SeqIndex: Indexing Sequences by Sequential Pattern Analysis*

Hong Cheng Xifeng Yan Jiawei Han

Department of Computer Science
University of Illinois at Urbana-Champaign
{hcheng3, xyan, hanj}@cs.uiuc.edu

Abstract

In this paper, we study the issues related to the design and construction of high-performance sequence index structures in large sequence databases. To build effective indices, a novel method, called SeqIndex, is proposed, in which the selection of indices is based on the analysis of discriminative, frequent sequential patterns mined from large sequence databases. Such an analysis leads to the construction of compact and effective indexing structures. Furthermore, we eliminate the requirement of setting an optimal support threshold beforehand, which is difficult for users to provide in practice. The discriminative, frequent pattern based indexing method is proven very effective based on our performance study.

1 Introduction

Sequential pattern mining is an important and active research theme in data mining [3, 9, 11, 4], with broad applications. However, with the diversity of searching and mining requests and daunting size of datasets, it is often too slow to perform mining on-the-fly but it is impossible to mine and store all the possible results beforehand.

A powerful but long-lasting alternative to this mining dilemma is to build a good sequence index structure which may serve as a performance booster for a large variety of search and mining requests. A sequence indexing structure will not only make data mining more flexible and efficient but also facilitate search and query processing in sequence databases.

Given a sequence database, there are in general two kinds of indexing structures that can be constructed for subsequence search: *consecutive* vs. *non-consecutive* subsequence indices. There are a number of *consecutive* sequence indexing methods developed for time-series data [1, 2, 5, 7, 12] and DNA sequences [8, 6]. Euclidean distance, dynamic time warping (DTW) and discrete fourier transform (DFT) are commonly used tools for indexing and similarity search in time-series data. DNAs is another kind of consecutive sequence. Since it is consecutive, suffix-tree and multi-resolution string (MSR) are developed for DNA indexing.

To the best of our knowledge, there is no *non-consecutive* sequence indexing method studied before. Nonconsecutive sequence mining and matching is important in many applications, such as customer transaction analysis, various event logs, sensor data flows, and video data. Indexing non-consecutive sequence poses great challenges to research. First, there exist an explosive number of subsequences due to the combination of gaps and symbols. Second, such an indexing mechanism cannot be built on top of the existing consecutive sequence indexing mechanisms. For example, models like suffix-tree or those based on time-series transformation, such as DFT and DTW, are no longer valid because sequences under our examination are not consecutive any more.

In this paper, we propose a new indexing methodology, called *discriminative, frequent sequential pattern-based (DFP) indexing*, which selects the best indexing features, based on frequent sequential pattern mining and discriminative feature selection, for effective indexing of sequential data. The study extends the recent work on graph indexing by Yan et al. [10] and examines the applicability of DFP in sequence databases.

Although the general framework of using frequent patterns as indexing features has been exposed in graph indexing [10], it is not obvious whether this framework can be successfully applied to sequences. In fact, there exists an inverted index based method to do sequence indexing. For each item, there is an id list associated with it. To process a query, just intersect the id lists. We call this algorithm ItemIndex. In this work, we compare ItemIndex and SeqIndex from multiple angles to explore the boundaries of these two algorithms.

* The work was supported in part by the U.S. National Science Foundation IIS-02-09199, IIS-03-08215. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

The remainder of the paper is organized as follows. Section 2 introduces the basic concepts related to sequence indexing and some notations used throughout the paper. Section 3 presents the algorithm to determine the size-increasing support function. Section 4 formulates the SeqIndex algorithm. We report and analyze our performance study in Section 5 and conclude our study in Section 6.

2 Preliminary Concepts

Let $I = \{i_1, i_2, \ldots, i_k\}$ be a set of all items. A *sequence* $s = \langle i_{j_1}, i_{j_2}, \ldots, i_{j_n} \rangle$ $(i_j \in I)$ is an ordered list. We adopt this sequence definition to simplify the description of our indexing model. A sequence $\alpha = \langle a_1, a_2, \ldots, a_m \rangle$ is a *sub-sequence* of another sequence $\beta = \langle b_1, b_2, \ldots, b_n \rangle$, denoted as $\alpha \sqsubseteq \beta$ (if $\alpha \neq \beta$, written as $\alpha \sqsubset \beta$), if and only if $\exists j_1, j_2, \ldots, j_m$, such that $1 \leqslant j_1 < j_2 < \ldots < j_m \leqslant n$ and $a_1 = b_{j_1}, a_2 = b_{j_2}, \ldots,$ *and* $a_m = b_{j_m}$. We also call β a *super-sequence* of α, and β *contains* α.

