General Match: A Subsequence Matching Method in Time-Series Databases Based on Generalized Windows

Yang-Sae Moon, Kyu-Young Whang, and Wook-Shin Han
Department of Computer Science and
Advanced Information Technology Research Center (AITrc)
Korea Advanced Institute of Science and Technology (KAIST)
Taejon, Korea
{ysmoon,kywhang,wshan}@mozart.kaist.ac.kr

ABSTRACT

We generalize the method of constructing windows in subsequence matching. By this generalization, we can explain earlier subsequence matching methods as special cases of a common framework. Based on the generalization, we propose a new subsequence matching method, *General Match*. The earlier work by Faloutsos et al. (called *FRM* for convenience) causes a lot of false alarms due to lack of *point-filtering effect*. *Dual Match*, recently proposed as a dual approach of FRM, improves performance significantly over FRM by exploiting point filtering effect. However, it has the problem of having a smaller allowable window size—half that of FRM—given the minimum query length. A smaller window increases false alarms due to *window size effect*. General Match offers advantages of both methods: it can reduce window size effect by using large windows like FRM and, at the same time, can exploit point-filtering effect like Dual Match. General Match divides data sequences into generalized sliding windows (*J-sliding windows*) and the query sequence into generalized disjoint windows(*J-disjoint windows*). We formally prove that General Match is correct, i.e., it incurs no false dismissal. We then propose a method of estimating the optimal value of the *sliding factor* J that minimizes the number of page accesses. Experimental results for real stock data show that, for low selectivities (10^{-6}~10^{-4}), General Match improves average performance by 117% over Dual Match and by 998% over FRM; for high selectivities (10^{-3}~10^{-1}), by 45% over Dual Match and by 64% over FRM. The proposed generalization provides an excellent theoretical basis for understanding the underlying mechanisms of subsequence matching.

1. INTRODUCTION

Time-series data are of growing importance in many new database applications such as data mining and data warehousing [14]. A time-series is a sequence of real numbers representing values at specific points in time. Typical examples of time-series data include stock prices, exchange rates, biomedical measurements, and weather data. The time-series data stored in a database are called *data sequences*. Finding data sequences similar to the given *query sequence* from the database is called *similar sequence matching* [1, 8]. Owing to faster computing speed and larger storage devices, there has been a number of efforts to utilize the large amount of time-series data. Accordingly, similar sequence matching has become an important research topic in data mining [1, 2, 8, 9, 12, 13].

Various similarity models have been studied in similar sequence matching. In this paper, we use the similarity model based on the Euclidean distance [1, 5, 8, 10]. In this model, two sequences $X = \{X[1], ..., X[n]\}$ and $Y = \{Y[1], ..., Y[n]\}$ of the same length n are said to be *similar* if the Euclidean distance $D(X, Y)$ $(= \sqrt{\sum_{i=1}^{n}(X[i] - Y[i])^2})$ is less than or equal to the user-specified *tolerance* ϵ [1]. More specifically, we define that two sequences X and Y are in ϵ-*match* if $D(X, Y)$ is less than or equal to ϵ. We define *n-dimensional distance computation* as the operation that computes the distance between two sequences of length n.

Similar sequence matching can be classified into two categories [8]:

- *Whole matching* [1]: Given N data sequences $S_1, ..., S_N$, a query sequence Q, and the tolerance ϵ, we find those data sequences that are in ϵ-match with Q. Here, the data and query sequences must have the same length.
- *Subsequence matching* [8, 10]: Given N data sequences $S_1, ..., S_N$ of varying lengths, a query sequence Q, and the tolerance ϵ, we find all the sequences S_i, one or more subsequences of which are in ϵ-match with Q, and the offsets in S_i of those subsequences.

Thus, subsequence matching is a generalization of whole matching [5, 6, 8, 18]. In this paper, we focus on subsequence matching.

Subsequence matching methods [8, 10] consist of index building and subsequence matching algorithms. In the index building algorithm, data sequences are divided into windows of size ω, and each window is transformed to a point in an f-dimensional space ($f \ll \omega$, we call it *lower-dimensional transformation*). Then, the transformed points are stored in a multidimensional index. In the subsequence matching algorithm, a query sequence is transformed into windows of size ω, and each window is transformed to an f-dimensional

ACM SIGMOD '2002 June 4-6, Madison, Wisconsin, USA

point. Then, range queries are constructed using the transformed points and the tolerance ϵ. By evaluating the range queries using the index, the *candidates* that are potentially in ϵ-match with the query sequence are identified. The result is subsequently refined by accessing the database and by selecting only those subsequences that are in ϵ-match with the query sequence.

Faloutsos et al. [8] have proposed a subsequence matching method that divides data sequences into *sliding windows* and the query sequence into *disjoint windows* (we simply call this solution *FRM* by taking the authors' initials). By dividing data sequences into sliding windows, FRM generates too many points to be stored individually in an index. Thus, it constructs *minimum bounding rectangles* (*MBRs*) that contain hundreds or thousands of points, using a heuristic method, and stores those MBRs into the index. However, storing MBRs only causes false alarms by not allowing point-to-point comparison (which is called *point-filtering effect* [10]) in the index level for checking distances. *False alarms* are the candidates that are actually not in ϵ-match with the query sequence and are the major cause of performance degradation [1].

A new subsequence matching method, *Dual Match* [10] has been proposed to solve this problem of FRM. Dual Match is a dual approach for FRM in constructing windows: it divides data sequences into disjoint windows and a query sequence into sliding windows. By dividing data sequences into disjoint windows rather than sliding windows, Dual Match reduces the number of points to store drastically—to $1/\omega$ of that for FRM. Thus, Dual Match is able to store individual points themselves rather than MBRs in the index with much smaller storage overhead. By storing individual points directly, Dual Match exploits point-filtering effect, and accordingly, reduces false alarms and improves performance significantly. Nevertheless, Dual Match has the problem of having a smaller allowable window size—approximately half that of FRM—given the minimum query length [10]. A smaller window increases false alarms. This effect is called *window size effect* [10] and will be explained in more detail in Section 2. Due to window size effect, performance is slightly degraded in Dual Match compared with FRM for high selectivities[1] (i.e., higher than 10^{-2}).

In this paper, we generalize the method of constructing windows. By this generalization we can explain both FRM and DualMatch as special cases of a common framework. Based on the generalization, we propose a new subsequence matching method, *General Match*. General Match has advantages of both FRM and Dual Match: it can use large windows like FRM and, at the same time, can exploit point-filtering effect like Dual Match. We first define *J-sliding windows* and *J-disjoint windows*, which are generalization of sliding windows and disjoint windows. Here, J is the *sliding factor* representing the amount of shifting among consecutive J-sliding windows. General Match divides data sequences into J-sliding windows and the query sequence into J-disjoint windows. We then formally prove that General Match is correct, i.e., it incurs no false dismissal. *False dismissals* are the subsequences that are in ϵ-match with the query sequence but that are discarded as non-similar subsequences. Many subsequence matching methods can be devised by varying the value of J. Thus, we also propose a method of estimating the optimal value of the sliding factor J that minimizes the number of page accesses. Finally, we empirically show superiority of General Match over both Dual Match and FRM.

Table 1: Summary of notation.

Symbols	Definitions
$Len(S)$	length of sequence S
$Total\_Len$	sum of lengths of all data sequences
$S[k]$	the k-th entry of sequence S ($1 \le k \le Len(S)$)
$S[i:j]$	a subsequence of S, including entries from the i-th one to the j-th (if $i > j$, then it means a null sequence of length 0)
$S[i:k]S[k+1:j]$	$S[i:j]$ divided into two subsequences $S[i:k]$ and $S[k+1:j]$
s_i	the i-th disjoint window of sequence S ($= S[(i-1)*\omega+1 : i*\omega], i \ge 1$)

The rest of this paper is organized as follows. Section 2 describes existing work related to subsequence matching. Section 3 presents the concept of generalization and proposes General Match. Section 4 presents the results of performance evaluation. Section 5 summarizes and concludes the paper.

2. RELATED WORK

We first summarize in Table 1 the notation to be used throughout the paper. The symbols in Table 1 are self explanatory and do not need further elaboration. We then review related work for whole matching and review FRM and Dual Match as the representative research results for subsequence matching.

