ_{UB} this part of the tree can be safely discarded. In a similar way we will only traverse the right subtree ($S_>$) if the lower bound between Q and VP is: $LB_{Q,VP} > \mu + \sigma_{UB}$. For any other case both subtrees will have to be visited (fig. 10 (c))

The index scan is performed using a depth-first traversal, and σ_{LB}, σ_{UB} are updated both by compressed objects in the leaves as well as by the vantage points. Additionally, as an optimization, the search is heuristically 'guided' towards the most promising nodes. Our heuristic works as follows: Consider the annulus (disc with a smaller disc removed) defined by the upper and lower bounds for a query centered around the current vantage point. We can divide this area into two areas (one of them possibly empty); one region in which the points that are further away than the median μ from the current vantage point and one in which the points are closer than the median. Each child of the current vantage point is naturally associated with one of these regions and we choose the child node associated with the region of larger size. Suppose, for example, that the lower and upper bounds of the query with respect to the current vantage point are in the range [LB-UB], as shown by the gray line in fig. 10(c). Because the distance range overlaps more with the subset $S_\leq$, we should follow this subset first. It seems likely that this approach will find a good match sooner and leading to quicker pruning of other parts of the tree.

Even though we prune parts of the tree using the upper bound σ_{UB} of the best-so-far match (and not the exact distance) the pruning power of the index is kept very high, because the use of algorithm $BestMinError$ can provide a significantly tight upper bound. This will be explicitly demonstrated in the experimental section.

After the tree traversal we have a set of compressed objects with their lower and upper bounds. The smallest upper bound (SUB) is computed and all objects with lower bound higher than SUB are excluded from examination. The full representation of the remaining objects is retrieved from the disk, in the order suggested by their lower bounds. A simple version of the 1NN search (without the traversal to the most promising node) is provided in fig. 11.

5. DETECTING IMPORTANT PERIODS

Using the periodogram we can visually identify the peaks as the k most dominant periods (period =1/frequency). However, we would like to have an automatic method that will return the important periods for a set of sequences (e.g., for the knn results). What we need is to set an appropriate threshold in the power spectrum, that will accurately distinguish the dominant periods. We will additionally require that this method not only identifies the strong periods, but also reduces the number of false alarms (i.e., it doesn't classify unimportant periods as important).

5.1 Number of significant periods

Next we devise a test to separate the important periods from the unimportant ones. To do so one needs to specify exactly what is meant by a non-periodic time series. Our canonical model of a non-periodic time-series is a sequence of points that are drawn independently and identically from a Gaussian distribution. Clearly this type of sequence can have no meaningful periodicities. In addition, under this assumption, the magnitudes of the coefficients of the DFT are distributed according to an exponential distribution. Even when the assumption of i.i.d. Gaussian samples does not hold, it is often the case that the histogram of the coefficient magnitudes has an exponential shape. Fig. 12 illustrates this for several non-periodic time series. Our approach is to identify significant periods by identifying outliers according to an exponential distribution.

Starting from the probability distribution function of the exponential distribution, we derive the cumulative distribution function:

$$\begin{aligned} f(x) &= \lambda e^{-\lambda x} & \Rightarrow \\ P(x \geq A) &= 1 - \int_0^{\infty} f(x)dx & = e^{-\lambda x} \end{aligned}$$

where λ is the inverse of the average value. The important periods will have powers that deviate from the power content of the majority of the periods, therefore we will seek for infrequent powers. Consequently, we will set this probability p to a very low value and calculate the derived power threshold. For example, if we want to be confident with probability 99.99% that the returned periods will be significant we have: $(100 - 99.99)\% = 0.01\% = 0.0001$, and

```
NNSearch(Q)
{
Input: Uncompressed Query Q
Output: Nearest Neighbor

    S <- Search(root-of-index, Q)
    // S = set of compressed objects returned from
    // tree traversal with associated
    // lower (S.LB) and upper bounds (S.UB)

    SUB = min(S(i).UB); // smallest upper bound
    Delete({S(i) | S(i).LB> SUB}); // prune objects
    Sort(S(i).LB); // sort by lower bounds

    bestSoFar.dist = inf;

    for i=1 to S.Length
    {
       if S(i).LB > bestSoFar.dist
          return bestSofar

       retrieve uncompressed time-series T
       of S(i) from database

       dist = D(T,Q); // full euclidean

       if dist < bestSoFar
       {
            bestSoFar.dist = dist;
            bestSoFar.ID = T;
       }
    }
}

Search(node,Q)
{
Input: Node of Vp-tree, uncompressed query Q
Output: set of unpruned compressed objects with
associated lower and upper bounds

    if node is leaf
    {
        for each compressed time-series cT in node
        {
            (LB,UB) <- BestMinError(cT,Q);
            queue.push(cT,LB,UB); // sorted by UB
        }
    }
    else // vantage point
    {
        (LB,UB) <- BestMinError(VP,Q);
        queue.push(VP,LB,UB);
        sigmaUB = queue.top; // get best upper bound

        if UB < node.median - sigmaUB
            search(node.left,Q);
        if LB > node.median + sigmaUB
            search(node.right,Q);
    }
}
```

