Non-contiguous Sequence Pattern Queries

Nikos Mamoulis and Man Lung Yiu

Department of Computer Science and Information Systems
University of Hong Kong
Pokfulam Road, Hong Kong
{nikos,mlyiu2}@csis.hku.hk

Abstract. Non-contiguous subsequence pattern queries search for symbol instances in a long sequence that satisfy some soft temporal constraints. In this paper, we propose a methodology that indexes long sequences, in order to efficiently process such queries. The sequence data are decomposed into tables and queries are evaluated as multiway joins between them. We describe non-blocking join operators and provide query preprocessing and optimization techniques that tighten the join predicates and suggest a good join order plan. As opposed to previous approaches, our method can efficiently handle a broader range of queries and can be easily supported by existing DBMS. Its efficiency is evaluated by experimentation on synthetic and real data.

1 Introduction

Time-series and biological database applications require the efficient management of long sequences. A sequence can be defined by a series of *symbol instances* (e.g., events) over a long timeline. Various types of queries are applied by the data analyst to recover interesting patterns and trends from the data. The most common type is referred to as "subsequence matching". Given a long sequence $\mathcal{T}$, a subsequence query q asks for all segments in $\mathcal{T}$ that match q. Unlike other data types (e.g., relational, spatial, etc.), queries on sequence data are usually *approximate*, since (i) it is highly unlikely for exact matching to return results and (ii) relaxed constraints can better represent the user requests.

Previous work on subsequence matching has mainly focused on (exact) retrieval of subsequences in $\mathcal{T}$ that contain or match *all* symbols of a query subsequence q [5,10]. A popular type of approximate retrieval, used mainly by biologists, is based on the *edit* distance [11,8]. In these queries, the user is usually interested in retrieving *contiguous* subsequences that approximately match *contiguous* queries. Recently, the problem of evaluating non-contiguous queries has been addressed [13]; some applications require retrieving a specific ordering of events (with exact or approximate gaps between them), without caring about the events which interleave them in the actual sequence. An example of such a query would be "find all subsequences where event a was transmitted approximately 10 seconds before b, which appeared approximately 20 seconds before c". Here, "approximately" can be expressed by an interval τ of allowed distances (e.g.,

$\tau_{a,b} = [9, 11]$ seconds), which may be of different length for different query components (e.g., $\tau_{a,b} = [9, 11]$ sec., $\tau_{b,c} = [18, 21]$ sec.). For such queries, traditional distance measures (e.g., Euclidean distance, edit distance) may not be appropriate for search, since they apply on contiguous sequences with fixed distances between consecutive symbols (e.g., strings).

In this paper, we deal with the problem of indexing long sequences in order to efficiently evaluate such non-contiguous pattern queries. In contrast to a previous solution [13], we propose a much simpler organization of the sequence elements, which, paired with query optimization techniques, allows us to solve the problem, using off-the-shelf database technology. In our framework, the sequence is decomposed into multiple tables, one for each symbol that appears in it. A query is then evaluated as a series of *temporal* joins between these tables. We employ temporal inference rules to tighten the constraints in order to speed-up query processing. Moreover, appropriate binary join operators are proposed for this problem. An important feature of these operators is that they are non-blocking; in other words, their results can be consumed at production time and temporary files are avoided during query processing. We provide selectivity and cost models for temporal joins, which are used by the query optimizer to define a good join order for each query.

The rest of the paper is organized as follows. Section 2 formally defines the problem and discusses related work. We present our methodology in Section 3. Section 4 describes a query preprocessing technique and provides selectivity and cost models for temporal joins. The application of our methodology to variants of the problem is discussed in Section 5. Section 6 includes an experimental evaluation of our methods. Finally, Section 7 concludes the paper.

2 Problem Definition and Related Work

2.1 Problem definition

Definition 1. *Let $\mathcal{S}$ be a set of* **symbols** *(e.g., event types). A* **sequence** $\mathcal{T}$ *is defined by a series of (s, t) pairs, where s is a symbol in $\mathcal{S}$ and t is a real-valued timestamp.*

As an example, consider an application that collects event transmissions from sensors. The set of event types defines $\mathcal{S}$. The sequence $\mathcal{T}$ is the collection of all transmissions over a long time. Figure 1 illustrates such a sequence. Here, $\mathcal{S} = \{a, b, c, d, e, f\}$ and $\mathcal{T} = \langle(a, 1.5), (c, 3), (d, 4.12), \ldots, (b, 32.14)\rangle$. Note that the definition is generic enough to include non-timestamped strings, where the distance between consecutive symbols is fixed. Given a long sequence $\mathcal{T}$, an analyst might want to retrieve the occurrences of interesting temporal patterns:

Definition 2. *Let $\mathcal{T}$ be a sequence defined over a set of symbols $\mathcal{S}$. A* **subsequence query pattern** *is defined by a connected directed graph $Q(V, E)$. Each node $n_i \in V$ is labeled with a symbol $l(n_i)$ from $\mathcal{S}$. Each (directed) edge $\langle n_i \rightarrow n_j\rangle$ in E is labeled by a* **temporal constraint** $\tau_{i,j}$ *modeling the allowed temporal distance $t(n_j) - t(n_i)$ between n_i and n_j in a query result. $\tau_{i,j}$ is defined*

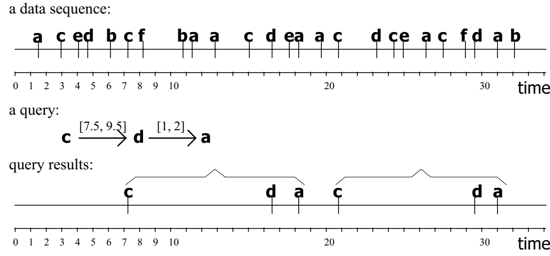


Fig. 1. A data sequence and a query

by an interval $[a_{i,j}, b_{i,j}]$ *of allowed values for* $t(n_j) - t(n_i)$. *The* **length** $|\tau_{i,j}|$ *of a temporal constraint* $\tau_{i,j}$ *is defined by the length of the corresponding temporal interval.*

Notice that a temporal constraint $\tau_{i,j}$ implies an equivalent $\tau_{j,i}$ (with the reverse direction), however, only one is usually defined by the user. A query example, illustrated in Figure 1, is $n_1 \to n_2 \to n_3$, $l(n_1) = c$, $l(n_2) = d$, $l(n_2) = a$, $\tau_{1,2} = [7.5, 9.5]$, $\tau_{2,3} = [1, 2]$. The lengths of $\tau_{1,2}$ and $\tau_{2,3}$ are $9.5 - 7.5 = 2$ and $2 - 1 = 1$ respectively.[1] This query asks for instances of c, followed by instances of d with time difference in the range $[7.5, 9.5]$, followed by instances of a with time difference in the range $[1, 2]$. Formally, a query result is defined as follows:

Definition 3. *Given a query* $Q(V, E)$ *with* N *vertices and a data sequence* $\mathcal{T}$, *a* **result** *of* Q *in* $\mathcal{T}$ *is defined by an instantiation* $\{n_1 \leftarrow (s_1, t_1), n_2 \leftarrow (s_2, t_2), \ldots n_N \leftarrow (s_N, t_N)\}$ *such that* $\forall 1 \leq i \leq N : (s_i, t_i) \in \mathcal{T} \wedge l(n_i) = s_i$ *and* $\forall \langle n_i \to n_j \rangle \in E : t_j - t_i \in \tau_{i,j}$.