Whole Matching

A solution for similar sequence matching has first been introduced by Agrawal et al. [1]. The outline of the method is as follows. First, each data sequence of length n is transformed into the frequency domain by using Discrete Fourier Transform (DFT), and the first f ($\le n$) features are extracted. They are regarded as an f-dimensional point, and this point is indexed using the R*-tree [3]. Only a small number of features are extracted because storing high-dimensional sequences in the R*-tree is difficult due to the high dimensionality problem [4, 15]. Here, the function used for dimensionality reduction, such as the one extracting f features after DFT, is called the *feature extraction function* [8]. Next, a query sequence is similarly transformed to an f-dimensional point, and a range query is constructed using the point and the given tolerance ϵ. Then, the R*-tree is searched to evaluate the query, and a candidate set is constructed consisting of the feature points that are in ϵ-match with the query sequence. This method guarantees there be no false dismissal, but may cause false alarms because it uses only f features instead of n. Thus, for each candidate sequence obtained, the actual data sequence is accessed from the disk, and the distance from the query sequence is computed. The candidate is discarded if it is a false alarm. This last step, which eliminates false alarms, is called the *post-processing step* [1].

[1] Selectivity = $\frac{\text{the number of subsequences that are in } \epsilon\text{-match with the query sequence}}{\text{the number of all possible subsequences in the database}}$

Subsequence Matching – FRM

Faloutsos et al. [8] have proposed the subsequence matching method (called *FRM* for convenience) as a generalization of whole matching. FRM consists of index building and subsequence matching algorithms.

In the index building algorithm, FRM constructs an R*-tree using data sequences. FRM divides data sequences into sliding windows of size ω and transforms each window to an f-dimensional point. FRM generates almost *Total_Len* f-dimensional points corresponding to sliding windows for data sequences, and thus, needs f times more storage space than is required by original data sequences. Moreover, the search performance may become even poorer than that of sequential scanning due to the excessive height of the R*-tree [8]. To solve this problem, FRM does not store individual points directly into the R*-tree, but stores only MBRs that contain hundreds or thousands of such points.

In the subsequence matching algorithm, FRM divides the query sequence Q into $p\,(= \lfloor Len(Q)/\omega \rfloor)$ disjoint windows of size ω. The validity of FRM is based on the following two Lemmas:

Lemma 1. [8]: *When two sequences S and Q of the same length are divided into p disjoint windows s_i and q_i $(1 \le i \le p)$ respectively, if S and Q are in ϵ-match, then at least one of the pairs (s_i, q_i) are in $\epsilon/\sqrt{p}$-match. That is, the following equation holds:*

$$D(S,Q) \le \epsilon \quad \Longrightarrow \quad \bigvee_{i=1}^{p} D(s_i, q_i) \le \epsilon/\sqrt{p} \qquad (1)$$

Lemma 2. [8]: *If two sequences S and Q of the same length are in ϵ-match, then any pair of subsequences $(S[i:j], Q[i:j])$ are also in ϵ-match. That is, the following equation holds:*

$$D(S,Q) \le \epsilon \quad \Longrightarrow \quad D(S[i:j], Q[i:j]) \le \epsilon \qquad (2)$$

Using Lemmas 1 and 2, we derive Lemma 3 for the query sequence Q and a subsequence $S[i:j]$ of data sequence S.

Lemma 3. [8]: *Suppose the data sequence S is divided into sliding windows of size ω, and the query sequence Q into disjoint windows of the same size. If the subsequence $S[i:j]$ of length $Len(Q)$ is in ϵ-match with Q, then at least one disjoint window q_k of Q is in $\epsilon/\sqrt{p}$-match with the sliding window $S[i+(k-1)*\omega : i+k*\omega-1]$ included in $S[i:j]$. Here, p is $\lfloor Len(Q)/\omega \rfloor$.*

Lemma 3 guarantees that the candidate set consisting of the subsequences $S[i:j]$ such that q_k and $S[i+(k-1)*\omega : i+k*\omega-1]$ are in $\epsilon/\sqrt{p}$-match do not contain any false dismissal.

For subsequence matching, FRM divides the query sequence into disjoint windows, transforms each window to an f-dimensional point, makes a range query using the point and the tolerance ϵ, and constructs a candidate set by searching the R*-tree. Finally, it performs the post-processing step to eliminate false alarms.

Subsequence Matching – Dual Match

Dual Match [10], recently proposed as a dual approach of FRM, also consists of index building and subsequence matching algorithms. Like in FRM, the index building algorithm constructs a multidimensional index using data sequences. However, it stores individual points directly in the index while FRM stores only MBRs. It generates approximately *Total_Len*/ω points by dividing data sequences into disjoint windows, and thus, the storage overhead for the index is about f/ω of that for the original data sequences. This is only approximately $1/\omega$ of the storage that FRM would take if it stored (approximately *Total_Len*) individual points directly in the index. Thus, even if Dual Match stores individual points directly in the index, it does not suffer from excessive storage overhead and performance degradation that are encountered in FRM.

In the subsequence matching algorithm, Dual Match relies on the following Lemma 4 to guarantee there be no false dismissals.

Lemma 4. [10]: *Suppose the data sequence S is divided into disjoint windows of size ω, and the query sequence Q into sliding windows of the same size. If the subsequence $S[i:j]$ of length $Len(Q)$ is in ϵ-match with Q, then at least one disjoint window $S[i+k : i+k+\omega-1]$ included in $S[i:j]$ is in $\epsilon/\sqrt{p'}$-match with the sliding window $Q[k : k+\omega-1]$ of Q. Here, p' is $\lfloor (Len(Q)+1)/\omega \rfloor - 1$.*

Lemma 4 guarantees that the candidate set consisting of the subsequences $S[i:j]$ such that $S[i+k : i+k+\omega-1]$ and $Q[k : k+\omega-1]$ are in $\epsilon/\sqrt{p'}$-match do not contain any false dismissal.

For subsequence matching, Dual Match divides the query sequence into sliding windows, transforms them into f-dimensional points, and constructs a query MBR containing multiple points to reduce the number of range queries. Here, it maintains the points themselves together with the MBR. This is possible since a query sequence contains only a small number of sliding windows, say, a few hundreds. Next, it makes a range query using the MBR and $\epsilon/\sqrt{p'}$, searches the R*-tree using the range query. Then, it constructs a candidate set by comparing each point in the query MBR and each point in the search result and by discarding the false alarms caused by using the MBR. Finally, it performs the post-processing step to eliminate false alarms.

According to Lemmas 1 and 2, there is the tendency that smaller windows increase false alarms. This effect is called *window size effect* [10]. For example, let the window size of the method A be twice as large as that of the method B. Then, by Lemmas 1 or 2, a candidate subsequence in the method A must also be a candidate in the method B. However, the inverse does not hold. Thus, to reduce false alarms, we need to use as large windows as possible. As we mentioned in Section 1, however, Dual Match has the problem of having a smaller allowable window size—approximately half that of FRM—given the minimum query length [10].

3. SUBSEQUENCE MATCHING BASED ON GENERALIZED WINDOWS

In this section we generalize the method of constructing windows. Based on the generalization, we then propose a new subsequence matching method, General Match. Section 3.1 presents the concept of General Match. Sections 3.2 describes the index building algorithm; Section 3.3 the subsequence matching algorithm. Section 3.4 derives the maximum allowable window size and the number of points stored in the index discussing the relationship between them. Finally, Section 3.5 proposes a method of estimating the optimal value of the sliding factor J.

3.1 The Concept

J-Sliding Windows

General Match divides data sequences into J-sliding windows defined in the following Definition 1.

Definition 1: A *J-sliding window* $(1 \le J \le \omega)$ s_i^J of size ω of the sequence S is defined as the subsequence of length ω starting from $S[(i-1)*J+1]$ $(1 \le i \le \frac{Len(S)-\omega}{J}+1)$. □

Example 1: Figure 1 shows an example of dividing a sequence S into 4-sliding windows (i.e., $J=4$) of length $\omega = 16$. Intuitively speaking, we construct windows by shifting a subsequence of length 16 by 4 entries, and thus, the starting entries of the 4-sliding windows are $S[1], S[5], S[9], \ldots$, respectively. □

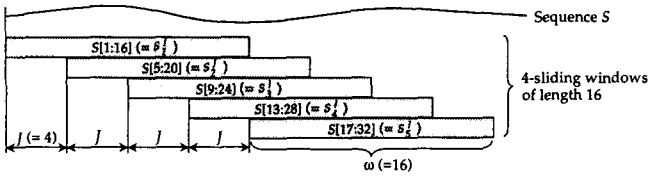


Figure 1: An example of dividing a sequence S into 4-sliding windows of size $\omega = 16$.