A sequence database, $D = \{s_1, s_2, \ldots, s_n\}$, is a set of sequences. The *support* of a sequence α in a sequence database D is the number of sequences in D which contain α, $support(\alpha) = |\{s | s \in D \textit{ and } \alpha \sqsubseteq s\}|$. Given a minimum support threshold, *min_sup*, a sequence is **frequent** if its support is no less than *min_sup*. The set of **frequent sequential pattern**, FS, includes all the frequent sequences.

Definition 2.1. *(Subsequence Search) Given a sequence database $D = \{s_1, ..., s_n\}$ and a query sequence q, it returns the answer set $D_q = \{s_i | s_i \in D, q \sqsubseteq s_i\}$.*

In sequence query processing, the major concern is query response time, which is composed of the time of searching, the time of fetching the candidate set from the disk and the cost to check the candidates. We want to minimize the search time since a query could have a lot of subsequences in the index. If searching in the index structure is very inefficient, a large amount of time would be wasted on intersecting some id lists which do not shrink the candidate answer set a lot. So we need an efficient algorithm to search in the "right" direction. In addition, the I/O part also plays an important role.

3 Determine the Size-increasing Support Function

The first step is to mine frequent patterns from sequence database. As pointed out in [10], the purpose of *size-increasing support constraint* is to make the mining tractable. Meanwhile, the overall index quality may be optimized. This constraint is effective in keeping a compact index size while providing high-quality indices. An interesting question is how to determine the size-increasing support function in a systematic way? Instead of using heuristic functions, a novel solution is to use the discriminative ratio as a guide to select the appropriate threshold. It becomes a data mining problem. In this setting, we not only need to mine frequent sequential patterns with *min_sup* but also have to decide what *min_sup* is.

Let $\psi(l)$ be the size-increasing support function. The automated setting of $\psi(l)$ is as follows. Given a discriminative threshold γ_{min} and a sequence database D, we first set the support of all length-1 patterns to be 1. Then we determine $\psi(l)$ in a level wise manner. When we decide for length $k \geq 2$, we set $\psi(k) = \psi(k-1)$. Under this support, we mine a set of frequent length-k patterns. For every length-k pattern x , we calculate its discriminative ratio with respect to patterns that have already been selected,

$$\gamma = \frac{|\bigcap_{\varphi i: f_{\varphi_i} \sqsubset x} D_{f_{\varphi_i}}|}{|D_x|} \tag{3.1}$$

If $\gamma \geq \gamma_{min}$, we say this pattern is discriminative. Let S_k be the set of length-k frequent patterns. Suppose the lowest support and the highest support in S_k is t_0 and t_h respectively. For every possible support value t, $t_0 \leq t \leq t_h$, we may calculate the number of patterns in S_k whose support is above t and of these patterns, how many of them have discriminative ratio great than γ_{min}.

Eventually we get a cut point t^* where p percentage of discriminative patterns are retained. $\psi(k)$ is then set at t^*. Using the above process, a user need not set the optimal support threshold any more.

We call the algorithm, shown in Figure 1, **AutoSupport**. AutoSupport only needs two parameters set by users, a discriminative ratio and a percentage cutoff. It will automatically adjust the support function according to the data distribution and the percentage cutoff. AutoSupport increases the support function to the extent where p percentage of discriminative patterns remain. This can reduce the number of patterns, especially those non-discriminative patterns substantially.

4 SeqIndex: A Sequence Indexing Algorithm

In this section, we present the SeqIndex algorithm. The algorithm can be divided into four steps: (1) discriminative feature selection, (2) index construction, (3) search, and (4) verification.