Figure 1 shows graphically the results of the example query in the data sequence (notice that they include *non-contiguous* event patterns). It is possible (not shown in the current example) that two results share some common events. In other words, an event (or combination of events) may appear in more than one results. The sequence patterns search problem can be formally defined as follows:

Definition 4. (problem definition) *Given a query* $Q(V, E)$ *and a data sequence* $\mathcal{T}$, *the* **subsequence pattern retrieval problem** *asks for all results of* Q *in* $\mathcal{T}$.

Definition 2 is more generic than the corresponding query definition in [13], allowing the specification of binary temporal constraints between any pair of symbol instances. However, the graph should be connected, otherwise multiple queries (one for each connected component) are implied. As we will see in Section

[1] We note here that the length of a constraint $\tau_{i,j} = [a_{i,j}, b_{i,j}]$ in a discrete integer temporal domain is defined by $b_{i,j} - a_{i,j} + 1$.

4.1, additional temporal constraints can be derived for non-existing edges, and the existing ones can be further tightened using a *temporal constraint network minimization* technique. This allows for efficient query processing and optimization.

2.2 Related work

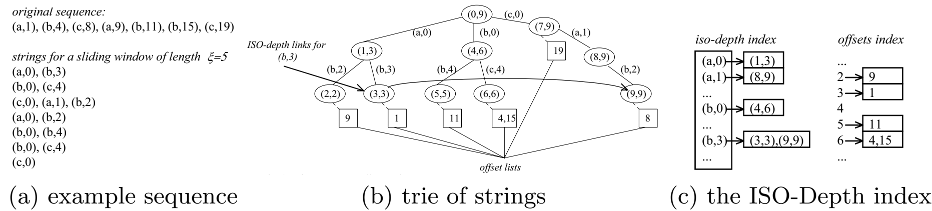
The subsequence matching problem has been extensively studied in time-series and biological databases, but for contiguous query subsequences [11,5,10]. The common approach is to slide a window of length w along the long sequence and index the subsequence defined by each position of the window. For time-series databases, the subsequences are transformed to high dimensional points in a Euclidean space and indexed by spatial access methods (e.g., R–trees). For biological sequences and string databases, more complex measures, like the *edit distance* are used. These approaches cannot be applied to our problem, since we are interested in non-contiguous patterns. In addition, search in our case is approximate; the distances between symbols in the query are not exact.

Wang et al. [13] were the first to deal with non-contiguous pattern queries. However, the problem definition there is narrower, covering only a subset of the queries defined in the previous section. Specifically, the temporal constraints are always between the first query component and the remaining ones (i.e., arbitrary binary constraints are not defined). In addition, the approximate distances are defined by an exact distance and a tolerance (e.g., a is 20 ± 1 seconds before b), as opposed to our interval-based definition. Although the interval-based and tolerance based definitions are equivalent, we prefer the interval-based one in our model, because inference operations can easily be defined, as we will see later.

[13] slide a temporal window of length ξ along the data sequence $\mathcal{T}$. Each symbol $s_0 \in \mathcal{T}$ defines a window position. The window at s_0 defines a *string* of pairs starting by $(s_0, 0)$ and containing (s, f) pairs, where s is a symbol and f is its distance from the previous symbol. The length of the string at s_0 is controlled by ξ; only symbols s with $t(s) - t(s_0) < \xi$ are included in it. Figure 2a shows an example sequence and the resulting strings after sliding a window of length $\xi = 5$.

The strings are inserted into a prefix tree structure (i.e., trie), which compresses their occurrences of the corresponding subsequences in $\mathcal{T}$. Each leaf of this trie stores a list of the positions in $\mathcal{T}$, where the corresponding subsequence exists; if most of the subsequences occur frequently in $\mathcal{T}$, a lot of space can be saved. The nodes of the trie are then labeled by a preorder traversal; node v is assigned a pair (v_s, v_m), where v_s is the preorder ID and v_m is the maximum preorder ID under the subtree rooted at v. From this trie, a set of *iso-depth lists* (one for each (s, d) pair, where s is a symbol and d is its offset from the beginning of the subsequence) are extracted. Figure 2b shows how the example strings are inserted into the trie and the iso-depth links for pair $(b, 3)$. These links are organized into consecutive arrays, which are used for pattern searching (see Figure 2c). For example, assume that we want to retrieve the results of query $\tau(c, a) = [1, 1]$ and $\tau(c, b) = [3, 3]$. We can use the ISO-Depth index to first

find the ID range of node $(c, 0)$, which is $(7, 9)$. Then, we issue a *containment query* to find the ID ranges of $(a, 1)$ within $(7, 9)$. For each qualifying range, $(8, 9)$ in the example, we issue a second containment query on $(b, 3)$ to retrieve the ID range of the result and the corresponding offset list. In this example, we get $(9, 9)$, which accesses in the right table of Fig. 2c the resulting offset 7. If some temporal constraints are approximate (e.g., $\tau(c, a)$ = [1,2]), in the next list a query is issued for each exact value in the approximate range (assuming a discrete temporal domain).