Lemma 5.: *If the sequence S is divided into J-sliding windows, the first J-sliding window included in the subsequence $S[i:j]$ is the $(\lceil \frac{i-1}{J} \rceil + 1)$-th J-sliding window of S.*

PROOF: See Appendix A. □

J can be any value in between 1 and ω. For convenience, in General Match we limit the value of J to be a divisor of ω to be able to use Lemma 1. Let s_a^J (starting offset $= (a-1)*J+1$) be a J-sliding window of the sequence S when we divide S into J-sliding windows. Then, the window resulting from moving s_a^J by ω entries must also be a J-sliding window of S, which is the $(a+\frac{\omega}{J})$-th J-sliding window $s_{a+\frac{\omega}{J}}^J$ (starting offset $= (a-1)*J+1+\omega$). From now on, we denote $\frac{\omega}{J}$ as k.

J-Disjoint Windows

General Match divides the query sequence into J-disjoint windows defined in the following Definition 2.

Definition 2: A *J-disjoint window* $(1 \le J \le \omega)$ $q_{(i,j)}^J$ of size ω of the sequence Q is defined as the subsequence of length ω starting from $Q[i+(j-1)*\omega]$ $(1 \le i \le J, 1 \le j \le \frac{Len(Q)-i+1}{\omega})$ in Q. □

Example 2: Figure 2 shows an example of dividing a sequence Q into 4-disjoint windows (i.e., $J=4$) of length $\omega = 16$. Intuitively speaking, we construct windows $Q[i : i+\omega-1], Q[i+\omega : i+2\omega-1], \ldots$ by dividing $Q[i : Len(Q)]$ into disjoint windows for every i $(1 \le i \le 4)$. □

The intuition behind dividing $Q[c : Len(Q)]$ into disjoint windows for every c in $1 \sim J$ when we construct J-sliding windows for a query sequence Q is as follows. In Figure 3, suppose that a data sequence S is divided into J-sliding windows, where the first J-sliding window included in a subsequence $S[i:j]$ is s_a^J, and the difference between the starting offset of $S[i:j]$ and that of s_a^J is b. Then, the value of b varies from 0 to $J-1$ according to the starting offset of $S[i:j]$. Here, if we are to compare Q with $S[i:j]$, we must use a window $Q[b+1 : b+\omega]$ $(= q_{(b+1,1)}^J)$ to compare it with s_a^J, and $Q[b+1+\omega : b+2\omega]$ $(= q_{(b+1,2)}^J)$ to compare it with the J-sliding window s_{a+k}^J constructed by moving s_a^J by ω entries. Thus, we need disjoint windows of $Q[c : Len(Q)]$ for every c in $1 \sim J$.

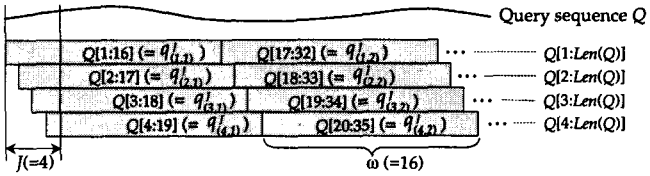


Figure 2: An example of dividing a sequence Q into 4-disjoint windows of size $\omega = 16$.

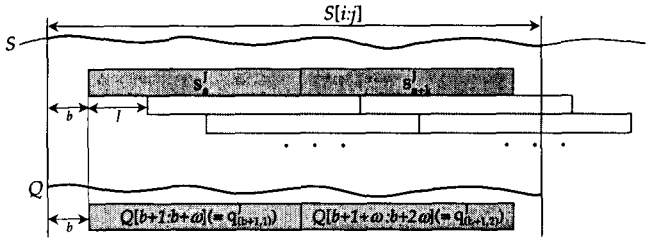


Figure 3: An example of constructing J-disjoint windows for a query sequence according to the starting offset of a subsequence.

FRM and Dual Match are typical examples of using J-sliding windows and J-disjoint windows. To divide data sequences, FRM uses sliding windows; Dual Match disjoint windows. In this case, we can regard sliding windows of FRM as 1-sliding windows and disjoint windows of Dual Match as ω-sliding windows. To divide the query sequence, FRM uses disjoint windows; Dual Match sliding windows. Similarly, we can regard disjoint windows of FRM as 1-disjoint windows and sliding windows of Dual Match as ω-disjoint windows. As a result, FRM uses the method of dividing data sequences into 1-sliding windows and the query sequence into 1-disjoint windows; Dual Match uses the method of dividing data sequences into ω-sliding windows and the query sequence into ω-disjoint windows.

Correctness of General Match

To explain correctness of General Match, we need the notion of included windows for a subsequence. Suppose that a sequence S is divided into J-sliding windows and that s_a^J is the first J-sliding window included in a subsequence $S[i:j]$. Then, we define the *included windows for S[i:j]* as those J-sliding windows s_{a+n*k}^J $(n \ge 0)$ included in $S[i:j]$. Here, we note that s_{a+n*k}^J is equivalent to the window constructed by moving s_a^J by $n*\omega$ entries.

Example 3: Figure 4 shows an example of included windows for a subsequence $S[i:j]$. In this figure, a data sequence S is divided into J-sliding windows, and the first J-sliding window included in $S[i:j]$ is s_a^J. Thus, the included windows for $S[i:j]$ are s_a^J, s_{a+k}^J, and s_{a+2k}^J. Here, s_{a+k}^J is equivalent to the window constructed by moving s_a^J by ω entries, and s_{a+2k}^J to the window constructed by moving s_a^J by 2ω entries □

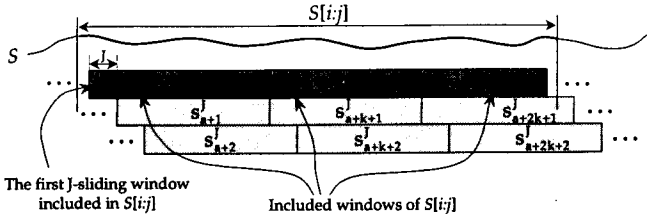


Figure 4: An example of included windows for a subsequence $S[i:j]$.

We now derive Theorem 1, which states the correctness of General Match.

Theorem 1.: *Suppose the data sequence S is divided into J-sliding windows of size ω, and the query sequence Q into J-disjoint windows of the same size. If the subsequence $S[i:j]$ of length $Len(Q)$ is in ϵ-match with Q, then at least one included window s^J_{a+n*k} $(0 \le n \le \rho - 1)$ for $S[i:j]$ is in $\epsilon/\sqrt{\rho}$-match with the J-disjoint window $q^J_{(b+1,n+1)}$ of Q. Here, ρ is* the number of included windows for $S[i:j]$. *That is, Eq. (3) holds:*

$$D(S[i:j], Q) \le \epsilon \Rightarrow \bigvee_{n=0}^{\rho-1} D(s^J_{a+n*k}, q^J_{(b+1,n+1)}) \le \epsilon/\sqrt{\rho}, \quad (3)$$

where a is $\lceil \frac{i-1}{J} \rceil + 1$, b is $(a-1)*J - i + 1$, and ρ is $\lfloor \frac{Len(Q)-b}{\omega} \rfloor$.

PROOF: See Appendix B. □

In Eq. (3) of Theorem 1, a means that the first J-sliding window of $S[i:j]$ is s^J_a, and b means that the starting offset of the J-disjoint window $q^J_{(b+1,1)}$ of Q that will be compared with s^J_a is $b+1$, which is the difference between the starting offset of $S[i:j]$ and that of s^J_a. Theorem 1 guarantees that the candidate set consisting of the subsequences $S[i:j]$ such that s^J_{a+n*k} and $q^J_{(b+1,n+1)}$ are in $\epsilon/\sqrt{\rho}$-match (i.e., satisfying the necessary condition of Eq. (3)) contain no false dismissal.[2]

3.2 Index Building Algorithm

Figure 5 shows the index building algorithm Build_Index. The input to the algorithm is a database containing data sequences; the output an f-dimensional index, which will be used in subsequence matching. In Step 2.1 of the algorithm, we divide each data sequence S, whose identifier is *S-id*, into J-sliding windows. In Step 2.2.1, we transform the c-th J-sliding window s^J_c of S to an f-dimensional point *f-point*. In Step 2.2.2, we construct a record consisting of the transformed point *f-point*, the data sequence identifier *S-id*, and the window index c of the J-sliding window. Here, the identifier will be used to find the actual data sequence that contains the candidate subsequence, and the window index to find the offset of the subsequence in the sequence.