We first mine the frequent patterns, and use a discriminative ratio γ_{min} to filter out those redundant patterns. The output is a set of discriminative sequential patterns that will be used as indexing features. After that, we construct an *index tree* T, which is a prefix tree, to store and retrieve those features.

```
Algorithm AutoSupport
Input: A sequence database $D$,
       discriminative threshold $\gamma_{min}$,
       a percentage cutoff $p$,
       maximum subsequence length $L$.
Output: $\psi(k)$, size-increasing support function.

$\psi(1) = 1$;
for length $k$ from 2 to $L$
    $\psi(k) = \psi(k-1)$ ;
    do
        Mine the frequent length-$k$ patterns
        with $\psi(k)$;
        for each length-$k$ pattern, calculate $\gamma$;
        Calculate pattern distribution under
        different support $t$;
        Find a support $t^*$ that $p$% of all
        discriminative patterns remain;
        $\psi(k) = t^*$;
end for
return $\psi(k)$;
```

Figure 1: Determine Size-increasing Support Function

```
Algorithm DFS Search
Input: A sequence database $D$,
       Index tree $T$, Query $q$,
       maximum subsequence length $L$.
Output: Candidate set $C_q$.

Let $C_q = D$;
DFS Traverse index tree $T$ {
    Check the sequence from the root to node $p$
    if (it is a subsequence of $q$)
        Label it and visit its child node;
    else
        Skip $p$ and its subtree;
    Once arriving at a leaf node $OR$
    before skipping a subtree, do
        Find the deepest labelled node $p'$
        along this path;
        $C_q = C_q \cap D_{p'}$;
        if ($|C_q| < minCanSize$)
            Early termination;}
return $C_q$;
```

Figure 2: DFS Search and Candidate Set Computation

The most important part is the search algorithm, since an efficient search algorithm can improve the query processing time substantially. We discuss it in the following.

4.1 Search.

Given a query q, SeqIndex enumerates all its subsequences within the maximum subsequence length L and searches them in the index tree. For those subsequences found in the index tree, SeqIndex intersects their id lists to get a set of candidate sequences.

There could be a large number of subsequences contained in a given query q. To optimize the search process, we should try to reduce the number of subsequences that need to be checked.

Two optimization techniques [10] are studied extensively in SeqIndex to reduce the search space. One is Apriori pruning and the other is maximum discriminative sequential patterns. With these two optimization techniques in consideration, we propose an efficient algorithm for searching the index tree efficiently. We traverse the index tree in a depth first search manner. At each node p, we check the sequence from the root to p and label it as a candidate if it is a subsequence of query q. Then we visit its child nodes recursively. The reason we just label it but not intersect its id list immediately is that we want to check whether there is a maximum discriminative sequential pattern. If there is, it is unnecessary to intersect the id list of node p. On the other hand, if the sequence from the root to node p is not a subsequence of q, node p and its subtree can be pruned according to Apriori pruning. Once reaching a leaf node, or before skipping a node, we need to find out the deepest labelled node, which is the maximum discriminative sequential pattern along this path. Only its id list should be intersected with C_q while other "smaller" subsequences on this path can be skipped. The algorithm using the DFS search is shown in Figure 2.

Further optimization can be explored by the following intuition. When more and more intersections are executed, the candidate answer set becomes smaller and smaller. At some point of the traversal, we may find that C_q is already small enough compared with a user-specified threshold *minCandSize*. We can decide at this point to stop traversing the index tree. The candidate set C'_q so obtained is a superset of the optimal candidate set C_q which can be obtained by searching the whole tree. If the tree is large, C'_q may be acceptable since the savings in traversing the remaining part of the tree would outweigh the reduction of C_q by further intersections. We call this technique *early termination.*

Besides the DFS search method introduced above, we developed an alternative search algorithm that performs better for dense datasets. Instead of traversing the index tree in a depth-first search manner, at each level, we visit the node that has the smallest id list.

Since we choose a highly selective node to follow, the size of C_q will be reduced very quickly. After traversing a path, we can start over from the root, pick the second smallest child node and perform the same operation until we get an answer set with acceptable size. We call this search method *MaxSel* since we always greedily select the path with the highest selectivity.

The reason that two search algorithms are proposed is that we expect that their performance is correlated with data distribution. If the dataset has a small number of distinct items, the tree becomes very "thin" with a small branching factor and each node is associated with a long id list. In this case, MaxSel algorithm can easily find the most selective path and reduce the size of C_q quickly by skipping those nodes with a long list. On the other hand, if the dataset has a large number of distinct items, the tree becomes very "wide" with a large branching factor and each node is associated with only a small id list. In this case, MaxSel will spend a lot of time at each level searching for the node with the smallest id list. Thus, DFS search with early termination will perform better. We will compare these two algorithms to verify our reasoning in the experiments.

