Purdue University
Purdue e-Pubs

Computer Science Technical Reports | Department of Computer Science

2008

Incremental Mining for Frequent Patterns in Evolving Time Series Datatabases

Mohamed Y. Eltabakh

Mourad Ouzzani

Mohamed A. Khalil

Walid G. Aref
Purdue University, aref@cs.purdue.edu

Ahmed K. Elmagarmid
Purdue University, ake@cs.purdue.edu

Report Number:
08-020

Eltabakh, Mohamed Y.; Ouzzani, Mourad; Khalil, Mohamed A.; Aref, Walid G.; and Elmagarmid, Ahmed K., "Incremental Mining for Frequent Patterns in Evolving Time Series Datatabases" (2008). *Computer Science Technical Reports.* Paper 1707.
http://docs.lib.purdue.edu/cstech/1707

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries. Please contact epubs@purdue.edu for additional information.

Incremental Mining for Frequent Patterns in Evolving Time Series Datatabases

Mohamed Eltabakh
Mourad Ouzzani
Mohamed Khalil
Walid Aref
Ahmed Elmagarmid

CSD TR #08-020
July 2008

Incremental Mining for Frequent Patterns in Evolving Time Series Databases

Mohamed Y. Eltabakh Mourad Ouzzani Mohamed A. Khalil
Walid G. Aref Ahmed K. Elmagarmid

Department of Computer Science, Purdue University

{meltabak, mourad, aref, ake}@cs.purdue.edu

Abstract

Several emerging applications warrant mining and discovering hidden frequent patterns in time series databases, e.g., sensor networks, environment monitoring, and inventory stock monitoring. Time series databases are characterized by two features: (1) The continuous arrival of data and (2) the time dimension. These features raise new challenges for data mining such as the need for online processing and incremental evaluation of the mining results. In this paper, we address the problem of discovering frequent patterns in databases with multiple time series. We propose an incremental technique for discovering the complete set of frequent patterns, i.e., discovering the frequent patterns over the entire time series in contrast to a sliding window over a portion of the time series. The proposed approach updates the mining results with the arrival of every new data item by considering only the items and patterns that may be affected by the newly arrived item. Our approach has the ability to discover frequent patterns that contain gaps between patterns' items with a user-defined maximum gap size. The experimental evaluation illustrates that the proposed technique is efficient

and outperforms recent sequential pattern incremental mining techniques.

Keywords: Time series databases, frequent patterns, incremental mining.

1 Introduction

Several emerging applications warrant mining and discovering hidden patterns in time series databases, e.g., sensor networks, stock prices, environment monitoring, and inventory stock monitoring. A general form of time series databases is that the database consists of sequences of itemsets. For example, inventory stock databases consist of sequences of customers' transactions where each transaction is a set of items. However, a special form of time series databases consists of sequences of items instead of itemsets. There are several time series applications, e.g., sensor networks, stock prices, and environment monitoring that generate sequences of simple items, e.g., numeric values. Thus, it is crucial to provide for these applications algorithms that are more efficient than general sequence mining algorithms, e.g., [7, 13, 14, 19]. Our focus in this paper will be on mining time series databases that consist of sequences of items.

A time series database consists of one or more time series where each time series is a collection of items ordered based on their timestamps. Time series databases are characterized by two features: (1) The continuous arrival of data and (2) the time dimension. These features introduce several challenges to data mining techniques. For example, since items are arriving continuously to the database, it will be very expensive to perform multiple scans over the data to discover interesting patterns. Instead, mining techniques need to *incrementally* update the mining results as data arrives, e.g., [7, 13, 14]. The timing constraint and the order among the items also add more challenges to data mining techniques. For example, mining techniques that discover frequent patterns in market-analysis databases, e.g., [2, 11], assume that there is no order among the items in the same transaction. Such techniques are not suitable for mining time series databases in which the order among the items is of concern. Another challenge is that patterns of interest may contain gaps [19], i.e., patterns'

items are not necessarily contiguous, which makes the discovery of such patterns a difficult task.

In this paper, we propose an incremental algorithm for discovering the complete set of frequent patterns in time series databases, i.e., we discover the frequent patterns over the entire time series in contrast to applying a sliding window over a portion of the time series. The proposed approach has the ability to discover frequent patterns that contain gaps between patterns' items with a maximum user-defined gap size. With the arrival of each new data item, the algorithm updates the existing mining results incrementally. We define a set of states for the patterns in the database depending on whether they are frequent or non-frequent. Based on the transitions among these states, the algorithm takes certain actions to update the existing frequent patterns. To allow gaps within the frequent patterns, the algorithm maintains data structures that store information about the most recent portion of each time series that can contribute to the discovery of future frequent patterns. The size of this portion depends on the user-defined maximum gap size. To achieve scalability, the proposed algorithm uses the external memory with efficient indexing and searching mechanisms to handle the case in which the size of the time series database cannot fit entirely into memory.

The contributions of this paper are summarized as follows:

1. We propose an incremental algorithm for discovering the complete set of frequent patterns in time series databases. The algorithm updates the existing frequent patterns with the arrival of each new item to the database.

2. We allow the frequent patterns to contain gaps. The maximum gap size between any two consecutive items is less than a user-defined gap threshold.

3. We introduce several optimization techniques to enhance both the processing time and storage requirements of the proposed algorithm. For example, we define a property

of the time series frequent patterns that significantly reduces the number of candidate patterns, i.e., patterns likely to be frequent, that we need to process.

The rest of the paper proceeds as follows. In Section 2, we discuss the related work. In Section 3, we present our terminology and the problem definition. In Section 4, we introduce the data structures that we maintain during the execution of the algorithm. We present the details of the proposed algorithm in Sections 5 and 6. Section 7 contains the experimental results and performance evaluation. We conclude the paper in Section 8.

2 Related Work

Mining time series databases has been studied from different perspectives including segmenting and approximating time series data, e.g., [18, 22], indexing time series data for fast similarity search [6, 8], and mining pattern similarity in time series for the purpose of clustering, classification, or identifying trends [5, 12, 15, 21]. With respect to sequential pattern mining, which is our focus in this paper, several non-incremental techniques have been proposed for mining sequential patterns, e.g., [3, 9, 10, 19]. Most of them depend on the *Apriori* property proposed in [2]. These techniques assume a static database and perform multiple scans over the data to discover sequential patterns. GSP [19] is an Apriori-like technique that allows the incorporation of gap constraints within the sequential patterns. More recent approaches for mining sequential patterns are the pattern-growth approaches [16, 4]. PrefixSpan [16] is based on generating the sequential patterns recursively. Its main advantage is in the use of projected databases instead of the candidate generation step. PrefixSpan is shown to be faster than Apriori-like techniques including GSP. However, PrefixSpan does not allow gap constraints. GenPrefixSpan [4] is a generalization of PrefixSpan that allows sequential patterns to include gaps.