(a) example sequence (b) trie of strings (c) the ISO-Depth index

Fig. 2. Example of the ISO-Depth index [13]

This complex ISO-Depth index is shown in [13] to perform better than naive, exhaustive-search approaches. It can be adapted to solve our problem, as defined in Section 2.1. However, it has certain limitations. First, it is only suitable for *star* query graphs, where (i) the first symbol is temporally *before* all other symbols in the query and (ii) the only temporal constraints are between the first symbol and all others. Furthermore, there should be a total temporal order between the symbols of the query. For example, constraint $\tau_{a,b} = [-1, 1]$, implies that a can be before or after b in the query result. If we want to process this query using the ISO-Depth index, we need to decompose it to two queries: $\tau_{a,b} = [0, 1]$ and $\tau_{b,a} = [1, 1]$, and process them separately. If there are multiple such constraints, the number of queries that we need to issue may increase significantly. In the worst case, we have to issue $N!$ queries, where N is the number of vertices in the query graph. An additional limitation of the ISO-Depth index is that the temporal domain has to be discrete and coarse for trie compression to be effective. If the time domain is continuous, it is highly unlikely that any subsequence will appear exactly in $\mathcal{T}$ more than once. Finally, the temporal difference between two symbols in a query is restricted by ξ, limiting the use of the index. In this paper, we propose an alternative and much simpler method for storing and indexing long sequences, in order to efficiently process arbitrary non-contiguous subsequence pattern queries.

3 Methodology

In this section, we describe the data decomposition scheme proposed in this paper and a simple indexing scheme for it. We provide a methodology for query

evaluation and describe non-blocking join algorithms, which are used as components in it.

3.1 Storage organization

Since the queries search for relative positions of symbols in the data sequence $\mathcal{T}$, it is convenient to decompose $\mathcal{T}$ by creating one table T_s for each symbol s. The table stores the (ordered) positions of the symbol in the database. A *sparse* B^+–tree B_s is then built on top of it to accelerate range queries. The construction of the tables and indexes can be performed by scanning $\mathcal{T}$ once. At index construction, for each table T_s we need to allocate (i) one page for the file that stores T_s and (ii) one page for each level of its corresponding index B_s. The construction of T_a and B_a for symbol a can be illustrated in Figure 3 (the rest of the symbols are handled concurrently). While scanning $\mathcal{T}$, we can insert the symbol positions into the table. When a page becomes full, it is written to disk and a new pointer is added to the current page at the B^+–tree leaf page. When a B^+–tree node becomes full, it is flushed to disk and, in turn, a new entry is added at the upper level.

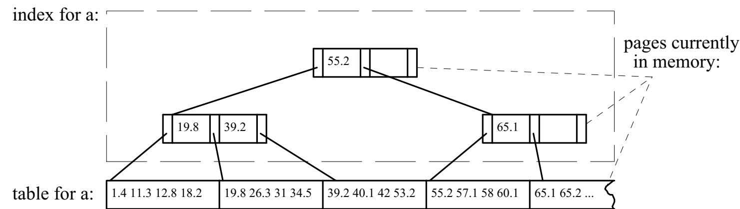


Fig. 3. Construction of the table and index for symbol a

Formally, the memory requirements for decomposing and indexing the data with a single scan of the sequence are $1 + \sum_{\forall s \in \mathcal{S}} (1 + h(B_s))$, where $h(B_s)$ is the height of the tree B_s that indexes T_s. For each symbol s, we only need to keep one page for each level of B_s plus one page of T_s. We also need one buffer page for the input. If the number of symbols is not extremely large, the system memory should be enough for this process. In a different case, the bulk-loading of indexes can be postponed and constructed at a second pass of each T_s.

3.2 Query evaluation

A pattern query can be easily transformed to a *multiway join query* between the corresponding symbol tables. For instance, to evaluate $\tau_{c,d} = [7.5, 9.5] \wedge \tau_{d,a} = [1, 2]$ we can first join table T_c with T_d using the predicate $\tau_{c,d} = [7.5, 9.5]$ and then the results with T_a using the predicate $\tau_{d,a} = [1, 2]$. This *evaluation plan* can be expressed by a tree $(T_c \bowtie_{\tau_{c,d}} T_d) \bowtie_{\tau_{d,a}} T_a$. Depending on the order and the algorithms used for the binary joins, there might be numerous query evaluation plans [12]. Following the traditional database query optimization approach,

we can transform the query to a tree of binary joins, where the intermediate results of each operator are fed to the next one [7]. Therefore, join operators are implemented as iterators that *consume* intermediate results from underlying joins and produce results for the next ones.

Like multiway spatial joins [9], our queries have a common join attribute in all tables (i.e., the temporal positions of the symbols). As we will see in Section 4.1, for each query, temporal constraints are inferred between every pair of nodes in the query graph. In other words, the query graph is *complete*. Therefore, the join operators also validate the temporal constraints that are not part of the binary join, but connect symbols from the left input with ones in the right one. For example, whenever the operator that joins $(T_c \bowtie_{\tau_{c,d}} T_d)$ with T_a using $\tau_{d,a}$ computes a result, it also validates constraint $\tau_{c,a}$, so that the result passed to the operator above satisfies all constraints between a,c, and d.

For the binary joins, the optimizer selects between two operators. The first is *index nested loops join* (INLJ). Since B$^+$–trees index the tables, this operator can be applied for all joins, where at least one of the joined inputs is a leaf of the evaluation plan. INLJ scans the left (outer) join input once and for each symbol instance applies a selection (range) query on the index of the right (inner) input according to the temporal constraint. For instance, consider the join $T_c \bowtie T_d$ with $\tau_{c,d} = [7.5, 9.5]$ and the instance $c = 3$. The range query applied on the index of d is $[10.5, 12.5]$. INLJ is most suitable when the left input is significantly smaller than the right one. In this case, many I/Os can be saved by avoiding accessing irrelevant data from the right input. This algorithm is non-blocking; it does not need to have the whole left input until it starts join processing. Therefore, join results can be produced before the whole input is available.

The second operator is *merge join* (MJ). MJ merges two sorted inputs and operates like the merging phase of external merge-sort algorithm [12]. The symbol tables are always sorted, therefore MJ can directly be applied for leaves of the evaluation plan. In our implementation of MJ, the output is produced sorted on the left input. The effect of this is that both INLJ and MJ produce results sorted on the symbol from the left input that is involved in the join predicate. Due to this property, MJ is also applicable for joining intermediate results, subject to memory availability, without blocking. The rationale is that joined inputs, produced by underlying operators, are not completely unsorted on their join symbol. A bound for the difference between consecutive values of their join symbol can be defined by the temporal constraints of the query.