5 Experimental Results

In this section, we report our experimental results that validate the effectiveness and efficiency of SeqIndex. We compare SeqIndex with ItemIndex.

5.1 ItemIndex Algorithm ItemIndex builds index for each single item in the sequence database. For an item i, keep every occurrence of i in the sequence database with a tuple $\langle seq\_id, position\rangle$. Therefore, the index of an item i contains a list of such tuples.

Given a query $q = i_0 i_1 ... i_n$, ItemIndex will compute the intersection of all pairs of adjacent items in the query q. When intersecting the index list of i_j and i_{j+1}, if a pair of tuples from the two lists match in the *seq_id* part, we will further check if *position* in i_j list is smaller than *position* in i_{j+1} list. If so, we know that i_j and i_{j+1} occur in the same sequence and i_j precedes i_{j+1} in that sequence.

We compare the performance of SeqIndex and ItemIndex in terms of CPU time and I/O costs. The data is generated by a synthetic data generator provided by IBM[1]. More details are referred to [3].

5.2 SeqIndex vs. ItemIndex We compare SeqIndex and ItemIndex in terms of CPU time and I/O costs. Since we proposed two search algorithms – DFS Search and Maximum Selectivity Search, we will test SeqIndex with these two alternatives. In the following figures, we use **DFS** to denote SeqIndex with DFS search and **MaxSel** to denote SeqIndex with maximum selectivity search.

We first test how the execution time of SeqIndex and ItemIndex changes when we vary the number of distinct items in the sequence database. The sequence database has 10,000 sequences with an average of 30 items in each sequence. The number of distinct items is varied from 10 to 1000. The result is shown in Figure 3(a).

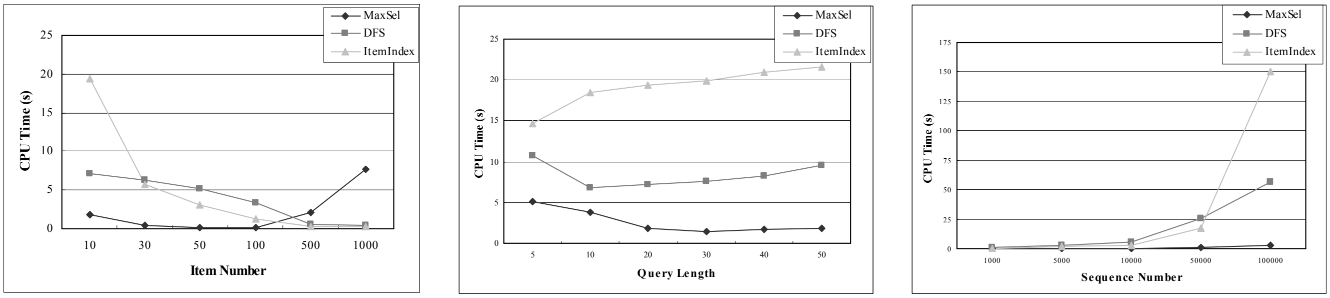
When the number of distinct items is very small, e.g. 10, MaxSel is the fastest, DFS stands in the middle while ItemIndex is the slowest. This is because the data distribution is very dense with a small number of items. MaxSel is effective by picking the most selective path, as we analyzed in the previous section.

As the number of items increases, CPU time of ItemIndex decreases dramatically and that of DFS also decreases but more slowly. When the number of items is 500, both DFS and ItemIndex run faster while MaxSel starts to slow down. This is due to sparser data and shorter id lists. On the other hand, the index tree of MaxSel turns to be bulky with lots of nodes at each level. Searching the most selective path involves visiting many nodes, thus becomes very inefficient. In this case, DFS turns out to be very efficient. This also gives us a hint – when the data is very dense, we can employ SeqIndex with maximum selectivity search; when the data is very sparse, we can switch to SeqIndex with DFS search. The "hybrid" SeqIndex will perform uniformly well while ItemIndex is quite sensitive to the data distribution.

The second experiment is to test the performance of SeqIndex and ItemIndex with varied size of queries. We test a database with 10,000 sequences. Each sequence has 30 items on average. The number of distinct items is 10. We vary the query length from 5 to 50. The result is shown in Figure 3(b).