Unlike non-incremental mining, incremental mining for sequential patterns received less attention. ISM [14] is an algorithm for incremental sequence mining which is based on

SPADE [24]. ISM mines the database after transforming it into a vertical layout representation. ISM maintains two lists; "maximally frequent sequences" and "minimally infrequent sequences". The main drawback of ISM is the large storage overhead due to the vertical layout representation of the database and the large size of the maintained lists. Recently, two incremental techniques, IncSpan [7] and IncSP [13], have been proposed. IncSpan [7] maintains two thresholds, *min_supp* and μ*min_supp*, where $\mu \leq 1$. All the patterns with support $\geq$ *min_supp* are considered frequent patterns and are stored in FS list, whereas all the patterns with support between *min_supp* and μ*min_supp* are considered semi-frequent patterns and stored in SFS list. These two lists are searched for any newly formed candidate pattern to get its support. If the pattern is not found in either lists, then a database scan is performed to get the pattern's support. IncSP [13] divides the candidate patterns into two sets, X_k which is the set of candidate patterns that are currently frequent in the database and X_k' which is the set of candidate patterns that are currently not frequent. For the patterns in X_k, IncSP simply updates their count since IncSP maintains a list of the currently frequent patterns. However, IncSP needs to perform a database scan for the patterns in X_k' because it does not keep information about these patterns. IncSpan and IncSP are shown to outperform ISM with respect to both storage requirements and processing time. We should note that all incremental techniques assume batch updates for the database, i.e., the database is updated with batches of items. This assumption may not be suitable for some applications such as monitoring and online decision making.

A key distinction between our algorithm and the existing algorithms, e.g., IncSpan and IncSP, is that we focus on mining sequences of items instead of itemsets. This is motivated by the wide range of applications that generate sequences of items, e.g., sensor networks, stock prices, and environment monitoring applications. Using the proposed algorithm is shown to be more efficient than using the general sequence mining algorithms for such applications.

3 Terminology, Assumptions, and Problem Definition

We assume a database D consisting of m time series; D = (T_1, T_2, $\cdots$, T_m). Each time series T_i is composed of a sequence of items associated with timestamps and the items are sorted in an ascending order based on their timestamps; T_i = (t_1, t_2, t_3, $\cdots$). A new data item which arrives to a time series is appended at the end of this time series. The size of the database |D| is defined as the total number of items that arrived to all time series in the database. A pattern P of length l is defined as a sequence of l items; P = (p_1, p_2, $\cdots$, p_l). A time series T_i is said to contain pattern P with a gap threshold g if the items in P appear in T_i in the same order and the gap between any two consecutive items is less than or equal to g, i.e., there exist integers i_1, i_2, ..., i_l such that $i_2 \leq i_1 + g + 1$, $i_3 \leq i_2 + g + 1$, $\cdots$, $i_l \leq i_{l-1} + g + 1$ and $p_1 = t_{i1}$, $p_2 = t_{i2}$, $\cdots$, $p_l = t_{il}$.

The *relative support* of a data item, say a, in the database is defined as the number of times a appears in the database divided by |D|. Similarly, the relative support of a pattern, say P, in the database is defined as the number of times P appears in the database divided by (|D| - |P| + 1); where |P| is the pattern's length. Given a user-defined minimum support threshold *min_supp*, a pattern is said to be *frequent* if the support is greater than or equal to *min_supp*, and *non-frequent* otherwise.

In this paper, we address the following problem: given a database D, a minimum support threshold *min_supp*, and a maximum allowed gap threshold g, we need to discover the complete set of frequent patterns in D. Furthermore, as D is updated with the arrival of new data items, we need to update the discovered frequent patterns as well.

There exist various gap constraints other than *maximum gap*, e.g., *exact gap*, *minimum gap*, and *unlimited gap*. Each of these gap constraints has its own applications and usage. The *exact gap* constraint requires prior knowledge from users about the exact distance between the related items. Moreover, *exact gap* does not accommodate for unexpected and noise events that may occur because the number of such events is usually unknown. The *minimum*

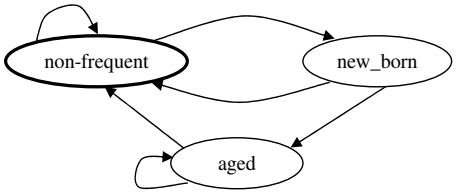


Figure 1: Patterns' state transitions

SID	SEQUENCE	EXPANDED
s_1	(20)	False
s_2	(30)	False
s_3	(50)	True
s_4	(50, 30)	False

(a) LDS_R_i

KEY	ENDING_SEQ
20	(s_1)
30	(s_2, s_4)
50	(s_3)
10	NULL

(b) LDS_L_i

Figure 2: LDS data structure

gap and *unlimited gap* are suitable for applications that allow items that are far away from each other, over the time dimension, to be related. *Maximum gap*, which we use in this paper, balances between the previous constraint types. It allows users to specify a maximum distance between the related items and accommodates for unexpected and noise events.

With the arrival of new items to the database, the frequency of the existing patterns changes. We define three states for the patterns in the database; *non-frequent*, *new-born*, and *aged* (See Figure 1). Initially, all patterns are non-frequent. When enough instances of the same pattern arrive to the database, i.e., the pattern's support exceeds *min_supp*, that pattern switches to the new-born state. The next arrival of an instance of a new-born pattern switches that pattern to the *aged* state. A frequent pattern may switch from the new-born or aged state to the non-frequent state if the pattern's support relative to the increasing database size becomes less than *min_supp*. These states and transitions will determine the actions taken by the algorithm upon the arrival of each new data item.

4 Data Structures

In this section, we describe the data structures that we maintain during the mining process. These data structures are of 2 types: (1) a local data structure, LDS, that is maintained separately for each time series in the database, and (2) a global data structure, GDS, that is shared by all time series in the database. LDS stores the most recent portion of each time series which may affect the discovery of the frequent patterns. The size of this portion depends on the gap threshold g. GDS stores the database items in a way that is more suitable for the mining algorithm and the frequent patterns discovered so far in the database.

4.1 LDS Data Structures

LDS keeps information about the most recent g items in each time series since these items may contribute to future frequent patterns. For each time series in the database, T_i, we maintain a counter, *COUNT(T_i)*, that reflects the current number of data items in T_i. *COUNT(T_i)* is incremented whenever a new data item arrives to T_i. In addition to *COUNT(T_i)*, LDS of T_i contains two tables, LDS_R$_i$ and LDS_L$_i$, that store the intermediate mining results found in the most recent g data items of T_i.

LDS_R$_i$ stores the frequent patterns that end with an item appearing in the most recent g data items in T_i. An example of LDS_R$_i$ is given in Figure 2(a). Each row in LDS_R$_i$ corresponds to one frequent pattern. *SID* is a unique identifier for each frequent pattern. *SEQUENCE* contains the frequent pattern's items. *EXPANDED* is a Boolean flag that specifies whether the corresponding pattern is expanded to generate longer patterns or not.

LDS_L$_i$ contains one entry for each data item in the most recent g items in T_i. An example of LDS_L$_i$ is given in Figure 2(b). *KEY* lists the most recent g data items sorted based on their relative positions in the time series, i.e., the most recent item in T_i is at the top of LDS_L$_i$ and the gth item in T_i is at the bottom of LDS_L$_i$. For a certain *KEY*, k, *ENDING_SEQ* contains the identifiers (which correspond to SID in LDS_R$_i$) of all frequent patterns that end with k. If k does not terminate any frequent patterns, then its corresponding *ENDING_SEQ* will be NULL. For example, assume a time series $T_i = (\cdots, 10, 50, 30, 20)$ which has the following frequent patterns: $S_1 = (20)$, $S_2 = (30)$, $S_3 = (50)$, $S_4 = (50, 30)$, and $g = 4$, then the entries of LDS_R$_i$ and LDS_L$_i$ are as given in Figure 2.