[2]We can think of a dual approach of General Match, which divides data sequences into J-disjoint windows and a query sequence into J-sliding windows. However, analysis and experiments indicate that the proposed General Match is much superior in performance to the dual approach. The reason is that, if two methods store the same number of points in the index, the maximum window size of General Match is larger than that of the dual approach, and thus, General Match better exploits the window size effect. Thus, we omit the detailed discussion about the dual approach.

Algorithm Build_Index

Input: Database *db* that contains data sequences

Output: f-dimensional *index* that will be used for subsequence matching

Algorithm:

1. Initialize the *index*.
2. For each data sequence S with identifier *S-id* in *db* DO
   - 2.1 Divide S into J-sliding windows.
   - 2.2 For each J-sliding window s^J_c DO
     - 2.2.1 Transform s^J_c to an f-dimensional point *f-point*.
     - 2.2.2 Construct a record <*f-point*, *S-id*, *c*>.
     - 2.2.3 Insert the record, whose key is *f-point*, into the index.

Figure 5: The index building algorithm Build_Index.

We subsequently insert the record into the index using the transformed point as the key in Step 2.2.3.

To exploit the point-filtering effect, the algorithm Build_Index stores the individual points directly in the index like Dual Match. For a small value of J, it is difficult to use this algorithm since the number of points stored in the index becomes too large. FRM is an extreme example of this case. In contrast, for a large value of J, this algorithm works well since the number of points stored in the index becomes small. Dual Match is an extreme example of this case. In Section 3.4, we will investigate the number of points to be stored in the index in more detail. From now on, we distinguish two different variations of FRM: 1) the one that stores individual points directly (we call it *FRM-POINT*) and 2) the original method that stores only MBRs (we call it *FRM-MBR*).

3.3 Subsequence Matching Algorithm

Figure 6 shows the subsequence matching algorithm General_Match. The inputs to the algorithm are the time-series database, index, query sequence Q, and tolerance ϵ. The output is the set of sequences containing subsequences that are in ϵ-match with Q and offsets of these subsequences. The algorithm General_Match consists of three steps: initialization, index searching, and post-processing.

In the initialization step, we construct MBRs to be used for searching the index. In Step 1.1, we divide the query sequence into J-disjoint windows and transform each window to an f-dimensional point. In Step 1.2, we construct MBRs containing multiple points to reduce the number of range queries. Here, we maintain the points themselves together with the MBR.

We may use various techniques for constructing MBRs. Examples are 1) using the heuristics used in FRM for constructing MBRs for data sequences [8], 2) using a fixed number of points in an MBR, and 3) using only one MBR containing all the points. The detailed discussion, however, is not a focus of the paper and is left as a further study. In general, if the query sequence is long, it is more effective to limit the sizes of the MBRs by using multiple MBRs. To simplify the problem, however, we use only one MBR.

In the index searching step, we construct the candidate set by searching the index. In Step 2.1, we make a range query using each MBR and $\epsilon/\sqrt{\rho}$, and then, retrieve the qualifying points by searching the index. In Step 2.2, we construct the candidate set by filtering at the index level. That is, we compute the f-dimensional distance between each point in the MBR and each point in the search result. We then

Algorithm General_Match

Input: (1) Database *db* that contains data sequences
(2) *f*-dimensional *index* that has been created by Algorithm Build_Index
(3) Query sequence Q and tolerance ε

Output: Data sequences containing subsequences that are in ε-match with Q and offsets of these subsequences

Algorithm:

1. Initialization
   1.1 Divide Q into J-disjoint windows and transform each window to an f-dimensional point.
   1.2 Construct MBRs using the transformed points.
2. Index searching: for each MBR DO
   2.1 Construct a range query using the MBR and the tolerance $\varepsilon/\sqrt{p}$; search the index using the range query.
   2.2 Construct the candidate set by filtering at the index level (compute the distance between each point in the MBR and each point in the search result; include in the candidate set only those records having the points that are in $\varepsilon/\sqrt{p}$ -match, together with the window index (a,b) of the matching J-disjoint window $q^J_{(a,b)}$).
3. Post-processing: for each record <*f-point, S-id, c*> in the candidate set DO
   3.1 Read from *db* the candidate subsequence *sub-S* of the data sequence *S*. This is done using *S-id*. The offset of *sub-S* in *S* is calculated as (c-1)**J* - (b-1)*ω - a + 2. Here, (a,b) is the window index of the *J*-disjoint window that has been stored with this record in Step 2.2.
   3.2 If *sub-S* and Q are in ε-match, then output *S-id* and the offset of *sub-S*.

Figure 6: The subsequence matching algorithm General_Match.

include in the candidate set only those records having the points that are in $\epsilon/\sqrt{p}$-match, together with the window index (a,b) of the matching J-disjoint window $q^J_{(a,b)}$. We use the filtering at index level to exploit point-filtering effect while obtaining the same candidate set as in the case of using individual points themselves instead of using MBRs [10].

In the post-processing step, we select only similar subsequences by discarding false alarms from the candidate set. In Step 3.1, for each record <*f-point, S-id, c*> in the candidate set, we read the candidate subsequence *sub-S* from the database. This is done using *S-id* and the starting offset of *sub-S*. If the J-sliding window is the c-th one, then we calculate the starting offset of *sub-S* as $(c-1)*J-(b-1)*\omega-a+2$. Here, (a,b) is the window index of the J-disjoint window that has been stored with this record in Step 2.2. In Step 3.2, we remove false alarms keeping only those subsequences that are in ϵ-match with the query sequence.

3.4 Maximum Size of Window and Number of Points Stored in Index

Given the minimum size of the query sequence, there is a maximum window size that can be used in General Match as in Lemma 6. This is because, to make General Match work correctly, the number of included windows for a subsequence must be at least one. This maximum window size determines the number of points that are to be stored in the index as in Lemma 7.

Lemma 6.: *If the minimum length of the query sequence is Min(Q), the maximum window size allowable is* $\left\lfloor \frac{Min(Q)-J+1}{J} \right\rfloor * J$.

PROOF: See Appendix C. □

Lemma 7.: *If the data sequence S is divided into J-sliding windows of size ω, the number of points stored in the index is* $\left\lfloor \frac{Len(S)-\omega}{J} \right\rfloor + 1$.

PROOF: See Appendix D. □

Lemmas 6 and 7 indicate that the maximum window size and the number of points stored in the index vary according to the value of J. Since the number of disk I/Os occurring in subsequence matching is significantly affected by the window size and the number of points stored, we need to find an optimal value of J to minimize the disk I/O's. Table 2 shows the maximum window sizes and the numbers of points stored in the index for various values of J when *Min(Q)* is 512 and *Len(S)* is 1000000. The value of J is in $1 \sim \lfloor (Min(Q)+1)/2 \rfloor$ (=256), the maximum window size of Dual Match [10], and J is in $1 \sim \omega$. As shown in the table, for a larger value of J, the maximum window size and the number of points become smaller. For a larger value of J, disk I/Os for accessing data sequences (i.e., the number of data page accesses) increase since a smaller window increases false alarms. In contrast, for a larger value of J, disk I/Os for accessing the index (i.e., the number of index page accesses) become smaller because the number of points becomes smaller. In Section 3.5, we propose a method of estimating the optimal value of J that minimizes disk I/Os.

Table 2: The maximum size of the window and the number of points stored in the index when *Min(Q)* is 512 and *Len(S)* is 1000000.

J	ω	Number of points	Comments
1	512	999489	FRM-POINT (J=1)
2	510	499746	-
3	510	333164	-
⋮	⋮	⋮	-
128	384	7810	-
171	342	5375	-
256	256	3906	Dual Match (J=ω)

3.5 Estimating the Optimal Value of J

The number of page accesses (= data page accesses + index page accesses) varies according to the query sequence Q and tolerance ϵ that are given by the user. Therefore, we assume a usage pattern [16] consisting of (Q,ϵ) pairs are given in advance and obtain the optimal value of J for those (Q,ϵ) pairs given. To do this, we first obtain the average number of page accesses for each value of J, and then, determine the optimal value of J minimizing the average number of page accesses. Table 3 summarizes the notation. To obtain $n_{page}(Q,j)$, we first estimate both $n_{ipage}(Q,\epsilon,j)$ and $n_{dpage}(Q,\epsilon,j)$ for each value j of J.