More specifically, assume that MJ performs the join $L \bowtie R$ according to predicate $\tau_{x,y}$, where x is a symbol from the left input L and y is from the right input R. Assume also that L and R are sorted with respect to symbols l_L and l_R, respectively. Let p_L^1 and p_L^2 be two consecutive tuples in L. Due to constraint τ_{x,l_L}, we know that $p_L^2[x] \geq p_L^1[x] - |\tau_{x,l_L}|$, or else the next value of x that appears in L cannot be smaller than the previous one decremented by the length of constraint τ_{x,l_L}. Similarly, the difference between two values of y in R is bounded by $|\tau_{y,l_R}|$. Consider the example query of Figure 1 and assume that INLJ is used to process $T_c \bowtie T_d$. For each instance x_c of c in T_c, a range query

$[x_c + 7.5, x_c + 9.5]$ is applied on T_d to retrieve the qualifying instances of d. The join results (x_c, x_d) will be totally sorted only on x_c. Moreover, once we find a value x_d in the join result, we know that we cannot find any value smaller than $x_d - |\tau_{c,d}|$, next.

We use this bound to implement a non-blocking version of MJ, as follows. The *next*() iterator function to an input of MJ (e.g., L) keeps fetching results from it in a buffer until we know that the smallest value of the join key (e.g., x) currently in memory cannot be found in the next result (i.e., using the bound $|\tau_{x,l_L}|$, described above). Then, this smallest value is considered as the next item to be processed by the merge-join function, since it is guaranteed to be sorted.

If the binary join has low selectivity, or when the inputs have similar size, MJ is typically better than INLJ. Note that, since both INLJ and MJ are non-blocking, temporary results are avoided and the query processing cost is greatly reduced. For our problem, we do not consider hash-join methods (like the partitioned-band join algorithm of [4]), since the join inputs are (partially or totally) sorted, which makes merge-join algorithms superior.

An interesting property of MJ is that it can be extended to a *multiway* merge algorithm that joins all inputs synchronously [9]. The multiway algorithm can produce on-line results by scanning all inputs just once (for high-selective queries), however, it is expected to be slower than a combination of binary algorithms, since it may unnecessarily access parts of some inputs.

4 Query Transformation and Optimization

In order to minimize the cost of a non-contiguous pattern query, we need to consider several factors. The first is how to exploit inference rules of temporal constraints to tighten the join predicates and infer new, potentially useful ones for query optimization. The second is how to find a query evaluation plan that combines the join inputs in an optimal way, using the most appropriate algorithms.

4.1 Query transformation

A query, as defined in Section 2.1, is a connected graph, which may not be complete. Having a complete graph of temporal constraints between symbol instances can be beneficial for query optimization. Given a query, we can apply *temporal inference* rules to (i) derive implied temporal constraints between nodes of the query graph, (ii) tighten existing constraints, and even (iii) prove that the query cannot have any results, if the set of constraints is *inconistent*.

Inference of temporal constraints is a well-studied subject in Artificial Intelligence. Dechter et. al [3] provide a comprehensive study on solving *temporal constraint satisfaction problems* (TCSPs). Our query definitions 2 and 3 match the definition of a simple TCSP, where the constraints between problem variables (i.e., graph nodes) are simple intervals. In order to transform a user query to a *minimal temporal constraint network*, with no redundant constraints, we use the following operations (from [3]):

- *inversion*: $\overline{\tau_{i,j}} := \tau_{j,i}$. By *symmetry*, the inverse of a constraint $\tau_{i,j}$ is defined by $a_{j,i} = -b_{i,j}$ and $b_{j,i} = -a_{i,j}$.
- *intersection*: $\tau \cap \tau'$. The intersection of two constraints is defined by the values allowed by both of them. For constraints $\tau_{i,j}$ and $\tau'_{i,j}$ on the same edge, intersection $\tau_{i,j} \cap \tau'_{i,j}$ is defined by $[\max\{a_{i,j}, a'_{i,j}\}, \min\{b_{i,j}, b'_{i,j}\}]$.
- *composition*: $\tau \propto \tau'$. The composition of two constraints allows all values w such that there is a value v allowed by τ, a value u allowed by τ' and $v+u = w$. Given two constraints $\tau_{i,j}$ and $\tau_{j,k}$, sharing node n_j, their composition $\tau_{i,j} \propto \tau_{j,k}$ is defined by $[a_{i,j} + a_{j,k}, b_{i,j} + b_{j,k}]$

Inversion is the simplest form of inference. Given a constraint $\tau_{i,j}$, we can immediately infer constraint $\tau_{j,i}$. For example if $\tau_{c,d} = [7.5, 9.5]$, we know that $\tau_{d,c} = [-9.5, -7.5]$. Composition is another form of inference, which exploits *transitivity* to infer constraints between nodes, which are not connected in the original graph. For example, $\tau_{c,d} = [7.5, 9.5] \wedge \tau_{d,a} = [1, 2]$ implies $\tau'_{c,a} = [8.5, 11.5]$. Finally, intersection is used to unify (i.e., *minimize*) the constraints for a given pair of nodes. For example, an original constraint $\tau_{c,a} = [8, 10]$ can be tightened to $[8.5, 10]$, using an inferred constraint $\tau'_{c,a} = [8.5, 11.5]$. After an intersection operation, a constraint $\tau_{i,j}$ can become *inconsistent* if $a_{i,j} > b_{i,j}$.

A temporal constraint network (i.e., a query in our setting) is *minimal* if no constraints can be tightened. It is *inconsistent* if it contains an inconsistent constraint. The goal of the query transformation phase is to either minimize the constraint network or prove it inconsistent. To achieve this goal we can employ an adaptation of Floyd-Warshall's all-pairs-shortest-path algorithm [6] with $O(N^3)$ cost, N being the number of nodes in the query. The pseudocode of this algorithm is shown in Figure 4. First, the constraints are initialized by (i) introducing inverse temporal constraints for existing edges and (ii) assigning "dummy" constraints to non-existing edges. The nested for-loops correspond to Floyd-Warshall's algorithm, which essentially finds for all pairs of nodes the lower constraint bound (i.e., shortest path) and the upper constraint bound (i.e., longest path). If some constraint is found inconsistent, the algorithm terminates and reports it. As shown in [3] and [6], the algorithm of Figure 4 computes the minimal constraint network correctly.

4.2 Query Optimization

In order to find the optimal query evaluation plan, we need accurate join selectivity formulae and cost estimation models for the individual join operators.

The selectivity of a join in our setting can be estimated by applying existing models for spatial joins [9]. We can model the join $L \bowtie R$ as a set of selections on R, one for each symbol in L. If the distribution of the symbol instances in R is uniform, the selectivity of each selection can be easily estimated by dividing the temporal range of the constraint by the temporal range of the data sequence. For non-uniform distributions, we extend techniques based on histograms. Details are omitted due to space constraints.