4.2 GDS Data Structures

GDS data structures consist of two tables; GDS_Q and GDS_H. GDS_Q is an alternative representation of the time series data that is more suitable for the mining algorithm. An example of GDS_Q is given in Figure 3(a). *KEY* lists, in a sorted order, all items that appear

KEY	COUNT	DATA
10	3	(T_1, 1), (T_2, 2, 4)
20	5	(T_1, 2, 4), (T_2, 3, 5), (T_3, 2)
30	4	(T_1, 3), (T_2, 1, 6), (T_3, 4)
40	2	(T_3, 1, 5)
60	1	(T_1, 5)
85	2	(T_2, 7), (T_3, 3)

(a) GDS_Q

FP_SEQ	COUNT	DATA
(10, 20)	3	(T_1, 2), (T_2, 3, 5)
(20, 30)	3	(T_1, 3), (T_2, 6), (T_3, 4)

(b) GDS_H

Figure 3: GDS data structures for all time series

in the database. For each *KEY*, k, *COUNT* represents the number of times k appears in the database, while *DATA* stores the time series identifiers in which k appears along with k's position(s) in each time series. These positions are kept sorted in an ascending order.

GDS_H stores all frequent non-singleton patterns found so far in the database. An example of GDS_H is given in Figure 3(b). GDS_H differs from GDS_Q in that GDS_H stores the frequent non-singleton patterns instead of the data items, i.e., *KEY* in GDS_Q is replaced with *FP_SEQ*, and *DATA* in GDS_H stores the ending position(s) of the corresponding frequent pattern in each time series sorted in an ascending order.

For example, assume that the database consists of three time series: T_1 = (10, 20, 30, 20, 60), T_2 = (30, 10, 20, 10, 20, 30, 85), and T_3 = (40, 20, 85, 30, 40), *min_supp* = 3, and g = 2, then the content of GDS_Q and GDS_H is as shown in Figure 3. Notice that although patterns (10), (20), and (30) are frequent, they are not stored in GDS_H because they are singleton patterns and their information can be retrieved directly from GDS_Q.

5 Mining the Complete Set of Frequent Patterns

The proposed algorithm needs to update the frequent patterns upon the arrival of each new data item. The algorithm decides which actions to take based on the occurrence of one or more of the events illustrated in Figure 4. In this section, we define each event, and in Section 5.1, we describe the actions taken by the algorithm upon detecting a certain event.

When a new instance of item k arrives to the database, k's state will be either non-frequent, new-born, or aged, which correspond to Events 1, 2, and 3, respectively, as illustrated in Figure 4. If k is new-born or aged frequent, then k may expand an already existing

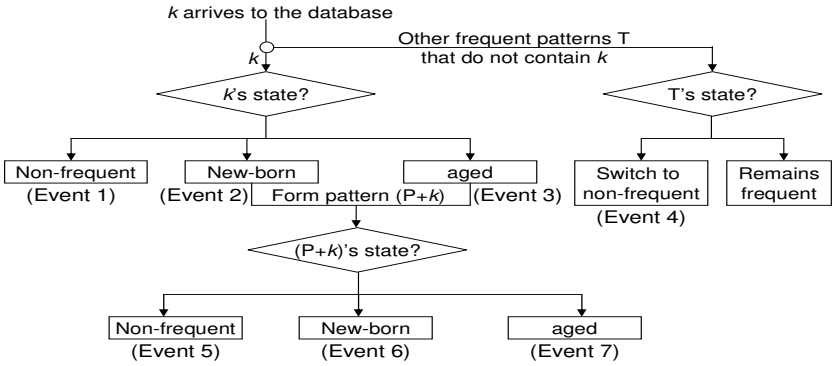


Figure 4: Events detected by the algorithm

Forming pattern (P + *k*)	**Counting pattern (P + *k*)**

Input: item *k* which arrived to time series T_i
Output: candidate patterns (P + *k*)
Steps:
- I = the set of the most recent g items in T_i (obtained from LDS_L_i)
- S = the set of all frequent patterns that end with items in I (obtained from LDS_R_i)
- For each pattern P in S
  - Return (P + *k*)
	Input: frequent pattern P and frequent item *k*
	Output: the count and position(s) of (P + *k*) in the database
	Steps:
	- If (P + *k*) exists in GDS_H Then
	  - Return the position(s) and the count of (P + *k*)
	- L_k = *k*'s entry in GDS_Q
	- L_p = P's entry in GDS_H
	- Join L_k and L_p. An occurrence of (P + *k*) is found if:
	  - (*k*'s position in $L_k \leq$ P's position in $L_p + g + 1$) and (k and P are in the same time series)
	- Return the position(s) and the count of (P + *k*)

Figure 5: Forming and counting pattern (P + k)

frequent pattern P to form a new candidate pattern (P + k); where '+' is the concatenation operation. The state of (P + k) will be either non-frequent, new-born, or aged, which correspond to Events 5, 6, and 7, respectively, as illustrated in Figure 4. The arrival of k increases the size of the database. Hence, an already frequent pattern T may switch to the non-frequent state if T's support becomes less than *min_supp* (Event 4).

To detect k's state upon the arrival of k, the algorithm retrieves *COUNT(k)* from k's entry in GDS_Q. If there is no entry for k in GDS_Q, then a new entry is added to GDS_Q with *COUNT(k)* = 1. Based on *COUNT(k)*, we decide whether k is non-frequent, new-born, or aged frequent. To detect (P + k)'s state, the algorithm searches for (P + k) in GDS_H which stores the frequent patterns discovered so far in the database. If we find (P + k) in GDS_H, then (P + k) is an aged frequent pattern. Otherwise, we get the support of (P + k) in the database using a *Counting* procedure that is described in Section 5.2. If the support of

(P + k) is equal to or greater than *min_supp*, then (P + k) is new-born frequent. Otherwise, (P + k) is non-frequent. For clarity, we present in Figure 5 a high level description of how the algorithm forms (P + k) and how the *Counting* procedure gets the count of (P + k).

5.1 Events Handling

5.1.1 New Arrivals of Non-Frequent Items and Patterns - (Events 1 and 5)

Handling Events 1 and 5 is straightforward. For Event 1, k is not frequent and cannot contribute to any frequent patterns. Therefore, we only need to add or update the entry of k in GDS_Q, and add an entry for k in LDS_L$_i$ with *ENDING_SEQ* = NULL. For Event 5, if we find that a candidate pattern (P + k) is non-frequent, we simply discard this pattern without any further processing.

5.1.2 New Arrivals of Aged Frequent Items and Patterns - (Events 3 and 7)

When an aged frequent item k arrives to time series T$_i$ there are two possibilities in which k can contribute to the set of frequent patterns, Backward Expansion and Forward Expansion.

- **Backward Expansion**

  k may expand already existing frequent patterns, e.g., P, to form pattern (P + k) (See Figure 5: Forming pattern (P + k)). A frequent pattern P that can be expanded by k has to satisfy the following two conditions: (1) P has to appear in the same time series as k, i.e., T$_i$, and (2) the gap between k and the last item in P is less than or equal to g. To retrieve all the frequent patterns that can be expanded by k we scan LDS_L$_i$, which contains the most recent g items in T$_i$, and for each item, e.g., x, we retrieve all the frequent patterns ending with x from LDS_R$_i$. Recall that *ENDING_SEQ* in LDS_L$_i$ cross references *SID* in LDS_R$_i$. For each retrieved frequent pattern P we form pattern (P + k) and check the following:

1. If (P + k) is in GDS_H, then (P + k) is an aged frequent pattern (Event 7). We increase *COUNT(P + k)* in GDS_H and add the new position to the *DATA* field. Then we insert (P + k) into LDS_R$_i$ and update the k's entry in LDS_L$_i$.