$n_{ipage}(Q,\epsilon,j)$: the number of index page accesses

We estimate $n_{ipage}(Q,\epsilon,j)$ as follows.[3] We evaluate one range query for each (Q,ϵ) finding $n_{pts}(Q,\epsilon,j)$ points out

[3] Faloutsos and Kamel [7] have estimated the number of index page accesses for the R*-tree by using the concept of fractal dimension. In this paper, however, we do not use their estimation method. The reasons are 1) it is not clear that the set of f-dimensional points stored in the index for subsequence matching is a fractal, and 2) they assume

Table 3: Notation for estimating the average number of page accesses.

Symbols	Definitions
$\mathbb{Q}$	Set of the (Q, ϵ) pairs given
$n_{page}(\mathbb{Q}, j)$	Average number of page accesses for all (Q, ϵ) pairs in $\mathbb{Q}$ when $J = j$
$n_{ipage}(Q, \epsilon, j)$	Number of index page accesses for (Q, ϵ) when $J = j$
$n_{dpage}(Q, \epsilon, j)$	Number of data page accesses for (Q, ϵ) when $J = j$
$size_i(j)$	Size of the index to be built in pages when $J = j$
$size_d$	Size of the time-series database in pages
f_{int}	Average fan-out of internal pages of the index
f_{leaf}	Average fan-out of leaf pages of the index
$n_{ipts}(j)$	Number of points to be stored in the index when $J = j$
$n_{pts}(Q, \epsilon, j)$	Number of points retrieved by evaluating the range query (Q, ϵ) when $J = j$
$n_{subs}(Len(Q))$	Number of all possible subsequences of $Len(Q)$ in the database
$n_{cand}(Q, \epsilon, j)$	Number of candidate subsequences retrieved for (Q, ϵ) when $J = j$
$H(j)$	Height of the index to be built when $J = j$ $(H(j) = 1 + \lceil \log_{f_{int}} \frac{n_{ipts}(j)}{f_{leaf}} \rceil)$

of a total of $n_{ipts}(j)$ points as the result. Thus, evaluation of the range query (Q, ϵ) requires approximately $\frac{n_{pts}(Q,\epsilon,j)}{n_{ipts}(j)}$ of total index pages obtaining Eq. (4).

$$n_{ipage}(Q, \epsilon, j) \approx size_i(j) \times \frac{n_{pts}(Q, \epsilon, j)}{n_{ipts}(j)} \quad (4)$$

We now estimate $size_i(j)$ as in Eq. (5) using $n_{ipts}(j)$, f_{int}, and f_{leaf}. In Eq. (5), $\lceil \frac{n_{ipts}(j)}{f_{leaf}} \rceil$ represents the estimated number of leaf pages and $\lceil \frac{n_{ipts}(j)}{f_{leaf} \times f_{int}^h} \rceil$ $(1 \le h \le H(j) - 1)$ that of internal pages.

$$\begin{aligned} size_i(j) &\approx \left\lceil \frac{n_{ipts}(j)}{f_{leaf}} \right\rceil + \left\lceil \frac{n_{ipts}(j)}{f_{leaf} \times f_{int}} \right\rceil + \cdots \\ &\quad + \left\lceil \frac{n_{ipts}(j)}{f_{leaf} \times f_{int}^{H(j)-1}} \right\rceil \\ &= \sum_{h=0}^{H(j)-1} \left\lceil \frac{n_{ipts}(j)}{f_{leaf} \times f_{int}^h} \right\rceil \end{aligned} \quad (5)$$

By substituting $size_i(j)$ with Eq. (5), we obtain Eq. (6) from Eq. (4).

$$n_{ipage}(Q, \epsilon, j) \approx \left(\sum_{h=0}^{H(j)-1} \left\lceil \frac{n_{ipts}(j)}{f_{leaf} \times f_{int}^h} \right\rceil \right) \times \frac{n_{pts}(Q, \epsilon, j)}{n_{ipts}(j)} \quad (6)$$

$n_{dpage}(Q, \epsilon, j)$: the number of data page accesses

We estimate $n_{dpage}(Q, \epsilon, j)$ as follows. We need to access data pages to retrieve candidate subsequences from the database. In subsequence matching, adjacent subsequences are similar and tend to be stored in the same data page. Thus, many candidate subsequences are accessed together in a clustered fashion [10]. Hence, we can estimate $n_{dpage}(Q, \epsilon, j)$ as in Eq. (7).

$$n_{dpage}(Q, \epsilon, j) \approx size_d \times \frac{n_{cand}(Q, \epsilon, j)}{n_{subs}(Len(Q))} \quad (7)$$

$n_{page}(\mathbb{Q}, j)$: the average number of page accesses

We obtain $n_{page}(\mathbb{Q}, j)$ as in Eq. (8) by adding $n_{ipage}(Q, \epsilon, j)$ in Eq. (6) and $n_{dpage}(Q, \epsilon, j)$ in Eq. (7) and by averaging over all the queries.

$$\begin{aligned} n_{page}(\mathbb{Q}, j) &= \underset{for\ (Q,\epsilon) \in \mathbb{Q}}{average} \left[(n_{ipage}(Q, \epsilon, j) + n_{dpage}(Q, \epsilon, j)) \right] \\ &\approx \underset{for\ (Q,\epsilon) \in \mathbb{Q}}{average} \left[\left(\left(\sum_{h=0}^{H(j)-1} \left\lceil \frac{n_{ipts}(j)}{f_{leaf} \times f_{int}^h} \right\rceil \right) \times \frac{n_{pts}(Q, \epsilon, j)}{n_{ipts}(j)} + size_d \times \frac{n_{cand}(Q, \epsilon, j)}{n_{subs}(Len(Q))} \right) \right] \end{aligned} \quad (8)$$

In Eq. (8), we have two categories of parameters: 1) the first ones that do not require access to the database in obtaining their values for each (Q, ϵ) and j, and 2) the second ones that require access to the database for each (Q, ϵ) and j. The parameters in the first category are $size_d$, f_{int}, f_{leaf}, $n_{ipts}(j)$, and $n_{subs}(Len(Q))$. Those in the second category are $n_{pts}(Q, \epsilon, j)$ and $n_{cand}(Q, \epsilon, j)$. The values of $n_{pts}(Q, \epsilon, j)$ and $n_{cand}(Q, \epsilon, j)$ vary according to the values of various factors such as query sequences, tolerances, and sizes of the windows. Thus, we obtain the values of all $n_{pts}(Q, \epsilon, j)$'s and $n_{cand}(Q, \epsilon, j)$'s by one database scan. Having obtained all the values of the parameters, we calculate $n_{page}(\mathbb{Q}, j)$ for each j in $1 \sim \lfloor (Min(Q) + 1)/2 \rfloor$, and then, determine the optimal value of J minimizing $n_{page}(\mathbb{Q}, j)$.

4. PERFORMANCE EVALUATION

In this section, we present the results of performance evaluation comparing FRM-POINT, FRM-MBR, Dual Match, and General Match. We describe the experimental data and environment in Section 4.1 and present the results of the experiments in Section 4.2.

4.1 Experimental Data and Environment

We have performed experiments using three different data sets. A data set consists of a long data sequence and has the same effect as the one consisting of multiple data sequences [8]. The first data set, a real stock data set[4] used in FRM [8] and Dual Match [10], consists of 329112 entries. We call this data set *STOCK-DATA*. The second data set, used in Dual Match, contains pseudo periodic synthetic time-series data[5] consisting of one million entries. We call this data set *PERIODIC-1M* (M means one million entries). The last

square-like MBRs of the same size at each level, but real MBRs that contain actual f-dimensional points show various shapes of rectangles.

[4]This data set can be obtained from ftp://ftp.santafe.edu-/pub/Time-Series/data/.

[5]This data set is one of those that are currently under construction with support from the National Science Foundation and can be obtained from http://kdd.ics.uci.edu/databases/synthetic/synthetic.html.

data set, also used in FRM and Dual Match, contains random walk data consisting of one million entries: the first entry is set to 1.5, and subsequent entries are obtained by adding a random value in the range (-0.001,0.001) to the previous one. We call this data set *WALK-1M*. We also generate *WALK-10M* consisting of ten million entries by repeating WALK-1M ten times and *WALK-100M* by repeating WALK-1M 100 times.