Figure 11: 1NN Search for VP-tree with compressed objects

the power probability is set to 10^{-4}. Solving for the power threshold T_p we get:

$$ln(p) = ln(e^{-\lambda T_p}) = -\lambda T_p \Rightarrow$$

$$T_p = -\frac{ln(p)}{\lambda} = -\mu \cdot ln(p)$$

and μ is the average signal power. For example for confidence 99.99%, $p = 10^{-4}$ and if the average signal power 0.02 $(= 1/n \sum_i x_i^2)$, then the *power density threshold* value is $T_p = 0.0184$.

Examples: We demonstrate the accuracy and usefulness

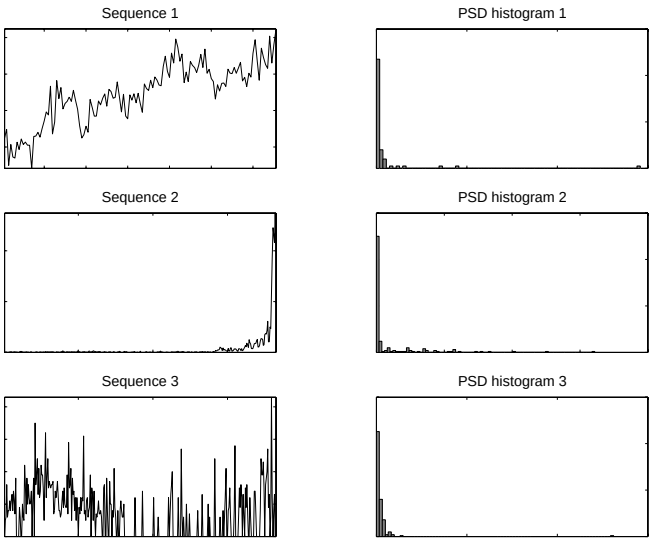

Figure 12: Typical histogram of power spectrum for various non-periodic sequences follow an exponential distribution

of the proposed method with several examples. In fig. 13 we juxtapose the demand of various queries during 2002 with the periods identified as important. We can distinguish a strong weekly component for queries 'cinema' & 'nordstrom', while for 'full moon' the monthly periodicity is accurately captured. For the last example we used a sequence without any clear periodicity and again our method is robust enough to set the threshold high enough, therefore avoiding false alarms. The large peak in the data happens during the day the famous British actor died, and of course its identification is important for the discovery of important (rare) events. We will expound how to discover such (or more subtle) bursts in the following section.