2. If (P + k) is not found in GDS_H, then the *Counting* procedure gets the count of (P + k) by joining the two lists, L$_k$ and L$_p$ (See Figure 5: Counting pattern (P + k)). Based on the retrieved count we process (P + k) as either a new-born frequent pattern (Event 6) or a non-frequent pattern (Event 5).

Notice that Backward Expansion is the procedure that extends the frequent patterns to generate longer patterns and allows the patterns' items to have a maximum gap of size g between any two consecutive items.

- **Forward Expansion**

  Since k is a frequent item, then the items that will come after k in T$_i$ may expand k or any frequent pattern that ends with k. Therefore, when k arrives to T$_i$ we insert k into the top of LDS_L$_i$, pushing all the entries in LDS_L$_i$ one step down and deleting the bottom entry. The *ENDING_SEQ(k)* will contain all the frequent patterns ending with k (they are discovered during the Backward Expansion procedure). The k's entry remains in LDS_L$_i$ until the next g items arrive to T$_i$. After that k's entry will reach the bottom of LDS_L$_i$ and it will be dropped automatically. When k's entry is dropped, all the frequent patterns ending with k in LDS_R$_i$ will be also dropped.

5.1.3 New-born Frequent Items and Patterns - (Events 2 and 6)

Handling Events 2 and 6 is similar to handling Events 3 and 7, except that Events 2 and 6 require extra steps to be performed because the new-born items or patterns have just switched their state from non-frequent to new-born frequent. For Event 2, when we detect that an item k is a new-born frequent item, we apply the Backward Expansion and Forward

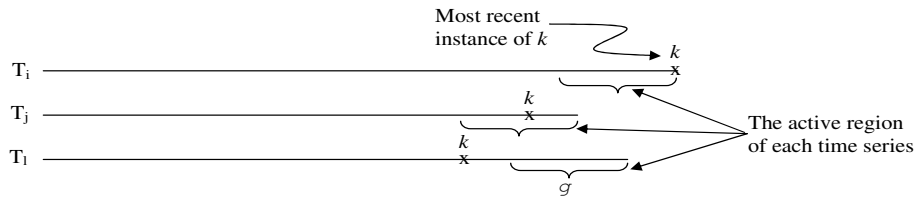


Figure 6: Reprocessing the instances in the active region of each time series

Expansion procedures to k as discussed in the previous section. However, the other instances of k that came to the database before the newly arrived instance are handled as being non-frequent and we may need to reprocess some of these instances to reflect that k is now frequent. The following Claim allows us to limit this extra processing to the minimal cost.

Claim 1: *The only instances of k in the database that we need to reprocess are the instances that appear in the active region of each time series in the database; where the active region of a time series is the most recent g data items of that time series.*

We explain the intuition behind Claim 1 by an example. Assume we have three time series in the database, T_i, T_j, and T_l, where the most recent arrival of k is in T_i as illustrated in Figure 6. When k arrives to T_i, k switches from a non-frequent state to a new-born state. Therefore, we insert k into LDS_L$_i$ along with all the frequent patterns that end with k. However, the k's instance which belongs to T_j is still in LDS_L$_j$ with *ENDING_SEQ* = NULL (because when this instance is processed k was non-frequent). Therefore, we need to insert an entry in LDS_R$_j$ indicating that k is a frequent item, and then we update the *ENDING_SEQ* value in LDS_L$_j$. On the other hand, the k's instance which belongs to T_l is already dropped from LDS_L$_l$ because more than g items came to T_l after k. We do not need to reprocess such instance because it cannot contribute to any future frequent patterns.

To find the instances of k that belong to an active region of a time series, we scan the k's entry in GDS_Q, which contains all the positions of k in all time series, and reprocess those instances as demonstrated in the example.

With respect to patterns, and following the same claim, any candidate pattern (P + k)

that is not found in GDS_H and is discovered via the *Counting* procedure to be frequent, i.e., (P + k) is a new-born frequent pattern, will be processed as follows. A new entry for (P + k) is inserted into GDS_H with all ending positions of (P + k) in the database. Recall that the *Counting* procedure reports the count and positions of a given pattern. Then we insert (P + k) into LDS_R$_i$ and update the k's entry in LDS_L$_i$. Additionally, for any instance of (P + k) that ends in the active region of any time series, e.g., T$_j$, we need to insert that instance into LDS_R$_j$ and reflect it in the k's entry in LDS_L$_j$.

5.1.4 Switching from the Frequent State to the Non-Frequent State - (Event 4)

When an item k switches from the frequent to the non-frequent state no extra processing is needed. The reason is that we keep k's entry in GDS_Q whether k is frequent or non-frequent.

To detect patterns that switch to the non-frequent state we need to scan GDS_H and check the support of each pattern P. If P's support is less than *min_supp*, then P is non-frequent and P's entry in GDS_H is deleted. However, scanning GDS_H with the arrival of every item to the database is very expensive. Therefore, we delay the detection of the patterns that became non-frequent until they are first referenced. For example, assume pattern P is in GDS_H and P changes its state to be non-frequent. When P is first referenced, we search for P in GDS_H, if P is found, then we compare P's current support against *min_supp*. If we find P frequent, then we process it as an aged frequent pattern. Otherwise, we detect that P is now non-frequent and delete its entry from GDS_H.

5.2 The Counting Procedure

The Backward Expansion (Section 5.1.2) is the procedure that extends the frequent patterns to generate longer patterns in the form of (P + k); where P is a frequent pattern and k is a frequent item. Given a pattern (P + k), the *Counting* procedure returns the count and the ending positions of (P + k) in the database. A sketch for the *Counting* procedure is

Algorithm 1 The Counting Procedure

```
//Search for P + k in GDS_H
- IF (P + k exists in GDS_H) and (support of (P + k) ≥ min_supp ) THEN
    //P + k is an aged frequent pattern (Event 7)
    - Increment COUNT(P + k) in GDS_H and update the DATA field
    - Return the count and positions of P + k
- ELSE IF (P + k exists in GDS_H) and (support of (P + k) < min_supp ) THEN
    //P + k switched to be a non-frequent pattern (Event 4)
    - Increment COUNT(P + k) in GDS_H and update the DATA field
    - IF (support of (P + k) is still less than min_supp) THEN
        - Delete (P + k) entry from GDS_H
    - END IF
    - Return the count and positions of P + k
- ELSE //P + k is not in GDS_H
    - Let L_k = k's entry in GDS_Q
    - IF (P is a singleton frequent pattern) THEN
        - Let L_p = P's entry in GDS_Q
    - ELSE
        - Let L_p = P's entry in GDS_H
    - END IF
    //Merge join L_p and L_k lists. The join condition is:
    //(k's position in L_k ≤ P's position in L_p + g + 1) and (k and P are in the same time series)
    - Let L_out = L_p ⋈ L_k
    - IF (support of (P + k) ≥ min_supp) THEN
        //P + k is a new-born frequent pattern (Event 6)
        - add entry for P + k in GDS_H
        - Return the count and positions of P + k
    - ELSE
        //P + k is a non-frequent pattern (Event 5)
        - Return the count and positions of P + k
    - END IF
- END IF
```

presented in Figure 5. The complete procedure is presented in Algorithm 1.