We conduct all the experiments on a SUN Ultra 60 workstation with 512 Mbytes of main memory. To avoid the buffering effect of the UNIX file system and to guarantee actual disk I/Os, we use raw disks for data and index files. The page size for data and indexes is set to be 4096 bytes. As the multidimensional index, we use the R*-tree for all the methods. The storage utilization of the R*-tree for estimating the optimal value of J is set to be a standard value of 69% [17]. We use Discrete Fourier Transform [11] as the feature extraction function and use six features.[6]

We use the number of candidates, the number of page accesses, and the wall clock time as the performance measures. We generate query sequences from the data sequence by taking subsequences of length *Len(Q)* starting from random offsets as has been done in FRM [8] and Dual Match [10]. We use 512~1024 as the lengths of query sequences and generate ten different query sequences for each length. We perform experiments for two ranges of selectivities: *Low-Range* (10^{-6}~10^{-4}) and *High-Range* (10^{-3}~10^{-1}). We obtain the desired selectivity by controlling the tolerance ϵ and uniformly distribute query sequences over various selectivities in the given range. Since the optimal value of J for General Match varies according to the data set and the range of selectivity, we estimate it for each data set and range of selectivity. Accordingly, each experiment uses its own optimal value of J.

4.2 Results of the Experiments

STOCK-DATA

We first perform an experiment that evaluates the accuracy of the method for estimating the optimal value of J proposed in Section 4.6. To do this, we estimate the optimal value of J (we call it *the estimated* J) by using the proposed method and compare it with the true optimal value of J (we call it *the real* J) obtained by building an index for each value of J and executing the subsequence matching algorithm. As the usage pattern [16], we use 3% of query sequences that are randomly selected among those in Low-Range. Since we limit the value of J (or $k = \omega/J$) to be a divisor of ω, multiple values of J have the same value of k. For the same value of k, we consider the largest J (the largest w) because, as the value of J becomes larger, the number of points stored in the index becomes smaller, and the window size becomes larger for the same value of k. Figure 7 (a) shows the estimated and real numbers of average page accesses to do subsequence matching for each value of J in Low-Range. As shown in the figure, the estimated J is 85, and the real J is 57. Figure 7 (a) shows that the estimated and the real numbers differ slightly but are very similar in trends. Figure 7 (b) shows the numbers of page accesses in Low-Range when J is 85 (the estimated J), 57 (the real J), 1 (the value for FRM-POINT), and 256 (the value for Dual Match) for various selectivity ranges. As shown in the figure, the number of pages accesses is much less when using the estimated J or the real J compared with those when using the values for FRM-POINT or Dual Match. In addition, when we use the estimated J or the real J, there is only a little difference (less than 1.58% on the average) in the number of page accesses. This result indicates that estimation of the optimal value obtained by the proposed method is reasonable.

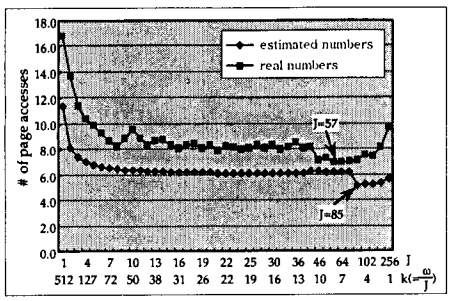
Figure 8 shows the results of the experiment in Low-Range for STOCK-DATA. Here, we use the estimated J of 85 (ω = 425). In the figure, we note that FRM-POINT, General Match, and Dual Match outperform FRM-MBR in all three measures. This difference is due to lack of the point-filtering effect in FRM-MBR. Having the largest window and exploiting the point-filtering effect, FRM-POINT shows the least number of candidates in Figure 8 (a). Due to the searching overhead for the index, however, it has higher values for the number of page accesses and for the wall clock time compared with General Match. General Match also shows a performance better than that of Dual Match since its window size is larger. In summary in Low-Range, General Match reduces the wall clock time averaged over the entire range of selectivities by 117% compared with Dual Match and 998% compared with FRM-MBR, reduces the number of candidates by 101% compared with Dual Match and 19200% compared with FRM-MBR, and reduces the number of page accesses by 47% compared with Dual Match and 395% compared with FRM-MBR.

Figure 9 shows the results of the experiment in High-Range for STOCK-DATA. Since the optimal value of J can vary according to the ranges of selectivities, we separately estimate the optimal value in High-Range. Here, we obtain the estimated J of 57 (ω = 456). (The real J is 64.) As shown in the figure, having the largest window, FRM-POINT has the least number of candidates; but, it significantly degrades the other two measures compared with the other three methods. This degradation of FRM-POINT is due to the searching overhead for the index. For very high selectivities ($\geq 10^{-2}$), Dual Match shows a minor degradation (less than 9.6%) in comparison with FRM-MBR since the window size effect becomes more eminent than the point-filtering effect in higher selectivities [10]. In contrast, General Match outperforms FRM-MBR even in very high selectivities since it not only exploits the point-filtering effect but also uses a relatively larger window (89% ($= \frac{456}{512}$) that of FRM). In summary in High-Range, General Match reduces the wall clock time averaged over the entire range of selectivities by 45% compared with Dual Match and 64% compared with FRM-MBR, reduces the number of candidates by 46% compared with Dual Match and 100% compared with FRM-MBR, and reduces the number of page accesses by 17% compared with Dual Match and 17% compared with FRM-MBR.

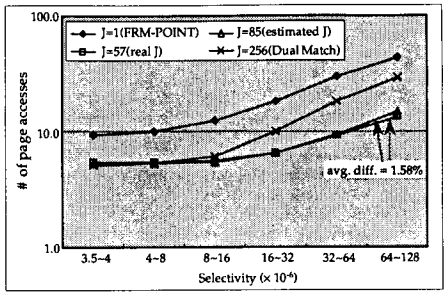
PERIODIC-1M

Figure 10 shows the results of the experiment in Low-Range for PERIODIC-1M. Here, we obtain the estimated J of 57 (ω = 456). (The real J is 85.) The results in Figure 10 show tendencies similar to those in Figure 8 for all three measures. However, the differences between General Match and FRM-MBR and between Dual Match and FRM-MBR are much larger than in Figure 8. PERIODIC-1M has the characteristic that the changes among adjacent

[6] We have used the real part of the fourth complex number instead of the imaginary part of the first one, which is 0.

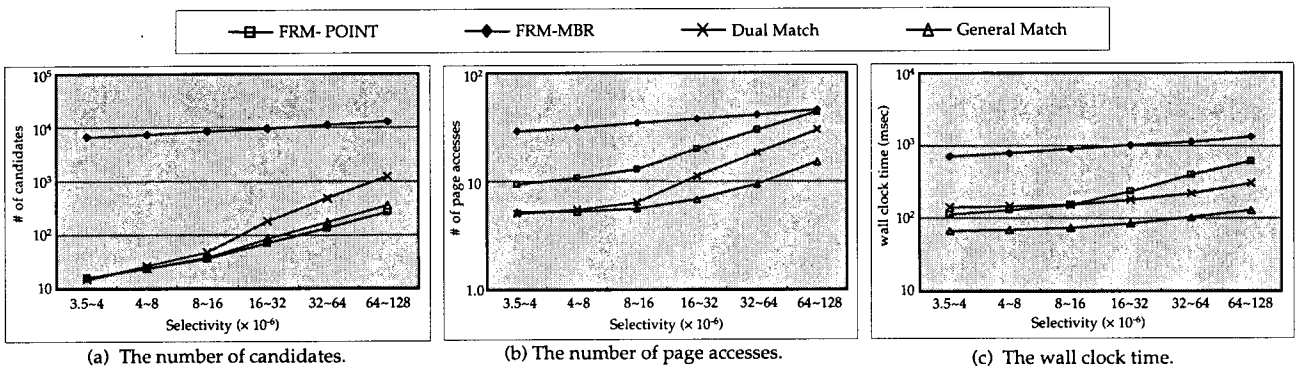


(a) Comparison of the estimated and the real numbers of average page accesses as J varies in Low-Range.



(b) Comparison of the numbers of page accesses by using 85 (estimated J), 57(real J), 1(FRM-POINT), and 256(Dual Match).

Figure 7: Comparison of the estimated J and the real J in Low-Range for STOCK-DATA.



(a) The number of candidates. (b) The number of page accesses. (c) The wall clock time.

Figure 8: Performance comparisons in Low-Range for STOCK-DATA when $J = 85$.

entries are relatively large. Therefore, adjacent windows in PERIODIC-1M tend to have distances among them larger than in STOCK-DATA. Thus, in FRM-MBR, which stores MBRs of multiple adjacent windows, many windows in the same MBR may be far apart from one another. Since these windows are included in the candidate set together, many false alarms are generated. In contrast, Genernal Match and Dual Match do not have this problem because they store individual points rather than MBRs. For this reason, General Match or Dual Match shows larger improvement in PERIODIC-1M than in STOCK-DATA [10]. For very low selectivities ($\leq 7.5 \times 10^{-6}$), the number of page accesses for General Match is larger than that for Dual Match. This is because we estimate the optimal value of J that minimizes the average number of page accesses in the whole Low-Range ($1.0 \times 10^{-6} \sim 1.0 \times 10^{-4}$) rather than minimizing for a specific selectivity.