Estimating the costs of INLJ and MJ is quite straightforward. First, we have to note that a non-leaf input incurs no I/Os, since the operators are non-blocking.

```
boolean Query_Transformation(query Q(V, E))
    for each pair of nodes ⟨n_i, n_j⟩
        if ⟨n_i → n_j⟩ ∈ E then τ_{j,i} := overline(τ_{i,j}); //inversion
        if n_i is not connected to n_j then τ_{i,j} := τ_{j,i} := [−∞, ∞];
    for k := 1 to N
        for i := 1 to N
            for j := i + 1 to N
                τ_{i,j} := τ_{i,j} ∩ (τ_{i,k} ∝ τ_{k,j});
                if a_{i,j} > b_{i,j} then return false; //inconsistent query
                τ_{j,i} := overline(τ_{i,j});
    return true; //consistent query
```

Fig. 4. Query transformation using Floyd-Warshall's algorithm

Therefore, we need only estimate they I/Os by INLJ and MJ for leaf inputs of the evaluation plan. Essentially, MJ reads both inputs once, thus its I/O cost is equal to the size of the leaf inputs. INLJ performs a series of selections on a B^+-tree. If an LRU memory buffer is used for the join, the index pages accessed by a selection query are expected to be in memory with high probability due to the previous query. This, because instances of the left input are expected to be sorted, or at least partially sorted. Therefore, we only need to consider the number of *distinct* pages of R accessed by INLJ.

An important difference between MJ and INLJ is that most accesses by MJ are sequential, whereas INLJ performs mainly random accesses. Our query optimizer takes this under consideration. From its application, it turns out that the best plans are left-deep plans, where the lower operators are MJ and the upper ones INLJ. This is due to the fact that our multiway join cannot benefit from the few intermediate results of bushy plans, since they are not materialized (recall that the operators are non-blocking). The upper operators of a left-deep plan have a small left input, which is best handled by INLJ.

5 Application to Problem Variants

So far, we have assumed that there is only one data sequence $\mathcal{T}$ and that the indexed symbols are relatively few with a significance number of appearances in $\mathcal{T}$. In this section we discuss how to deal with more general cases with respect to these two factors.

5.1 Indexing and querying multiple sequences

If there are multiple small sequences, we can concatenate them to a single long sequence. The difference is that now we treat the beginning time of one sequence as the end of the previous one. In addition, we add a long temporal gap W, corresponding to the maximum sequence length (plus one time unit), between

every pair of sequences in order to avoid query results, composed of symbols that belong to different sequences.

For example, consider three sequences: $\mathcal{T}_1 = \langle (b,1),(a,3.5),(d,4.5),(a,6)\rangle$, $\mathcal{T}_2 = \langle (a,0.5),(d,3),(b,9.5)\rangle$, and $\mathcal{T}_3 = \langle (c,2),(a,3.5),(b,4)\rangle$. Since the longest sequence $\mathcal{T}_2$ has length 9, we can convert all of them to a single long sequence $\mathcal{T} = \langle (b,0),(a,2.5),(d,3.5),(a,5),(a,20),(d,22.5),(b,29),(c,40),(a,41.5),(b,42)\rangle$.

Observe that in this conversion, we have (i) computed the maximum sequence length and added a time unit to derive $W = 10$ and (ii) shifted the sequences, so that sequence $\mathcal{T}_i$ begins at $(i-1)*2W$. The differences between events in the same sequence have been retained. Therefore, by setting the maximum possible distance between any pair of symbols to W, we are able to apply the methodology described in the previous sections for this problem. If the maximum sequence length is unknown at index construction time (e.g., when the data are online), we can use a large number for W that reflects the maximum anticipated sequence length.

Alternatively, if someone wants to find patterns, where the symbols appear in *any* data sequence, we can simply merge the events of all sequences treating them as if they belonged to the same one. For example, merging the sequences $\mathcal{T}_1$–$\mathcal{T}_3$ above would result in $\mathcal{T} = \langle (a,0.5),(b,1),(c,2),(d,3),(a,3.5),(a,3.5),(b,4),\dots\rangle$.

5.2 Handling infrequent symbols

If some symbols are not frequent in $\mathcal{T}$, disk pages may be wasted after the decomposition. However, we can treat all decomposed tables as a single one, after determining an ordering of the symbols (e.g., alphabetical order). Then, occurrences of all symbols are recorded in a single table, sorted first by symbol and then by position. This table can be indexed using a B$^+$–tree in order to facilitate query processing. We can also use a second (header) index on top of the sorted table, that marks the first position of each symbol. This structure resembles the *inverted file* used in Information Retrieval systems [1] to record the occurrences of index terms in documents.

5.3 Indexing and querying patterns in DBMS tables

In [13], non-contiguous sequence pattern queries have been used to assist exploration of DNA Micro-arrays. A DNA micro-array is an expression matrix that stores the expression level of genes (rows) in experimental samples (columns). It is possible to have no result about some gene-sample combinations. Therefore, the micro-array can be considered as a DBMS table with NULL values.

We can consider each row of this table as a sequence, where each non-NULL value v at column s is transformed to a (s,v) pair. After sorting these pairs by v, we derive a sequence which reflects the expression difference between pairs of samples on the same gene. If we concatenate these sequences to a single long one, using the method described in Section 5.1, we can formulate the problem of finding genes with similar differences in their expression levels as a subsequence pattern retrieval problem.

Figure 5 illustrates. The leftmost table corresponds to the original microarray, with the expression levels of each gene to the various samples. The middle table shows how the rows can be converted to sequences and the sequence of Figure 5c is their concatenation. As an example, consider the query "find all genes, where the level of sample s_1 is lower than that of s_2 at some value between 20 and 30, and in the level of sample s_2 is lower than that of s_3 at some value between 100 and 130". This query would be expressed by the following subsequence query pattern on the transformed data: $\tau_{s_1,s_2} = [20, 30] \wedge \tau_{s_2,s_3} = [100, 130]$.

	s_1	s_2	s_3
g_1	50	30	NULL
g_2	190	NULL	120
g_3	15	105	150
...	...	...	...

(a) A DBMS table

g_1	$\langle(s_2, 30), (s_1, 50)\rangle$
g_2	$\langle(s_3, 120), (s_1, 150)\rangle$
g_3	$\langle(s_1, 15), (s_2, 105), (s_3, 150)\rangle$
...	...

(b) Transformed sequences

$(s_2, 0), (s_1, 20),$
$(s_3, 400), (s_1, 430),$
$(s_1, 800), (s_2, 890), (s_3, 935)$
...