The procedure searches for (P + k) in GDS_H, if the pattern is found, then we check the support of (P + k) to determine if it is aged frequent or non-frequent (Lines 1-11). If (P + k) is not found in GDS_H, then we join the k's entry in GDS_Q with the P's entry in GDS_Q (if P is a singleton item) or in GDS_H (if P is a non-singleton item) to find the count and positions of (P + k) in the database (Lines 12-30). The join between k's entry and P's entry (Line 19-21) consists of a single pass over both entries since the entries are sorted. Notice that the join condition allows for a maximum gap of size g between P and k.

Correctness Proof (Algorithm 1):

We prove that Algorithm 1 returns the correct frequency and positions of (P + k) in the database based on the following facts: (1) P is a frequent pattern or item, (2) k is a frequent item, (3) list L_p contains the complete list of all P's ending positions in the database, and (4) list L_k contains the complete list of all k's positions in the database. If either of facts (1) or (2) is not correct, then (P + k) would not have been formed in the first place. List L_p is the P's entry in GDS_Q or GDS_H (Lines 14-17 in Algorithm 1), and list L_k is the k's entry in GDS_Q (Line 13 in Algorithm 1). We divide the proof into two cases.

Case 1 (P+k is not found in GDS_H): In this case, we join L_p and L_k to get the frequency and positions of (P + k). Since L_p and L_k are complete lists, then the obtained frequency and positions of (P + k) are correct.

Case 2 (P+k is found in GDS_H): We prove this case by *Induction*. (P + k) is added initially to GDS_H when (P + k) is found to be frequent for the first time. This step is performed by joining L_p and L_k which corresponds to Case 1 that is proved correct. Assuming that the current frequency and positions of (P + k) in GDS_H are correct, then a new arrival of (P + k) to the database will increment the frequency of (P + k) and add the new position to L_p . Therefore, the updated frequency and positions in GDS_H are correct.

We propose three different approaches, termed *Normal-Join*, *Enhanced-Join*, and *Optimized-*

Join, for handling the case in which the *Counting* procedure detects that (P + k) is not frequent. We describe the three approaches in the next subsections.

5.2.1 Normal-Join

In the *Normal-Join* approach, the *Counting* procedure discards any non-frequent patterns without storing them (Lines 26-29 in Algorithm 1). The *Counting* procedure presented in Algorithm 1 corresponds to the *Normal-Join* approach. *Normal-Join* does not require any extra storage. However, our experiments in Section 7 show that most of the mining time is spent in performing the expensive join operations. For example, assume we have *min_supp* set to 50 and the support of (P + k) is 20. This means that for the next 30 appearances of (P + k), we perform the join operation to get the support of (P + k), and subsequently discard this pattern as it is still non-frequent. To address this issue, we propose the *Enhanced-Join* and *Optimized-Join* approaches that significantly improve the mining time.

5.2.2 Enhanced-Join

In the *Enhanced-Join* approach, the *Counting* procedure stores the encountered non-frequent patterns in a new data structure, termed Candidate Frequent Patterns (CFP). CFP consists of two columns, *Pattern* which stores the patterns in the form P + k, and *P_Count* which stores the pattern's frequency. CFP does not store the pattern's positions. The *Counting* procedure presented in Algorithm 1 is slightly modified in the following way. If the procedure does not find (P + k) in GDS_H, then it searches for (P + k) in CFP. If (P + k) does not exist, then the procedure performs the join operation. Otherwise, the procedure only increments *P_COUNT(P + k)* in CFP and if the support exceeds *min_supp*, then we delete (P + k)'s entry from CFP and insert it into GDS_H.

In the *Enhanced-Join* approach, the number of join operations that may be applied to any pattern (P + k) until it becomes frequent is at most 2. The worst case is that after the first join operation (P + k) is found to be non-frequent, and hence it is inserted into CFP.

The second join operation is performed when (P + k) becomes frequent and move from CFP to GDS_H. The second join operation is performed to get the ending positions of (P + k).

Our experiments in Section 7 show that the *Enhanced-Join* approach outperforms the *Normal-Join* approach by more than one order of magnitude with respect to the mining time. However, the *Enhanced-Join* approach suffers from the dramatic increase in the size of CFP. This drawback motivates the *Optimized-Join* approach, described in the next section.

5.2.3 Optimized-Join

Similar to *Enhanced-Join*, *Optimized-Join* stores the encountered non-frequent patterns in CFP. *Optimized-Join* deploys two storage optimization techniques to reduce the size of the CFP list: (1) Eliminating unnecessary candidates, and (2) Candidate storage compaction.

(1) Eliminating Unnecessary Candidates

A candidate pattern has the form (P + k); where P is a frequent pattern and k is a frequent item. However, it is not necessary that (P + k) be a *real* candidate, i.e., a candidate to be frequent. If (P + k) is not a real candidate, then we can detect that (P + k) is non-frequent even without storing (P + k) in CFP or counting its support in the database. The Apriori property [2] states that a pattern of length t is a candidate frequent pattern only if all its sub-patterns of length $t-1$ are already frequent. The Apriori property is used to significantly reduce the number of candidate patterns. However, the Apriori property is not valid for time series frequent patterns due to the order of the patterns' items and the gap constraints within the patterns. For example, assume that the database consists of two time series; $T_1 = (\cdots, a, b, c, x_1, d, \cdots, a, b, c, x_2, d)$ and $T_2 = (\cdots, a, b, c, x_3, d)$, $g = 1$, and *min_supp* = 3. Then, pattern (a, b, c, d) is frequent, while, pattern (a, b, d) is non-frequent because the gap between b and d exceeds g. Although the Apriori property is not valid for the time series frequent patterns, the following Claim holds.

Claim 2: *A pattern S of length t is a real candidate only if the two sub-patterns S_1 and*

Frequent Pattern
(10)
(20, 10)
(30, 20, 10)
(40, 30, 20, 10)
(90, 40, 30, 20, 10)
(40, 20, 10)
(50, 10)
(20, 50, 10)
(40, 20, 50, 10)
(70, 20, 50, 10)

(a) Frequent patterns

Candidate Pattern	*Counting* Procedure Result
(10, 80)	Frequent
(20, 10, 80)	Not Frequent
(30, 20, 10, 80)	Not Frequent
(40, 30, 20, 10, 80)	Not Frequent
(90, 40, 30, 20, 10, 80)	Not Frequent
(40, 20, 10, 80)	Not Frequent
(50, 10, 80)	Not Frequent
(20, 50, 10, 80)	Not Frequent
(40, 20, 50, 10, 80)	Not Frequent
(70, 20, 50, 10, 80)	Not Frequent

(b) Candidate patterns

Figure 7: Counting procedure results

Pattern	Inserted into
(10, 80)	GDS_H
(20, 10, 80)	CFP
(30, 20, 10, 80)	CFP
(40, 30, 20, 10, 80)	CFP
(90, 40, 30, 20, 10, 80)	CFP
(40, 20, 10, 80)	CFP
(50, 10, 80)	CFP
(20, 50, 10, 80)	CFP
(40, 20, 50, 10, 80)	CFP
(70, 20, 50, 10, 80)	CFP

(a) The *Enhanced-Join* approach

Pattern	Inserted into
(10, 80)	GDS_H
(20, 10, 80)	CFP
(30, 20, 10, 80)	*Ignored*
(40, 30, 20, 10, 80)	*Ignored*
(90, 40, 30, 20, 10, 80)	*Ignored*
(40, 20, 10, 80)	*Ignored*
(50, 10, 80)	CFP
(20, 50, 10, 80)	*Ignored*
(40, 20, 50, 10, 80)	*Ignored*
(70, 20, 50, 10, 80)	*Ignored*

(b) The *Optimized-Join* approach

Figure 8: Algorithm actions

S_2 of length $t-1$ are already frequent; where S_1 and S_2 are equal to S without the leftmost and the rightmost data items, respectively.