In High-Range for PERIODIC-1M, General Match again outperforms Dual Match and FRM-MBR since it has advantages of both FRM-MBR and Dual Match. We omit the detailed results of this experiment because of space limitation of the paper.

WALK-1M/10M/100M

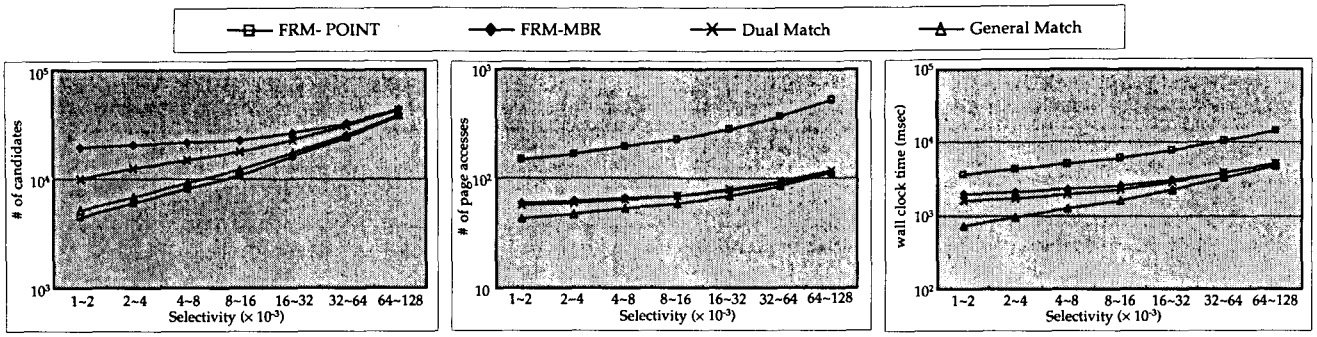
Figure 11 shows the results of the experiment in Low-Range for WALK-1M/WALK-10M/WALK-100M.[7] Here, we use the estimated J of 57 ($\omega = 456$). (The real J is 27.) In this experiment we obtain the value of each measure by averaging over all the selectivities in Low-Range for illustrative purposes. As shown in this figure, General Match improves performance over Dual Match as well as over FRM-MBR. Moreover, the performance differences among the three methods stay relatively constant regardless of database sizes. In summary, we conclude that General Match outperforms the other methods even for very large databases.

The results in High-Range show tendencies similar to those in Low-Range. We omit the detailed results of this experiment because of space limitation of the paper.

5. CONCLUSIONS

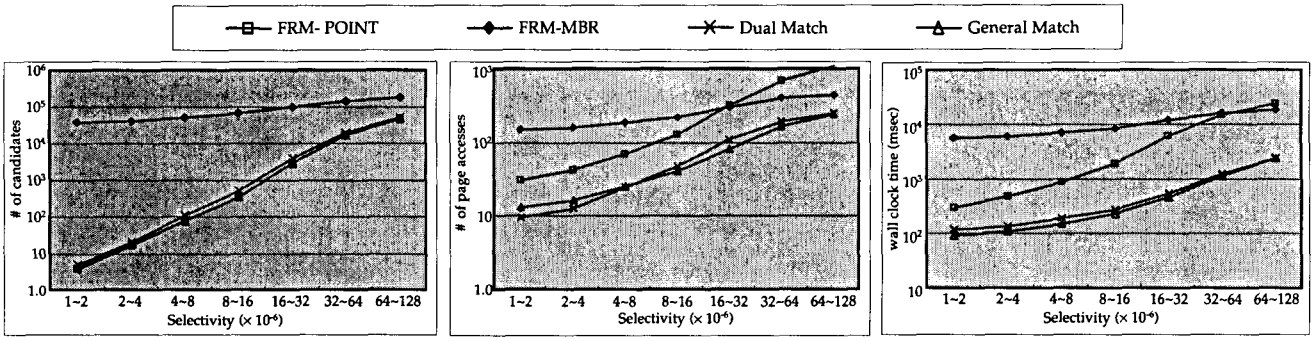
In this paper, we have generalized the method of constructing windows in subsequence matching. Based on the generalization, we have proposed a new subsequence matching method, General Match. From the point of view of this generalization, the previous subsequence matching methods, FRM [8] and Dual Match [10], can be considered special cases of a common framework. By using General Match, we can construct windows in such a way as to achieve optimal performance. General Match divides data sequences into generalized sliding windows (J-sliding windows) and the query sequence into generalized disjoint windows (J-disjoint windows). General Match outperforms both FRM and Dual-Match. This is because it has advantages of both FRM and Dual Match: it can use large windows like FRM and, at the same time, can exploit the point-filtering effect like Dual Match.

[7] We have not tested FRM-POINT since the index size becomes larger than 200Gbytes, which is too large a size to build. If it were tested, the curve would be between FRM-MBR and Dual Match in Low-Range, and much higher than FRM-MBR in High-Range.



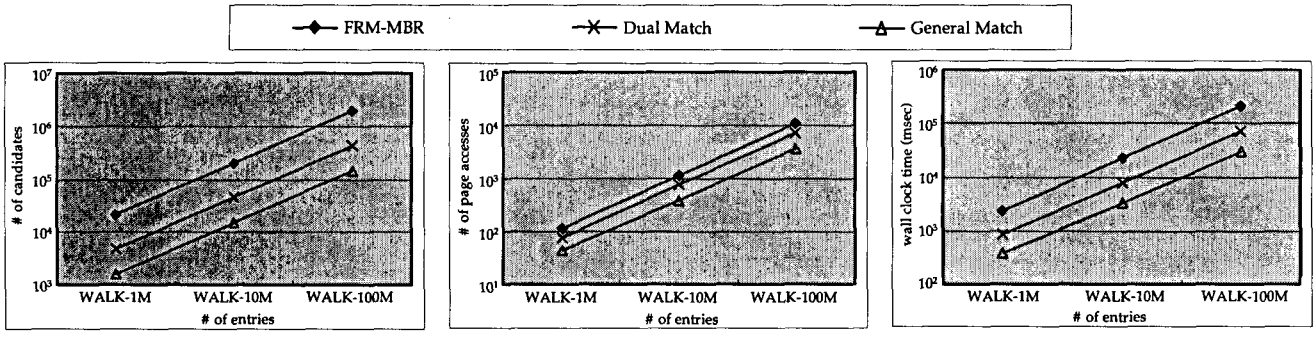
(a) The number of candidates. (b) The number of page accesses. (c) The wall clock time.

Figure 9: Performance comparisons in High-Range for STOCK-DATA when $J = 57$.



(a) The number of candidates. (b) The number of page accesses. (c) The wall clock time.

Figure 10: Performance comparisons in Low-Range for PERIODIC-1M when $J = 57$.



(a) The number of candidates. (b) The number of page accesses. (c) The wall clock time.

Figure 11: Performance comparisons in Low-Range for WALK-1M/10M/100M when $J = 57$.

We have formally proven the correctness of General Match in Theorem 1. That is, Theorem 1 guarantees that General Match perform subsequence matching without false dismissals. We have also proven that, given the minimum length of the query sequences, there is a maximum bound of the window size to guarantee correctness of General Match. We have derived the maximum window size in Lemma 6. Finally, we have proposed a method of estimating the optimal value of the sliding factor J that minimizes the number of page accesses.

We have performed extensive experiments for General-Match using various data sets and selectivity ranges. Experimental results show that, regardless of data sets, selectivity ranges, and database sizes, General Match significantly reduces the number of candidates, the number of page accesses, and the wall clock time compared with Dual Match as well as compared with FRM. Experimental results for real stock data show that, for low selectivities (10^{-6}~10^{-4}), General Match improves performance (the wall clock time) averaged over the entire range of selectivities by 117% over Dual Match and by 998% over FRM; for high selectivities (10^{-3}~10^{-1}), by 45% over Dual Match and by 64% over FRM.

Overall, these results indicate that General Match is a new subsequence matching method significantly more efficient than existing methods in supporting various database applications. These results also provide an excellent theoretical basis for understanding the underlying mechanisms in subsequence matching and for formally analyzing the performance.