Figure 3(b) shows that MaxSel performs very well as the query length increases. MaxSel first searches the index tree and produces a candidate set. Then it verifies those candidate sequences. When the query length is small, e.g. 5, there are more candidate sequences since it is usually easier to satisfy a short query. As the query length increases, the size of candidate set decreases since it is harder to satisfy a long query. That is why the performance of MaxSel improves as the query length increases. On the other hand, the performance of ItemIndex degrades as the query length increases. Since the number of intersections executed in ItemIndex is proportional to the query length, the execution time increases roughly linearly as the query length increases.

[1] http://www.almaden.ibm.com/cs/quest



(a) varying item number

(b) varying query length

(c) varying sequence number

Figure 3: Performance study

We also test the performance of SeqIndex and ItemIndex with varied database size. We vary the number of sequences in the sequence database from 1,000 to 100,000 (number of distinct items is 50 and the query length is 20). The result is shown in Figure 3(c). As we increase the number of sequences in the database. MaxSel has the best scalability, DFS stands in between and ItemIndex shows poor scalability.

We finally test how effective our algorithm AutoSupport is. Experimental results show that the index is more compact using AutoSupport in comparison with the uniform-support method. The mining time of AutoSupport also outperforms the uniform-support method significantly. The figure is omitted due to space limit.

6 Conclusions

In this paper, we broaden the scope of sequential pattern mining beyond the "narrowly defined" spectrum of knowledge discovery. Our study is focused on the design and construction of high-performance nonconsecutive sequence index structures in large sequence databases. A novel method, SeqIndex, is proposed, in which the selection of indices is based on the analysis of discriminative, frequent sequential patterns mined from large sequence databases. Such an analysis leads to the construction of compact and effective indexing structures. The effectiveness of the approach has been verified by our performance study.

References

[1] R. Agrawal, C. Faloutsos, and A.N. Swami. Efficient Similarity Search In Sequence Databases. In *Proceedings of the 4th International Conference of Foundations of Data Organization and Algorithms (FODO)*, pages 69–84, Chicago, Illinois, 1993.

[2] R. Agrawal, K.I. Lin, H.S. Sawhney, and K. Shim. Fast similarity search in the presence of noise, scaling, and translation in time-series databases. In *Proc. 1995 Int. Conf. on Very Large Databases*, pages 490–501, 1995.

[3] R. Agrawal and R. Srikant. Mining sequential patterns. In *Proc. 1995 Int. Conf. Data Engineering (ICDE'95)*, pages 3–14, March 1995.

[4] J. Ayres, J. E. Gehrke, T. Yiu, and J. Flannick. Sequential pattern mining using bitmaps. In *Proc. 2002 ACM SIGKDD Int. Conf. Knowledge Discovery in Databases (KDD'02)*, July 2002.

[5] K. Chan and A.W. Fu. Efficient time-series matching by wavelets. In *Proc. 15th IEEE Int. Conf. on Data Engineering*, pages 126–133, 1999.

[6] T. Kahveci and A.K. Singh. Efficient index structures for string databases. In *The VLDB Journal*, pages 351–360, 2001.

[7] E. Keogh. Exact indexing of dynamic time warping. In *Proc. 2002 Int. Conf. Very Large Data Bases (VLDB'02)*, Aug 2002.

[8] C. Meek, J.M. Patel, and S. Kasetty. Oasis: An online and accurate technique for local-alignment searches on biological sequences. In *Proc. 2003 Int. Conf. Very Large Data Bases (VLDB'03)*, Sept 2003.

[9] J. Pei, J. Han, B. Mortazavi-Asl, H. Pinto, Q. Chen, U. Dayal, and M.-C. Hsu. PrefixSpan: Mining sequential patterns efficiently by prefix-projected pattern growth. In *Proc. 2001 Int. Conf. Data Engineering (ICDE'01)*, pages 215–224, April 2001.

[10] X. Yan, P.S. Yu, and J. Han. Graph indexing: A frequent structure-based approach. In *Proc. 2004 Int. Conf. on Management of Data (SIGMOD'04)*, pages 253–264, 2004.

[11] M. Zaki. SPADE: An efficient algorithm for mining frequent sequences. *Machine Learning*, 40:31–60, 2001.

[12] Y. Zhu and D. Shasha. Warping indexes with envelope transforms for query by humming. In *Proc. 2003 Int. Conf. on Management of Data (SIGMOD'03)*, pages 181–192, 2003.