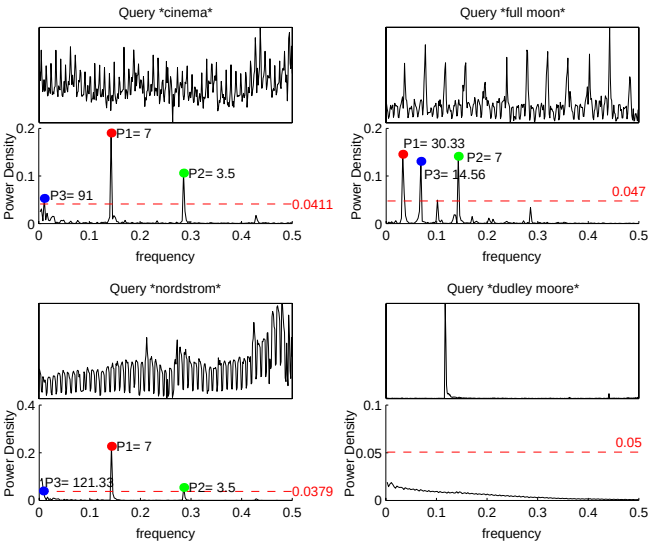

Figure 13: Discovered periods for four queries using the power density threshold

6. BURST DISCOVERY

Our final method for knowledge extraction, involves the detection of bursts. In the setting of this work, interactive burst discovery will involve three tasks; first we have to detect the bursts, then we need to compact them, in order to facilitate an efficient storage scheme in a DBMS system and finally based on compacted features we can pose queries in the system ('query-by-burst'). The query-by-burst feature can be thought of as a fast alternative of weighted Euclidean matching, where the focus is given on the bursty portion of a sequence. Compared to the work of Zhu & Shasha [17], our approach is more flexible since it does not require a custom index structure, but can easily be integrated in any relational database. Moreover, our framework requires significantly less storage space and in addition we can support similarity search based on the discovered bursts. Our method is also simpler and less computationally intensive than the work of [11], where the focus is on the modeling of text streams.

6.1 Burst Detection

For discovering regions of burst in a sequence, our approach is based on the computation of the moving average (MA), with a subsequent annotation of bursts as the points with value higher than x standard deviations above the mean value of the MA. More concretely:

1. `Calculate Moving Average` MA_w `of length w for sequence` $t = (t_1, ...t_n)$.
2. `Set cutoff = mean(`MA_w`) + x*std(`MA_w`)`
3. `Bursts =` $\{t_i \mid MA_w(i) > cutoff\}$

For our database we used sliding windows of two lengths w; one used a moving average of 30 days (long-term bursts) and one a 7 day moving average (short-term bursts), which we found to cover sufficiently well the bursts ranges in the database sequences. Typical values for the cutoff point are 1.5-2 times the standard deviation of the MA. In fig. 14, we can see a run of the above algorithm on the query 'Halloween' during the year 2002. We observe that the burst discovered is indeed during the October and November months. In fig. 15 another execution of the algorithm is demonstrated, this time for the word "Easter", during a span of three years.

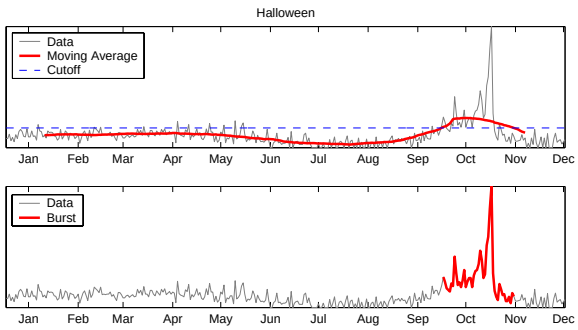

Figure 14: User demand for the query 'Halloween' during 2002 and the bursts discovered

6.2 Burst Compaction

We would like to identify in an large database, sets of sequences that exhibit similar burst patterns. In order to speedup this process, we choose not to store all points of the bursty portion of a sequence, but we will perform some kind of *feature compaction*.

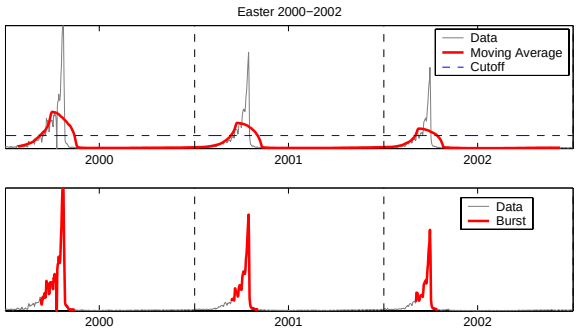

Figure 15: History of the query 'Easter' during the years 2000-2002 and the bursts discovered

For this purpose, we represent each consecutive sequence of values identified as a burst, by their average value. Suppose, for example, that $B^{(X)} = (x_p, \ldots, x_{p+k}, x_q, \ldots, x_{q+m})$ signify the points identified as bursts on a sequence $X = (x_1, x_2, \ldots, x_n)$. In this case, sequence B contains two burst regions and the compact form of the burst is:

$$\begin{aligned} B^{(X)} &= (\overbrace{x_p, x_{p+1} \ldots x_{p+k}}, \overbrace{x_q, x_{q+1} \ldots x_{q+m}}) \\ &= (B_1^{(X)}, B_2^{(X)}) \\ &= ([p, p+k, \frac{1}{p+k-1}\sum_{i=p}^{p+k} x_i], [q, q+m, \frac{1}{q+m-1}\sum_{i=q}^{q+m} x_i]) \end{aligned}$$

In other words, each burst is characterized by a triplet *[startDate, endDate, average value]*, indicating the start and ending point of the burst, and the average burst value during that period, respectively. The length of a burst B will be indicated by $|B| = endDate - startDate + 1$.