(c) Single sequence ($W = 200$)

Fig. 5. Converting a DBMS table, domain=$[0, 200)$

6 Experimental Evaluation

Our framework, denoted by SeqJoin thereafter, and the ISO-Depth index method were implemented in C++ and tested on a Pentium-4 2.3GHz PC. We set the page (and B^+–tree node) size to 4Kb and used an LRU buffer of 1Mb. To smoothen the effects of randomness in the queries, all experimental results (except from the index creation) were averaged over 50 queries with the same parameters.

For comparison purposes, we generated a number of data sequences $\mathcal{T}$ as follows. The positions of events in $\mathcal{T}$ are integers, generated uniformly along the sequence length; the average difference of consecutive events was controlled by a parameter $\overline{G}$. The symbol that labels each event was chosen among a set of $\mathcal{S}$ symbols according to a Zipf distribution with a parameter θ. Synthetic datasets are labeled by D$|\mathcal{T}|$-G$\overline{G}$-A$|\mathcal{S}|$-Sθ. For instance, label D1M-G100-A10-S1 indicates that the sequence has 1 million events, with 100 average gap between two consecutive ones, 10 different symbols, whose frequencies follow a Zipf distribution with skew parameter $\theta = 1$. Notice that $\theta = 0$ implies that the labels for the events are chosen uniformly at random.

We also tested the performance of the algorithms with real data. Gene expression data can be viewed as a matrix where a row represents a gene and a column represents the condition. From [2], we obtained two gene expression matrices (i) a Yeast expression matrix with 2884 rows and 17 columns, and (ii) a Human expression matrix with 4026 rows and 96 columns. The domains of Yeast and Human datasets are $[0, 595]$ and $[-628, 674]$ respectively. We converted the above data to event sequences as described in Section 5.3 (note that [13] use the same conversion scheme).

The generated queries are star and chain graphs connecting random symbols with soft temporal constraints. Thus, in order to be fair in our comparison

with ISO-Depth, we chose to generate only queries that satisfy the restrictions in [13]. Chain graph queries with positive constraint ranges can be converted to star queries, after inferring all the constraints between the first symbol and the remaining ones. On the other hand, it may not be possible to convert random queries to star queries without inducing overlapping, non-negative constraints. Note that these are the best settings for the ISO-Depth index, since otherwise queries have to be transformed to a large number subqueries, one for each possible order of the symbols in the results. The distribution of symbols in a generated query is a Zipfian one with skew parameter $Sskew$. In other words, some symbols have higher probability to appear in the query according to the skew parameter. A generated constraint has average length $\overline{R}$ and ranges from $0.5 \cdot \overline{R}$ to $1.5 \cdot \overline{R}$.

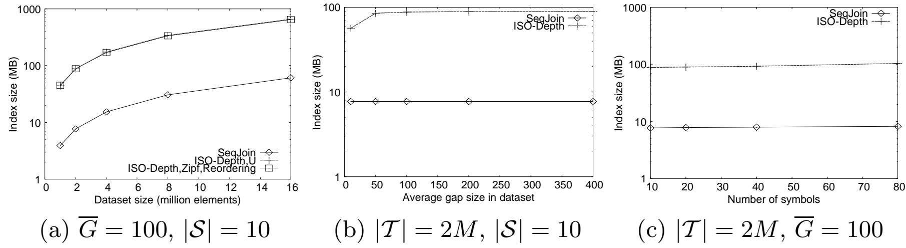
6.1 Size and construction cost of the indexes

In the first set of experiments, we compare the size and construction cost of the data structures used by the two methods (SeqJoin and ISO-Depth) as a function of three parameters; the size of $\mathcal{T}$ (in millions of elements), the average gap $\overline{G}$ between two consecutive symbols in the sequence, and the number $|\mathcal{S}|$ of distinct symbols in the sequence. We used uniform symbol frequencies ($\theta = 0$) in $\mathcal{T}$ and skewed frequencies ($\theta = 1$). Since the size and construction cost of SeqJoin is independent of the skewness of symbols in the sequence, we compare three methods here (i) SeqJoin, (ii) simple ISO-Depth (for uniform symbol frequencies), and (iii) ISO-Depth with reordering [13] (for skewed symbol frequencies).

Figure 6 plots the sizes of the constructed data structures after fixing two parameter values and varying the value of the third one. Observe that ISO-Depth with and without reordering have similar sizes on disk. Moreover, the size of the structures depends mainly on the database size, rather on the other parameters. The size of the ISO-Depth structures is roughly ten times larger than that of the SeqJoin data structures. The SeqJoin structures are smaller than the original sequence (note that one element of $\mathcal{T}$ occupies 8 bytes). A lot of space is saved because the symbol instances are not repeated; only their positions are stored and indexed. On the other hand, the ISO-Depth index stores a lot of redundant information, since a subsequence is defined for each position of the sliding window (note that $\xi = 4 \cdot \overline{G}$ for this experiment). The size difference is insensitive to the values of the various parameters.

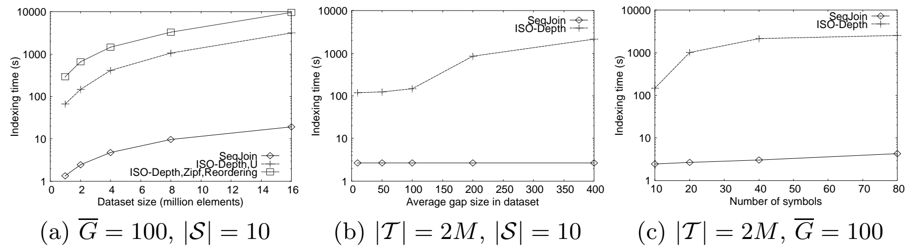
Figure 7 plots the construction time for the data structures used by the two methods. The construction cost for ISO-Depth is much higher than that of SeqJoin and further increases when reordering is employed. The costs for both methods increase proportionally to the database size, as expected. However, observe that the cost for SeqJoin is almost insensitive to the average gap between symbols and to the number of distinct symbols in the sequence. On the other hand, there is an obvious cost increase in the cost of ISO-Depth with $\overline{G}$ due to the low compression the trie achieves for large gaps between symbols. There is also an increase with the number of distinct symbols, due to the same reason.