6. ACKNOWLEDGEMENTS

This work was supported by the Korea Science and Engineering Foundation (KOSEF) through the Advanced Information Technology Research Center (AITrc).

7. REFERENCES

[1] R. Agrawal, C. Faloutsos, and A. Swami. Efficient similarity search in sequence databases. In *Proc. the 4th Int'l Conf. on Foundations of Data Organization and Algorithms*, pages 69–84, 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. the 21st Int'l Conf. on Very Large Data Bases*, pages 490–501, 1995.

[3] 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. Int'l Conf. on Management of Data*, pages 322–331, 1990.

[4] S. Berchtold, C. Bohm, and H.-P. Kriegel. The pyramid-technique: Towards breaking the curse of dimensionality. In *Proc. Int'l Conf. on Management of Data*, pages 142–153, 1998.

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

[6] K. W. Chu and M. H. Wong. Fast time-series searching with scaling and shifting. In *Proc. the 15th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 237–248, 1999.

[7] C. Faloutsos and I. Kamel. Beyond uniformity and independence: Analysis of r-trees using the concept of fractal dimension. In *Proc. the 13th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 4–13, 1994.

[8] C. Faloutsos, M. Ranganathan, and Y. Manolopoulos. Fast subsequence matching in time-series databases. In *Proc. Int'l Conf. on Management of Data*, pages 419–429, 1994.

[9] H. V. Jagadish, A. O. Mendelzon, and T. Milo. Similarity-based queries. In *Proc. the 14th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 36–45, 1995.

[10] Y.-S. Moon, K.-Y. Whang, and W.-K. Loh. Duality-based subsequence matching in time-series databases. In *Proc. the 17th Int'l Conf. on Data Engineering*, pages 263–272, 2001.

[11] A. V. Oppenheim and R. W. Schafer. *Digital Signal Processing.* Prentice-Hall, 1975.

[12] S. Park, W. W. Chu, J. Yoon, and C. Hsu. Efficient searches for similar subsequences of different lengths in sequence databases. In *Proc. the 16th Int'l Conf. on Data Engineering*, pages 23–32, 2000.

[13] D. Rafiei. On similarity-based queries for time series data. In *Proc. the 15th Int'l Conf. on Data Engineering*, pages 410–417, 1999.

[14] D. Rafiei and A. Mendelzon. Similarity-based queries for time series data. In *Proc. Int'l Conf. on Management of Data*, pages 13–25, 1997.

[15] R. Weber, H.-J. Schek, and S. Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *Proc. the 24th Int'l Conf. on Very Large Data Bases*, pages 194–205, 1998.

[16] K. Whang, G. Wiederhold, and D. Sagalowicz. Separability–an approach to physical database design. *IEEE Trans. on Computers*, c-33(3):209–222, 1984.

[17] A. C.-C. Yao. On random 2-3 trees. *Acta Informatica*, 9:159–170, 1978.

[18] B.-K. Yi, H. V. Jagadish, and C. Faloutsos. Efficient retrieval of similar time sequences under time warping. In *Proc. the 14th Int'l Conf. on Data Engineering*, pages 201–208, 1998.

Appendix A

PROOF OF LEMMA 5: Let the first J-sliding window included in the subsequence $S[i:j]$ be s_a^J, the a-th J-sliding window of S. Then, the following Eq. (9) holds because the first entry of s_a^J is $S[(a-1)*J+1]$.

$$(a-1)*J+1 \geq i \iff a \geq \frac{i-1}{J}+1 \tag{9}$$

In Eq. (9), since a is an integer, Eq. (10) holds.

$$a \geq \left\lceil \frac{i-1}{J} \right\rceil + 1 \tag{10}$$

Since s_a^J is the first J-sliding window included in $S[i:j]$, $a = \lceil \frac{i-1}{J} \rceil + 1$ must be satisfied. □

Appendix B

PROOF OF THEOREM 1: In Figure 12, suppose the subsequence $S[i:j]$ is in ϵ-match with the query sequence Q, and the first J-sliding window included in $S[i:j]$ is s_a^J (starting offset $= (a-1)*J+1$). Then, $S[i:j]$ can be represented as $S[i:(a-1)*J]\,S[(a-1)*J+1:j]$. Suppose $S[(a-1)*J+1:j]$ is divided into ρ disjoint windows. Then, $S[i:j]$ must include $s_a^J, s_{a+k}^J, \cdots, s_{a+(\rho-1)*k}^J$ where $k=\frac{\omega}{J}$. Note that we limit the value of J to be a divisor of ω. It also includes (possibly null) subsequences s_h (at the head) and s_t (at the tail). Thus, $S[i:j]$ can also be represented as $s_h\, s_a^J s_{a+k}^J \cdots s_{a+(\rho-1)*k}^J\, s_t$. Then, Q can be represented as $q_h\, q_{(b+1,1)}^J\, q_{(b+1,2)}^J \cdots q_{(b+1,\rho)}^J\, q_t$, where $Len(q_h)=Len(s_h)$ and $Len(q_t)=Len(s_t)$.

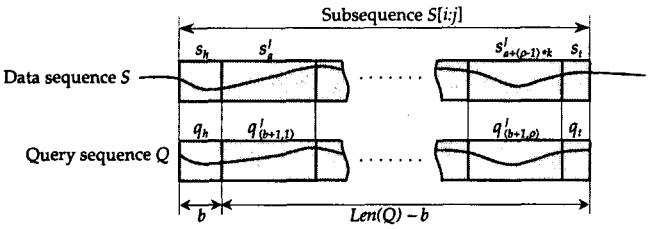


Figure 12: A subsequence $S[i:j]$ in ϵ-match with the query sequence Q.

Finally, we obtain Eq. (11) by using Lemmas 1 and 2.

$$
\begin{aligned}
& D(S[i:j],Q) \le \epsilon \\
& \quad \Longrightarrow D(s_a^J \cdots s_{a+(\rho-1)*k}^J, q_{(b+1,1)}^J \cdots q_{(b+1,\rho)}^J) \le \epsilon \\
& \quad \Longrightarrow \bigvee_{n=0}^{\rho-1} D(s_{a+n*k}^J, q_{(b+1,n+1)}^J) \le \epsilon/\sqrt{\rho} \qquad (11)
\end{aligned}
$$

Here, since s_a^J is the first window of $S[i:j]$, a is $\lceil \frac{i-1}{J} \rceil + 1$ by Lemma 5; b is $(a-1)*J-i-1$ since $b\,(=Len(s_h)) = Len(S[i:(a-1)*J])$; ρ is $\left\lfloor \frac{Len(Q)-b}{\omega} \right\rfloor$ since $Q[b+1:Len(Q)]$ is divided into $\left\lfloor \frac{Len(Q)-b}{\omega} \right\rfloor$ disjoint windows. □

Appendix C

PROOF OF LEMMA 6: By Theorem 1, a subsequence that is in ϵ-match with the query sequence must include at least one J-disjoint window. In other words, $\rho \ge 1$. Then, we obtain the relationship between the length of the query sequence and the size of the window as follows:

$$
\begin{aligned}
& \rho = \lfloor (Len(Q)-b)/\omega \rfloor \ge 1 \\
\Longleftrightarrow\ & (Len(Q)-b)/\omega \ge 1 \ \text{(holds since 1 is an integer)} \\
\Longleftrightarrow\ & Len(Q) \ge \omega + b
\end{aligned}
$$

Since $Min(Q)$ is the minimum length of the query sequence, $\omega \le Min(Q)-b$ must be satisfied. Since b can be as large as $J-1$, $\omega \le Min(Q)-J+1$ must also be satisfied. Finally, since the value of J is limited to be a divisor of ω, the maximum window size allowable is $\left\lfloor \frac{Min(Q)-J+1}{J} \right\rfloor * J$. □

Appendix D

PROOF OF LEMMA 7: Suppose the data sequence S is divided into n J-sliding windows. Then, the last J-sliding window of S is $s_n^J\,(=S[(n-1)*J+1:(n-1)*J+\omega])$. Thus, $(n-1)*J+\omega \le Len(S) \Leftrightarrow n \le \frac{Len(S)-\omega}{J}+1$. Next, $n \le \left\lfloor \frac{Len(S)-\omega}{J} \right\rfloor + 1$ is also satisfied since n is an integer. Here, since s_n^J is the last window of S, it must be that $n = \left\lfloor \frac{Len(S)-\omega}{J} \right\rfloor + 1$. □