The burst triplets of each sequence can now be stored as records in a DBMS table with the following fields:

`[sequenceID, startDate, endDate, average burst value]`

In fig. 16 we elucidate the compact burst representation, as well as the high interpretability of the results. For the query 'flowers', we discover two long-term bursts during the months February and May. This is consistent with our expectation that flower demand tends to peak during Valentine's Day and Mother's Day. For the query 'full moon' (using short term burst detection) we can effectively distuinguish the monthly bursts (that is once for every completion of the moon circle).

6.3 Burst Similarity Measures

Now we will define the similarity $Bsim$ between the burst triplets. Let time-series X and Y and their respective set of burst features $B^{(X)} = (B_1^{(X)}, \ldots, B_k^{(X)})$ and $B^{(Y)} = (B_1^{(Y)}, \ldots, B_m^{(Y)})$. We have :

$$BSim = \sum_{i=1}^{k}\sum_{j=1}^{m} intersect(B_i^{(X)}, B_j^{(Y)}) * similarity(B_i^{(X)}, B_j^{(Y)})$$

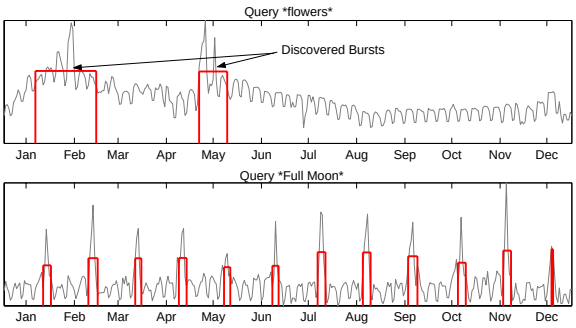

Figure 16: Compact burst representation by using the average value of the bursts, and high interpretability of the discovered bursts.

where *similarity* captures how close the average burst values are:

$$similarity(B_i^{(X)}, B_j^{(Y)}) = \frac{1}{1 + dist(B_i^{(X)}, B_j^{(Y)})}$$
$$= \frac{1}{1 + (avgValue(B_i^{(X)}) - avgValue(B_j^{(Y)})}$$

and *intersect* returns the degree of overlap between the bursts:

$$intersect = \frac{1}{2}\left(\frac{overlap(B_i^{(X)}, B_j^{(Y)})}{|B_i^{(X)}|} + \frac{overlap(B_i^{(X)}, B_j^{(Y)})}{|B_j^{(Y)}|}\right)$$

The function *overlap* simply calculates the time intersection between two bursts. fig. 17 briefly demonstrates for bursts A,B the calculation of $overlap(A, B)$.

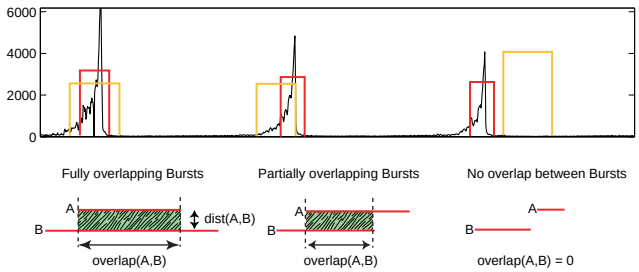

Figure 17: Burst overlaps between time-series

Before the burst features are extracted, the data are standardized (subtract mean, divide by std) in order to compensate for the variation of counts for different queries.

Execution in a DBMS system: Since all identified burst features are stored in a database system, it is very efficient to discover burst features that overlap with the query's bursts.

In fig. 18 we illustrate the search for overlapping bursts. Essentially we need to discover features with *startDate* earlier than the ending date of the query burst and with *endDate* later than the burst starting date. This procedure is extremely efficient, if we create an index (basically a B-tree) on the *startDate* and *endDate* attributes. For the qualifying burst features, the similarity measures are accordingly calculated for their respective sequences as described.