Table 1 shows the corresponding index size and construction cost for the real datasets used in the experiments. Observe that the difference between the two



(a) $\overline{G} = 100$, $|\mathcal{S}| = 10$ (b) $|\mathcal{T}| = 2M$, $|\mathcal{S}| = 10$ (c) $|\mathcal{T}| = 2M$, $\overline{G} = 100$

Fig. 6. Index size on disk (synthetic data)



(a) $\overline{G} = 100$, $|\mathcal{S}| = 10$ (b) $|\mathcal{T}| = 2M$, $|\mathcal{S}| = 10$ (c) $|\mathcal{T}| = 2M$, $\overline{G} = 100$

Fig. 7. Index construction time (synthetic data)

methods is even higher compared to the synthetic data case. The large construction cost is a significant disadvantage of the ISO-Depth index, which adds to the fact that it cannot be dynamically updated. If the data sequence is frequently updated (e.g., consider on-line streaming data from sensor transmissions), the index has to be built from scratch with significant overhead. On the other hand, our symbol tables T_s and B$^+$–trees can be efficiently updated incrementally. The new event instances are just appended to the corresponding tables. Also, in the worst case only the rightmost paths of the indexes are affected by an incremental change (see Section 3.1).

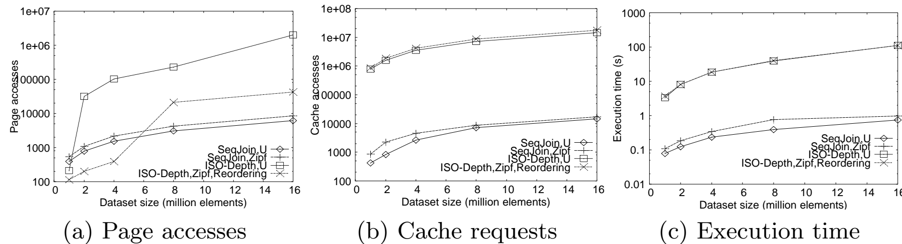
Dataset	Method	Index Size (Mb)	Construction time (s)
Yeast	SeqJoin	0.3	0.4
	ISO-Depth	5.5	4.2
Human	SeqJoin	2.14	3.1
	ISO-Depth	251	416.2

Table 1. Index size and construction time (real data)

6.2 Experiments with synthetic data

In this paragraph, we compare the search performance of the two methods on generated synthetic data. Unless otherwise stated, the dataset used is D2M-G100-A10-S0, the default parameters for queries are $\overline{R} = 50$, $Sskew = 0$, and the number N of nodes in the query graphs is 4.

Figure 8 shows the effect of database size on the performance of the two algorithms in terms of page accesses, memory buffer requests, and overall execution time. For each length of the data sequence we tested the algorithms on both uniform ($Sskew = 0$) and Zipfian ($Sskew = 1$) symbol distributions. Figure 8a shows that SeqJoin outperforms ISO-Depth in terms of I/O in most cases, except for small datasets with skewed distribution of symbols. The reason behind this unstable performance of ISO-Depth, is that the I/O cost of this algorithm is very sensitive to the memory buffer. Skewed queries on small datasets access a small part of the iso-depth lists with high locality and cache congestion is avoided. On the other hand, for uniform symbol distributions or large datasets the huge number of cache requests by ISO-Depth (see Figure 8b), incur excessive I/O. Figure 8c plots the overall execution cost of the algorithms; SeqJoin is one to two orders of magnitude faster than ISO-Depth. Due to the relaxed nature of the constraints, ISO-Depth has to perform a huge number of searches.[2]



(a) Page accesses (b) Cache requests (c) Execution time

Fig. 8. Performance with respect to the data sequence length

Figure 9 compares the performance of the two methods with respect to several system, data, and query parameters. Figure 9a shows the effect of cache size (i.e., memory buffer size) on the I/O cost of the two algorithms. Observe that the I/O cost of SeqJoin is almost constant, while the number of page accesses by ISO-Depth drops as the cache size increases. ISO-Depth performs a huge number of searches in the iso-depth lists, with high locality between them. Therefore, it is favored by large memory buffers. On the other hand, SeqJoin is insensitive to the available memory (subject to a non-trivial buffer) because the join algorithms scan the position tables and indexes at most once. Even though ISO-Depth outperforms SeqJoin in terms of I/O for large buffers, its excessive computational cost (which is almost insensitive to memory availability) dominates the overall execution time. Moreover, most of the page accesses of ISO-Depth are random, whereas the algorithm that accesses most of the pages for SeqJoin is MJ (at the lower parts of the evaluation plan), which performs mainly sequential accesses.

[2] In fact, the cost of ISO-Depth for this class of approximate queries is even higher than that of a simple linear scan algorithm, as we have seen in our experiments.

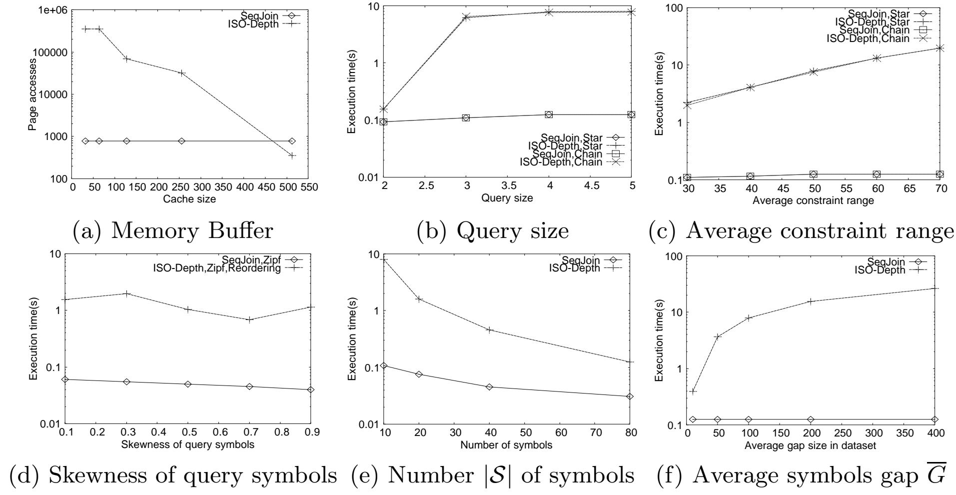
Figure 9b plots the execution cost of SeqJoin and ISO-Depth as a function of the number of symbols in the query. For trivial 2-symbol queries, both methods have similar performance. However, for larger queries the cost of ISO-Depth explodes, due to the excessive number of iso-depth list accesses it has to perform. For an average constraint length $\overline{R}$, the worst-case number of accesses is $\overline{R}^{N-1}$, where N is the number of symbols in the query. Since the selectivity of the queries is high, the majority of the searches for the third query symbol fail, and this is the reason why the cost does not increase much for queries with more than three symbols.