The reason that only sub-patterns S_1 and S_2 have to be frequent is because any other sub-pattern of S of length $t-1$ will drop items from the middle of S, and this may violate the gap constraint imposed by the problem definition. Continuing with the example above, since pattern (a, b, c, d) is frequent, then patterns (a, b, c) and (b, c, d) have to be frequent.

Based on Claim 2, for a pattern (P + k) to be a real candidate, the two sub-patterns P and (P' + k) have to be already frequent; where P' equals to P without the leftmost data item. The first sub-pattern, i.e., P, is guaranteed to be frequent from the definition of the candidate patterns, thus it remains to check if (P' + k) is frequent or not. If (P' + k) is frequent, then (P + k) is a real candidate. Otherwise, (P + k) is not yet a real candidate. In *Optimized-Join*, CFP will contain only the real candidates that are not yet frequent.

The following example demonstrates how we can significantly reduce the size of CFP based on Claim 2. Assume that the frequent patterns ending with a data item (10) are as given in Figure 7(a). Let the data item coming after (10) be (80) which is found to be a frequent data item. Item (80) will extend the frequent patterns ending with (10) to generate the candidate patterns represented in Figure 7(b). Each of these patterns is checked by the *Counting* procedure to determine if the pattern is frequent or not. The results from the *Counting* procedure are given in Figure 7(b) and the actions taken by the *Enhanced-Join* and *Optimized-Join* approaches are given in Figures 8(a) and 8(b), respectively. Notice that

Enhanced-Join inserts all the candidate non-frequent patterns into CFP, whereas, *Optimized-Join* inserts only the two real candidates, i.e., (20, 10, 80) and (50, 10, 80), into CFP.

We present the *Counting* procedure with the optimization of *Eliminating Unnecessary Candidates* in Algorithm 2.

Correctness Proof (Algorithm 2): The proof of correctness of Algorithm 2 is similar to that of Algorithm 1 except for the case in which (P + k) is not found in GDS_H and is found in CFP. We prove this case by *Induction*. The first time (P + k) is added to the CFP list is performed by joining L_p and L_k, where (P + k) is found to be infrequent. Since L_p and L_k are complete lists, then the frequency of (P + k) stored in CFP is correct. Assuming the current frequency of (P + k) in CFP is correct, then each new arrival of (P + k) will increment the frequency in CFP until (P + k) becomes frequent. Therefore, the frequency of (P + k) in CFP is always correct. When (P + k) becomes frequent, we join L_p and L_k to get the ending positions of (P + k) in the database and move (P + k) from CFP to GDS_H. Since L_p and L_k are complete lists, then the obtained positions are also correct.

(2) Candidate Storage Compaction

The proposed algorithm assumes discrete data items. Without loss of generality, the data items domain can be mapped to a contiguous range of integer values. This mapping allows us to compress the CFP list to achieve further storage reduction in addition to the *Eliminating Unnecessary Candidates* optimization. The following example demonstrates how the *Storage Compaction* technique works. Assume that the candidate patterns stored in CFP (after applying the *Eliminating Unnecessary Candidates* optimization) are as given in Figure 9(a), and *min_supp* = 50. Instead of storing each candidate pattern separately, we can group the patterns based on P to form contiguous ranges instead of individual items as illustrated in Figure 9(b). Each entry in the compressed CFP represents a group of candidate patterns, and the *group support* (*g_supp* for short) represents the upper bound for the supports of the patterns participating in the group. This means that as long as *g_supp* is less than *min_supp*,

Algorithm 2 The Counting Procedure: Eliminating Unnecessary Candidates

```
//Search for P + k in GDS_H
- IF (P + k exists in GDS_H) and (support of (P + k) ≥ min_supp ) THEN //P + k is aged frequent (Event 7)
    - Increment COUNT(P + k) in GDS_H and update the DATA field
    - Return the count and positions of P + k
- ELSE IF (P + k exists in GDS_H) and (support of (P + k) < min_supp ) THEN //(Event 4)
    - Increment COUNT(P + k) in GDS_H and update the DATA field
    - IF (support of (P + k) is still less than min_supp) THEN
        - Delete (P + k) entry from GDS_H
    - END IF
    - Return the count and positions of P + k
- ELSE //P + k is not in GDS_H
    //Check if P + k is a real candidate
    - Let P' = P without the leftmost data item
    - IF (P' + k exists in GDS_H) and (support of (P' + k) ≥ min_supp ) THEN //P + k is a real candidate
        - IF (P + k exists in CFP) THEN
            - Increment P_COUNT(P + k) in CFP
            - IF (support of (P + k) ≥ min_supp ) THEN
                - Move P + k from CFP to GDS_H //Needs a join operation
                - Return the count and positions of P + k
            - ELSE
                - Return the count of P + k
            - END IF
        - ELSE //P + k is not in CFP
            - Let L_k = k's entry in GDS_Q
            - IF (P is a singleton frequent pattern) THEN
                - Let L_p = P's entry in GDS_Q
            - ELSE
                - Let L_p = P's entry in GDS_H
            - END IF
            //Merge join L_p and L_k lists. The join condition is:
            //(k's position in L_k ≤ P's position in L_p + g + 1) and (k and P are in the same time series)
            - Let L_out = L_p ⋈ L_k
            - IF (support of (P + k) ≥ min_supp) THEN //P + k is a new-born frequent pattern (Event 6)
                - add entry for P + k in GDS_H
                - Return the count and positions of P + k
            - ELSE //P + k is non-frequent but a real candidate
                - add entry for P + k in CFP
                - Return the count of P + k
            - END IF
        - END IF
    - ELSE //P + k is non-frequent and is not a real candidate
        - Return (P + k) is non-frequent
    - END IF
- END IF
```

Candidate Pattern P + k	Pattern's Support
(30, 50) + 10	24
(30, 50) + 11	20
(30, 50) + 12	13
(30, 50) + 13	5
(30, 50) + 14	41
(30, 50) + 17	20
(30, 50) + 18	33
(30, 50) + 19	17

(a) Uncompressed CFP

Candidate Group P + [k_i ...k_j]	Group's Support
(30, 50) + [10 ... 14]	41
(30, 50) + [17... 19]	33

(b) Compressed CFP

Figure 9: The frequent and candidate patterns

Candidate Group P + [k_i ...k_j]	Group's Support
(30, 50) + [10 ... 14]	41
(30, 50) + [15 ... 15]	28
(30, 50) + [17... 19]	33

(a) Add separate entry

Candidate Group P + [k_i ...k_j]	Group's Support
(30, 50) + [10 ... 15]	41
(30, 50) + [17... 19]	33

(b) Merge with the predecessor entry

Figure 10: The insertion of entry '(30, 50) + [15 $\cdots$ 15]'

then it is guaranteed that none of the group's patterns are frequent.

Applying the compression procedure incrementally, i.e., with the arrival of new data items, is a simple process. The CFP list is sorted based on pattern P and the entries corresponding to P are sorted based on k. Therefore, adding a pattern to CFP involves only checking the predecessor and the successor entries of the newly added entry for possible merging. For example, assume that we need to insert pattern '(30, 50) + 15' with support 28 into CFP (Figure 10). First, we insert a separate entry, i.e., '(30, 50) + [15 $\cdots$ 15]', into its proper position in CFP (Figure 10(a)) and then we check the predecessor and the successor entries for possible merging. We find that entry '(30, 50) + [15 $\cdots$ 15]' can be merged with its predecessor entry (Figure 10(b)). Notice that the group support of the newly constructed entry, i.e.,'(30, 50) + [10 $\cdots$ 15]' is the maximum support of the two merged entries.