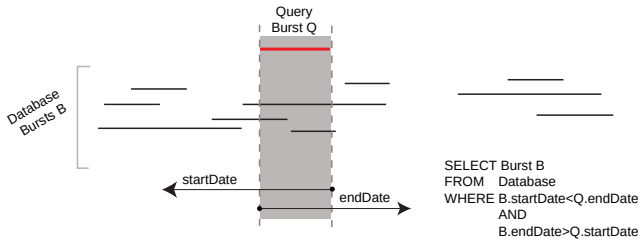

Figure 18: Identifying Overlapping Bursts in a DBMS

In fig. 19 we show some results of burst similarity measure. It is prevalent that 'query-by-burst' can be a powerful asset for our knowledge discovery toolbox. This type of representation and approach to search is especially useful for finding matches for time series with non-periodic bursts.

7. EXPERIMENTAL EVALUATION

Now we will demonstrate with extensive experiments the effectiveness of the new similarity measures and compare their performance with other widely used Euclidean approximations. We also exhibit that our bounds lead to good pruning performance and that our approach to indexing has good performance on nearest neighbor queries.

For our experiments all sequences had length of 1024 points, capturing almost 3 years of query logs (2000-2002). The dataset sizes range up to 2^{15} time-series, effectively containing more than 30 million daily measurements in our database. All sequences were standardized and the queries were sequences not found in the database. The experiments have been conducted on a 2Ghz Intel Pentium 4, with 1GB of RAM and 60GB of secondary storage.

7.1 Space Requirements

For fair comparison of all competing methods, it is imperative to judge their performance when the *same amount of memory* is alloted for the coefficients of each approach.

The storage of the first k Fourier coefficients requires just 2k doubles (or 2k*8 bytes). However, when utilizing the k best coefficients for each sequence, we also need to store their positions in the original DFT vector. That is, the compressed representation with the k largest coefficients is stored as pairs of `[position-coefficient]`.

For the purposes of our experiments 2048 points are more than adequate, since they capture more than 5 years of log data for each query. Taking under account the symmetric property we just need to store 1024 positions, so 10 bits would be sufficient. However, since on disk we can write only as multiples of bytes, each position requires 2 bytes. In other words, each coefficient in our functions requires 16+2 bytes, and if GEMINI uses k coefficients, then our method will use $\lfloor 16k/18 \rfloor = \lfloor k/1.125 \rfloor$ coefficients.

For some distance measures we also use one additional double to record the error (sum of squares of the remaining coefficients). Therefore, for the measures that don't use the error we need to allocate one additional number and we choose this to be the middle coefficient of the full DFT vector, which is a real number [12] (since we have real data with lengths power of two). If in some cases the middle coefficient happens to be one of the k best ones, then these sequences just use 1 less double than all other approaches. The follow-