Figure 9c shows how the average constraint length $\overline{R}$ affects the cost of the algorithms. The cost of SeqJoin is almost independent of this factor. However, the cost of ISO-Depth increases superlinearly, since the worst-case number of accesses is $\overline{R}^{N-1}$, as explained above. We note that for this class of queries the cost of ISO-Depth in fact increases quadratically, since most of the searches after the third symbol fail. Figure 9d shows how $Sskew$ affects the cost of the two methods, for star queries. The cost difference is maintained for a wide range of symbol frequency distributions. In general, the efficiency of both algorithms increases as the symbol occurrence becomes more skewed for different reasons. SeqJoin manages to find a good join ordering, by joining the smallest symbol tables first. ISO-Depth exploits the symbol frequencies in the trie construction to minimize the potential search paths for a given query, as also shown in [13]. The fluctuations are due to the randomness of the queries. Figure 9e shows the effect of the number of distinct symbols in the data sequence. When the number of symbols increases the selectivity of the query becomes higher and the cost of both methods decreases; ISO-Depth has fewer paths to search and SeqJoin has smaller tables to join. SeqJoin maintains its advantage over ISO-Depth, however, the cost difference decreases slightly.

Finally, Figure 9f shows the effect of the average gap between consecutive symbol instances in the sequence. In this experiment, we set the average constraint length $\overline{R}$ in the queries equal to $\overline{G}/2$ in order to maintain the same query selectivity for the various values of $\overline{G}$. The cost of SeqJoin is insensitive to this parameter, since the size of the joined tables and the selectivity of the query is maintained with the change of $\overline{G}$. On the other hand, the performance of ISO-Depth varies significantly for two reasons. First, for datasets with small values of $\overline{G}$, ISO-Depth achieves higher compression, as the probability for a given subsequence to appear multiple times in $\mathcal{T}$ increases. Higher compression ratio results in a smaller index and lower execution cost. Second, the number of search paths for ISO-Depth increase significantly with $\overline{G}$, because of the increase of $\overline{R}$ with the same rate. In summary, ISO-Depth can only have competitive performance to SeqJoin for small gaps between symbols and small lengths of the query constraints.

6.3 Experiments with real data

Figure 10 shows the performance of SeqJoin and ISO-Depth on real datasets. In both Yeast and Human datasets, SeqJoin has significantly low cost, in terms of I/Os, cache requests, and execution time. For these real datasets, we need to

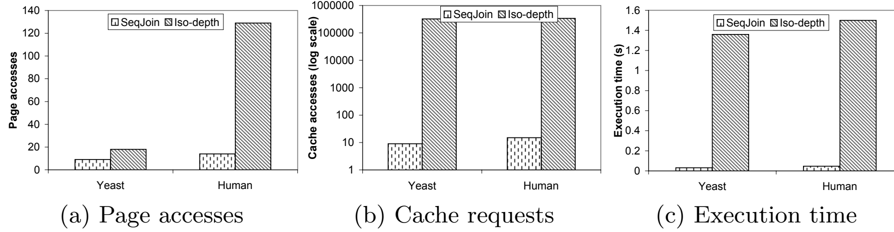


(a) Memory Buffer (b) Query size (c) Average constraint range

(d) Skewness of query symbols (e) Number $|\mathcal{S}|$ of symbols (f) Average symbols gap $\overline{G}$

Fig. 9. Performance comparison under various factors

slide a window ξ as long as the largest difference between a pair of values in the same row. In other words, the indexed rows of the expression matrices have an average length of $\frac{|\mathcal{S}|+1}{2}$. Thus, for these real datasets, the ISO-Depth index could not achieve high compression. For instance, the converted weighted sequence from Human dataset only has 360K elements but it has a ISO-Depth index of comparable size as that of synthetic data with 8M elements. In addition, the approximate queries (generated according to the settings of Section 6.2) follow a large number of search paths in the ISO-Depth index.



(a) Page accesses (b) Cache requests (c) Execution time

Fig. 10. Random queries against real datasets

7 Conclusions and Future Work

In this paper, we presented a methodology of decomposing, indexing and searching long symbol sequences for non-contiguous sequence pattern queries. SeqJoin has significant advantages over ISO-Depth [13], a previously proposed method for this problem, including:

– It can be easily implemented in a DBMS, utilizing many existing modules.
– The tables and indexes are much smaller than the original sequence and they can be incrementally updated.
– It is very appropriate for queries with approximate constraints. On the other hand, the ISO-Depth index generates a large number of search paths, one for each exact query included in the approximation.
– It is more general since (i) it can deal with real-valued timestamped events, (ii) it can handle queries with approximate constraints between any pair of objects, and (iii) the maximum difference between any pair of query symbols is not bounded.

The contributions of this paper also include the modeling of a non-contiguous pattern query as a graph, which can be refined using temporal inference, and the introduction of a non-blocking merge-join algorithm, which can be used by the query processor for this problem. In the future, we plan to study the evaluation of this class of queries on unbounded and continuous event sequences from a stream in a limited memory buffer.

References

1. R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. ACM and Mc-Graw Hill, 1999.
2. Y. Cheng and G. M. Church. Biclustering of expression data. In *Proc. of International Conference on Intelligent Systems for Molecular Biology*, 2000.
3. R. Dechter, I. Meiri, and J. Pearl. Temporal constraint networks. *Artificial Intelligence*, 49(1–3):61–95, 1991.
4. D. J. DeWitt, J. F. Naughton, and D. A. Schneider. An evaluation of non-equijoin algorithms. In *Proc. of VLDB Conference*, 1991.
5. C. Faloutsos, M. Ranganathan, and Y. Manolopoulos. Fast subsequence matching in time-series databases. In *Proc. of ACM SIGMOD International Conference on Management of Data*, 1994.
6. R. W. Floyd. ACM Algorithm 97: Shortest path. *Communications of the ACM*, 5(6):345, June 1962.
7. G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170, 1993.
8. T. Kahveci and A. K. Singh. Efficient index structures for string databases. In *Proc. of VLDB Conference*, 2001.
9. N. Mamoulis and D. Papadias. Multiway spatial joins. *ACM Transactions on Database Systems (TODS)*, 26(4):424–475, 2001.
10. Y.-S. Moon, K.-Y. Whang, and W.-S. Han. General match: a subsequence matching method in time-series databases based on generalized windows. In *Proc. of ACM SIGMOD International Conference on Management of Data*, 2002.
11. G. Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31–88, 2001.
12. R. Ramakrishnan and J. Gehrke. *Database Management Systems*. Mc-Graw Hill, third edition, 2003.
13. H. Wang, C.-S. Perng, W. Fan, S. Park, and P. S. Yu. Indexing weighted-sequences in large databases. In *Proc. of Int'l Conf. on Data Engineering (ICDE)*, 2003.