In the case of inserting a pattern that already exists in CFP, e.g., pattern '(30, 50) + 12', we only increment the support of the entry in which the given pattern is participating, e.g., increment the support of entry '(30, 50) + [10 $\cdots$ 14]' in Figure 9(b) to be 42. It can be the case that '(30, 50) + 12' is not the pattern that has support 41 in the group. However, since we do not know the exact support of each pattern, we need to increment the group support to keep it always as the upper-bound for the supports of the patterns inside the group.

It remains to discuss how the *Counting* procedure handles the case in which *g_supp* of an entry in CFP exceeds *min_supp*. We introduce two states for each entry in CFP, termed *Safe* and *Unsafe*. The *Safe* state means that, most probably, the candidate patterns in the CFP entry are still not frequent, whereas the *Unsafe* state means that, most probably, one or more of the candidate patterns in the CFP entry are close to be frequent. We have two extreme cases. The *pessimistic case* assumes that the support distribution among the entry's candidate patterns is very skewed. Therefore, the pessimistic case assumes that an entry is *Safe* as long as the entry's *g_supp* is less than *min_supp*, i.e., it is guaranteed that none of the entry's patterns are frequent, and the entry is *Unsafe* otherwise. Whereas, the *optimistic case* assumes that the support distribution among the entry's candidate patterns is uniform. Therefore, the optimistic case assumes that an entry is *Safe* as long as the entry's *g_supp* is less than *(C * min_supp)*, where C is the number of candidates in that entry, and the entry is *Unsafe* otherwise.

The *Counting* procedure compromises between the two extreme cases by defining a factor $F \geq 1$, such that an entry's state in CFP is determined by the following equation:

$$State = \begin{cases} Safe & g_supp < min_spp \\ Safe & g_supp/C < min_spp/F \\ Unsafe & \text{Otherwise} \end{cases}$$

This means that an entry is considered *Safe* if *g_supp* is less than *min_supp* or the average pattern's support in the entry, i.e., (*g_supp/C*), is less than (*min_supp/F*). Notice that the optimistic case corresponds to $F = 1$, and the pessimistic case corresponds to $F = C$.

The action that the *Counting* procedure takes when an entry, say e, in CFP becomes *Unsafe* is to delete e from CFP. As a result, any appearance of a pattern that was part of e will cause the *Counting* procedure to perform a join operation to compute the exact pattern's support. The *Counting* procedure will perform this join operation because the pattern is a real candidate and it is neither in GDS_H nor in CFP.

The effect of F on the algorithm performance is as follows: If F is very small (close to 1), then the detection of a frequent pattern may be delayed because we may consider an entry to be *Safe* while it contains frequent patterns. If F is very large, then we may pay extra processing time for deleting *Unsafe* entries and performing extra join operations while the patterns of the deleted entries have low support. Since the best value for F depends on the distribution of the data which may change over time, we propose a heuristic that dynamically adjusts F. The heuristic works as follows:

- F will have a large initial value, e.g., 20, which is very pessimistic.
- The *Counting* procedure will mark the *Unsafe* entries as deleted instead of physically deleting them. These entries are kept, for a while, only to vote for either incrementing, decrementing, or fixing the value of F, and then they will be physically deleted. We do not take the marked entries into account while searching for a given pattern in CFP.
- The database is logically divided into batches, e.g., batches of size 500 items. At the end of each batch, i.e., after the arrival of each 500 items, the marked entries in CFP will vote for F. Based on the collected votes F will be updated and the marked entries will be physically deleted from CFP.
- A marked entry e in CFP votes for F based on one of the following rules:
  1. If at least one of e's patterns appears in the database (after e is marked) and the *Counting* procedure finds that the pattern's support is larger than or equal to *min_supp*, then e votes to increment F (region C).
  2. If at least one of e's patterns appears in the database (after e is marked) and the *Counting* procedure finds that the pattern's support is between μ*min_supp* and *min_supp*, where $\mu < 1$, then e votes to keep F at the current value (region B).
  3. Otherwise, e votes to decrement F (region A).

The choice of μ determines the size of regions A and B. We found experimentally that setting μ to 0.75 is a reasonable choice.

- After collecting the votes from all the marked entries, the value of F will be updated based on the following rules:

  1. If there is at least one vote to increment F, then F will be incremented.
  2. If there is at least one vote to fix F or there are no votes (there are no marked entries), then F will not change.
  3. Otherwise, F will be decremented.

The heuristic is eager to increment or fix F than to decrement F. This is because we want to decrement F only when we are almost sure that the detection of the frequent patterns will not be affected, i.e., delayed. In Section 7 we show that this heuristic works very well when we dynamically adjust the value of F.

6 Memory Management

The data structures maintained by the algorithm are divided into (1) local data structures that include LDS_R$_i$, and LDS_L$_i$ for each time series T$_i$, and (2) global data structures that include GDS_Q, GDS_H, and CFP. We assume that the local data structures can fit entirely into the main memory since their size is usually very small. However, the size of GDS_Q and GDS_H can be large and may exceed the available main memory. Therefore, we need to use the external memory to store parts of these global structures in case they cannot fit entirely into memory.

The idea is to allocate a fixed partition for each of GDS_Q and GDS_H in the main memory. Then, whenever we need to add a new entry to GDS_Q or GDS_H and the corresponding partition is full, we move a subset of the *least recently referenced* entries from that partition to disk. The size of the subset to be moved to the disk is a heuristic parameter

and we evaluate its effect on the algorithm performance in Section 7. Basically, if the subset size is too small, then the memory partition fills up very quickly and data is moved to the disk more frequently. If the subset size is too large, then the memory is under utilized and more disk operations are needed. To keep track with which entries are the least recently referenced, we add a *Timestamp* column to both GDS_Q and GDS_H. A timestamp of an entry is updated whenever that entry is referenced, i.e., the entry's pattern is referenced by the algorithm.

The search for a given key k in GDS_Q proceeds as follows. First, we search for k among the main memory entries. If k is not found, we search for k on the disk. To allow efficient disk-based operations over GDS_Q, we build a B-tree index over the KEY column of GDS_Q. The B-tree index scales very well with the increase in the number of keys in the database and allows fast retrieval of a given key entry from disk. If k is found on the disk, we copy k's entry to the main memory and update the entry's timestamp. Copying an entry from the disk to the memory may require moving a subset of the *least recently referenced* entries of the targeted partition to the disk if the partition is full.