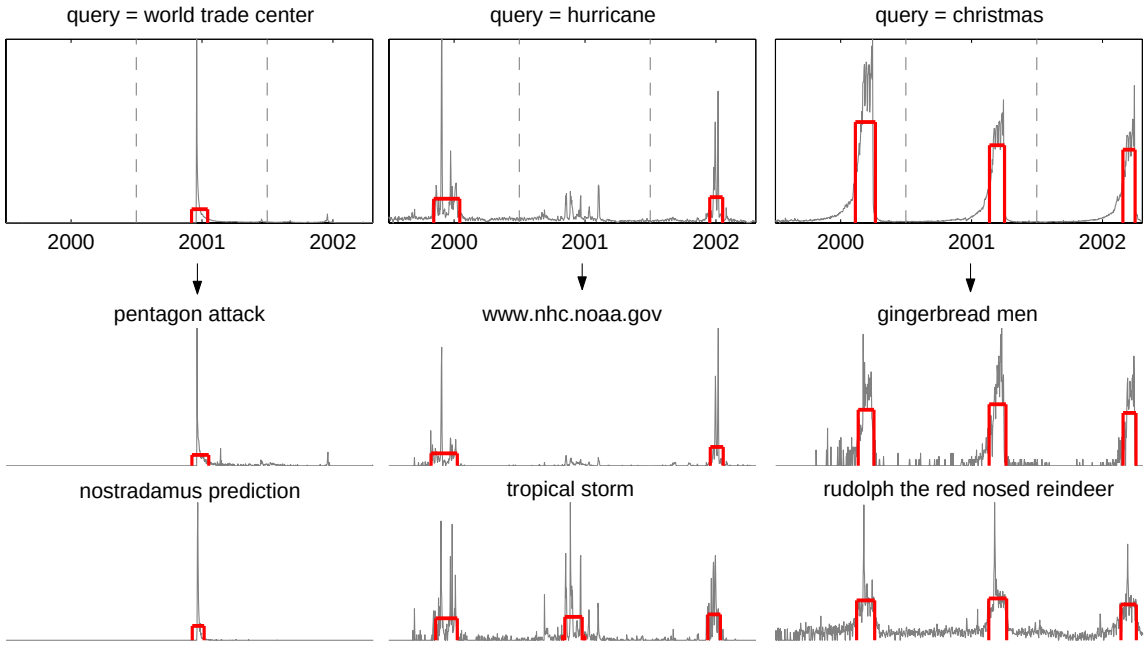

Figure 19: Three examples of 'query-by-burst'. We depict a variety of interesting results discovered when using the burst similarity measures.

ing table summarizes how the same amount of memory is allocated for each compressed sequence of every approach.

GEMINI	c First Coeffs + Middle Coeff
Wang	c First Coeffs + Error
BestMin	$\lfloor c/1.125 \rfloor$ Best Coeffs + Middle Coeff
BestError	$\lfloor c/1.125 \rfloor$ Best Coeffs + Error
BestMinError	$\lfloor c/1.125 \rfloor$ Best Coeffs + Error

Table 1: Requirements for usage of same storage for each approach

Therefore, when in the following figures we mention memory usage of [2*(32)+1] doubles, the number in parenthesis essentially denotes the coefficients used for the GEMINI and Wang approach (+ 1 for the middle coefficient or the error, respectively). For the same example, our approach uses the 28 best coefficients but has the same memory requirements.

7.2 Tightness of Bounds

This first experiment measures the tightness of the lower and upper bounds of our algorithms. We compare with the approaches that utilize the first coefficients (GEMINI) and with the bounds proposed by Wang using the first coefficients in conjunction with the error.

In figures 20 and 21 we show the lower and upper bounds for all approaches in conjunction with the actual Euclidean distance. The distance shown is the cumulative euclidean distance over 100 random pairwise distance computations from the MSN query database. First, observe that the *BestMinError* provides the best improvement. Second, the Wang approach provides the best approximation when the first coefficients are used. Using *BestMinError* there is a noticeable 6-9% improvement in the lower bound and a 13-18% improvement in the upper bounds, compared to the next best method (LB_Wang). However, as we will see in the following section, this improvement in distance leads

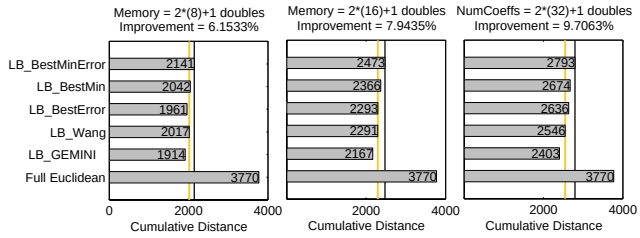

Figure 20: Lower Bounds: The methods taking advantage of the best coefficients (and especially BestMinError) outperform the approaches using the first coefficients.

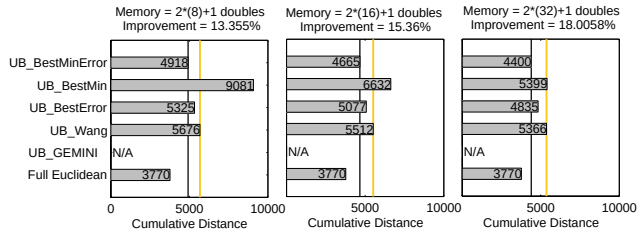

Figure 21: In the calculation of Upper Bounds, BestMinError provides the tightest Euclidean approximation

to a significant improvement in pruning power. For the remainder of the paper we do not report results for the *BestMin* and *BestError* methods due to the superiority of the *BestMinError* method.

7.3 Pruning Power

We evaluate the effectiveness the Euclidean distance bounds in a way that is not effected by implementation details or the use of an index structure. The basic idea is to measure

the average fraction $\mathcal{F}$ of the database objects examined in order to find the 1NN for a set of queries. In order to compute $\mathcal{F}$ for a given query Q, we first compute the lower and upper bound distances to Q for each object using its compressed representation. Note that we do not compute an upper bound for GEMINI. We find the smallest upper bound (SUB) and objects that have $LowerBound > SUB$ are pruned. For the remaining sequences the full representations are retrieved from disk and compared with the query, in increasing order as suggested by the lower bounds. We stop examining objects, when the current lower bound is higher than the best-so-far match. Similar methods for evaluation have also appeared in [8, 10].

The results we present are for a set of 100 randomly selected queries not already in the database. Fig. 22 shows the dramatic reduction in the number of objects that need to be examined, a reduction that ranges from 10-35% compared to the next best method. These positive effects can be attributed to four reasons:

- Our methods make use of *all* coefficients of a query, thus giving tigher distance estimates.
- The best coefficients provide a high quality reconstruction of the indexed sequences.
- The knowledge of the remaining energy significantly tightens the distance estimation.
- Finally, the calculation of upper bounds, reduces the number of candidate sequences that need to be examined.

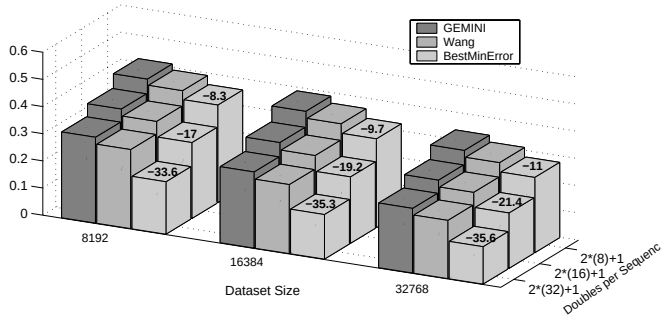

Figure 22: Fraction of database objects examined for three compression factors. BestMinError inspects the least number of objects, even though fewer coefficients are used.

7.4 Index Performance

In our final experiment, we measure the CPU time required by the Linear Scan and our index structure to return the 1-Nearest-Neighbor to 50 queries (not already found in the database).

For our test datasets, due to the high compression, the index size and the compressed features could easily fit in memory, therefore we provide two running times for our index; the first one is with all the compressed features in memory and the second one is with the compressed sequences in secondary storage.

In fig. 23 we report the running time for the linear scan which uses the uncompressed sequences, in comparison with the index running time for various compression factors and database sizes. Both approaches were optimized to perform an early termination of the Euclidean distance, when the running sum exceeded the best-so-far match. We can observe that when the compressed features are retrieved from the disk the index is approximately 20-25 times faster than the sequential scan. In the case where the compressed sequences fit in memory the speedup exceeds the 120 times. Notice that this running time, includes the random I/O to read the uncompressed sequences from the disk. The best performance for the memory resident index in observed when more coefficients are utilized. However, for the external memory index the highest compression factors achieve the best performance. This is attributed to the reduced I/O costs for this case, and it is also a significant indicator that a few number of the best coefficients can capture accurately the sequence shape. The result is very promising since it demonstrates that we can achieve exceptional performance with compact external memory indices.

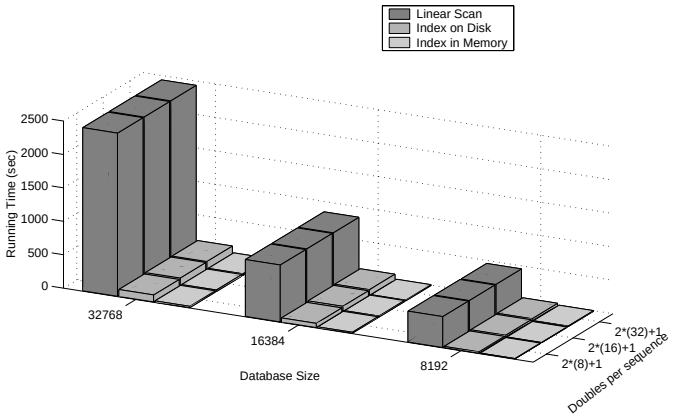

Figure 23: Fraction of running time required by our index structure to return the 1NN, compared to Linear Scan. The observed speedup is at least 20 times (for disk based index), exceeding 2 orders of magnitude when the compressed features reside in memory.

7.5 The S2 Similarity Tool

We conclude this section with a brief description of a tool that we developed which incorporates many of the features discussed in this paper.

Our tool is called S2 which stands for *Similarity Tool*. The program is implemented in C# and it interacts with a remote SQL database server to retrieve the actual sequences, while the compressed features are stored locally for faster access. Realtime response rates are observed for the subset of the top 80000+ sequences, whose compressed representation the program utilizes. The user can pose a search keyword and similar sequences from the MSN query database are retrieved. A snapshot of the main window form is presented in fig. 24. The program offers three major functionalities:

- Identification of important periods
- Similarity search

- Burst Detection & Query-by-Burst

The user can examine at any time the quality of the time-series approximation, based on the best-k coefficients. Additionally, a presentation of the discovered bursts for the sequence is also possible. It is at the user's discretion to use all or some of the best-k periods for similarity search, therefore effectively concentrating on just the periods of interest. Similar functionality is provided for burst search.

Figure 24: Snapshot of the S2 tool and demonstration of 'query-by-burst'

8. CONCLUSIONS AND FUTURE WORK

In this work we proposed methods for improving the tightness of lower and upper bounds on Euclidean distance, by carefully selecting the information retained in the compressed representation of sequences. In addition, the selected information allows us to utilize a full query rather than a compressed representation of the query which further improves the bounds and significantly improves the index pruning performance. Moreover, we have presented simple and effective ways for identifying periodicities and bursts. We applied these methods on real datasets from the MSN query logs and demonstrated with extensive examples the applicability of our contributions.

In the approach described here, we choose a fixed number of coefficients for each object. A natural extension of this approach is to allow for a variable number of coefficients. For instance, one possibility in the case of Fourier coefficients is to add the best coefficients until the compressed representation contains $k\%$ of the energy in the signal (or, equivalently, the error is below some threshold). This type of compressed representation is easily indexed using our customized VP-tree index.

In addition, we feel that this approach can be fruitfully applied for other types of similarity queries. In particular, we believe that a similar approach could prove useful in the computation of linear-cost lower and upper bounds for expensive distance measures like dynamic time warping [9].

9. REFERENCES

[1] R. Agrawal, C. Faloutsos, and A. Swami. Efficient Similarity Search in Sequence Databases. *In Proc. of the 4th FODO*, pages 69–84, Oct. 1993.

[2] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The r*-tree: An efficient and robust access method for points and rectangles. In *Proc. of ACM SIGMOD*, 1990.

[3] T. Bozkaya and M. Özsoyoglu. Distance-based indexing for high-dimensional metric spaces. In *Proc. of SIGMOD*, 1997.

[4] S. Brin. Near neighbor search in large metric spaces. In *Proc. of 21th VLDB*, 1995.

[5] A. W. chee Fu, P. M. Chan, Y.-L. Cheung, and Y. Moon. Dynamic vp-tree indexing for n-nearest neighbor search given pair-wise distances. *Journal of VLDB*, 2000.

[6] T. Chiueh. Content based image indexing. In *Proc. of VLDB*, 1994.

[7] P. Ciaccia, M. Patella, and P. Zezula. M-tree: An efficient access method for similarity search in metric spaces. In *Proc. of 23rd VLDB*, pages 426–435, 1997.

[8] J. Hellerstein, C. Papadimitriou, and E. Koutsoupias. Towards an analysis of indexing schemes. In *Proc. of 16th ACM PODS*, 1997.

[9] E. Keogh. Exact indexing of dynamic time warping. In *Proc. of VLDB*, 2002.

[10] E. Keogh, K. Chakrabarti, S. Mehrotra, and M. Pazzani. Locally adaptive dimensionality reduction for indexing large time series databases. In *Proc. of ACM SIGMOD*, pages 151–162, 2001.

[11] J. Kleinberg. Bursty and hierarchical structure in streams. In *Proc. of 8th SIGKDD*, 2002.

[12] A. Oppenheim, A. Willsky, and S. Nawab. *Signals and Systems, 2nd Edition*. Prentice Hall, 1997.

[13] D. Rafiei and A. Mendelzon. Efficient retrieval of similar time sequences using dft. In *Proc. of FODO*, 1998.

[14] C. Wang and X. S. Wang. Multilevel filtering for high dimensional nearest neighbor search. In *ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery*, 2000.

[15] D. Wu, D. Agrawal, A. E. Abbadi, A. K. Singh, and T. R. Smith. Efficient retrieval for browsing large image databases. In *Proc. of CIKM*, pages 11–18, 1996.

[16] P. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *Proc. of 3rd SIAM on Discrete Algorithms*, 1992.

[17] Y. Zhu and D. Shasha. Efficient elastic burst detection in data streams. In *Proc. of 9th SIGKDD*, 2003.