For the GDS_H structure, the main operations we perform are searching for and deleting a given pattern p. Building an index over GDS_H is slightly more complex than that over GDS_Q because the FP_SEQ column in GDS_H is a multi-dimensional key. Therefore, we define a function *key_fun()* that computes a single value for each pattern in FP_SEQ, and we build a B-tree index over these computed values. An example of *key_fun()* is to sum the keys of a given pattern. The search for a given pattern p in GDS_H proceeds as follows. First, we search for p among the main memory entries. If p is not found, we search the B-tree index for *key_fun(p)*. The index may return multiple entries that match *key_fun(p)* because many patterns may map to the same indexed value. Therefore, we need to scan the returned entries to find the pattern matching p (if it exists). If p is found on the disk, we copy p's entry to the main memory and update the entry's timestamp. The deletion

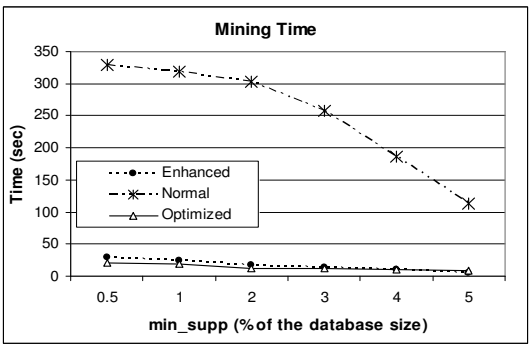
of a given pattern p from GDS_H requires deleting p from both the main memory and the disk. A pattern that turned to be non-frequent in GDS_H is deleted when the pattern either re-appears or becomes among the *least recently referenced* patterns in GDS_H.

With respect to the CFP list, the size of CFP can be limited to the available main memory without a need to swap entries between the main memory and the disk. The reason is that CFP is an auxiliary list that speeds up the algorithm. Unlike GDS_Q and GDS_H, deleting entries from CFP does not affect the mining results completeness. Therefore, we allocate a memory partition for CFP, and when we need to add a new entry to CFP and the memory partition is full, we delete the *least recently referenced* entry in CFP to make room for the new entry. To keep track with which entry is the least recently referenced, we add a *Timestamp* column to CFP. A timestamp of an entry is updated whenever that entry is referenced by the algorithm.

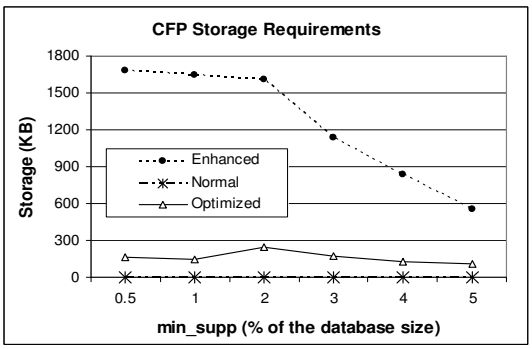
7 Performance Evaluation

In this section, we study the performance of the various join approaches of the proposed algorithm, i.e., *Normal-Join, Enhanced-Join,* and *Optimized-Join.* We also compare the proposed algorithm with two recent incremental techniques; IncSpan [7] and IncSP [13].

The dataset used in the experiments is generated using a *Synthetic Classification Data Set Generator* from Simon Fraser University (SCDS) [1, 17]. SCDS generates synthetic data sets of various distributions. The dataset consists of 500,000 items distributed over 5 time series with 250 distinct keys. The 500,000 items are generated as pairs, each pair consists of a time series identifier that is uniformly distributed over interval [1...5] and a value that is uniformly distributed over interval [1...250]. We vary the values of the following parameters: (1) the minimum support threshold (*min_supp*), and (2) the maximum gap size (g). In the case of *Optimized-Join*, we assign F an initial value of 20 which will change dynamically during the execution.



(a) Time performance

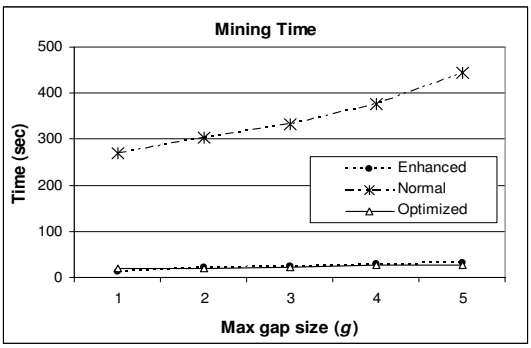


(b) CFP storage requirements

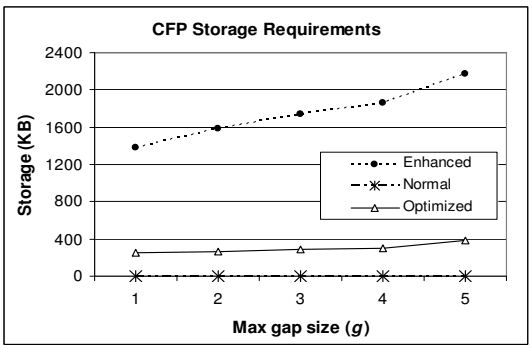
Figure 11: Comparison of the proposed join approaches with varying *min_supp*

We study the performance of the *Normal-Join, Enhanced-Join,* and *Optimized-Join* approaches in Figures 11 and 12. In Figure 11, we fix $g = 2$ and vary *min_supp* from 0.5% to 5%. Figure 11(a) illustrates that *Enhanced-Join* and *Optimized-Join* achieve significant time improvement over *Normal-Join*. This is because they store the candidate patterns in the CFP list and hence avoid many join operations. In Figure 11(b), we present the CFP storage requirements for the various join approaches. The storage overhead of *Enhanced-Join* drops quickly with the increase of *min_supp* because the number of frequent patterns and the candidate patterns drops significantly. Notice that *Optimized-Join* has a peak at *min_supp* = 2% which appears due to the increase in the number of CFP entries. The reason for this increase is not because the number of real candidate patterns increases, but because many patterns that are real candidates at *min_supp* = 1% are not real candidates at *min_supp* = 2%. As a result, many of the CFP entries could not merge together as they could not form contiguous ranges any more. With further increase in *min_supp*, the number of real candidate patterns in CFP becomes smaller and the size of CFP decreases.

In Figure 12, we study the performance of the three join approaches with fixed *min_supp* = 1% and g varying from 1 to 5. With the increase in g, the number of discovered frequent patterns increases, and hence the mining time and the storage requirements also increase. The figure demonstrates the efficiency of the *Optimized-Join* approach over the

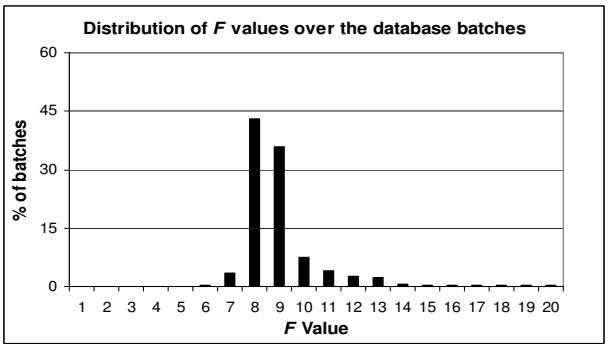


(a) Time performance

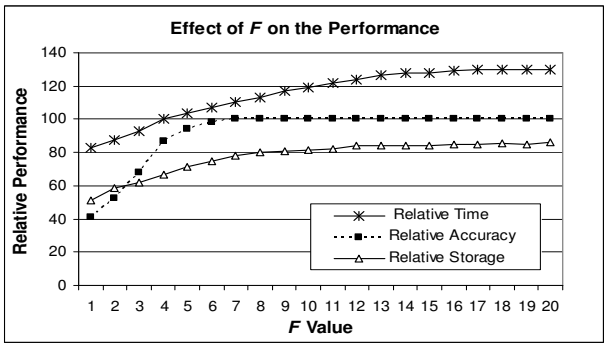


(b) CFP storage requirements

Figure 12: Comparison of the proposed join approaches with varying g



(a) F distribution using the heuristic



(b) Algorithm performance with fixed F

Figure 13: The performance of the heuristic for adjusting F

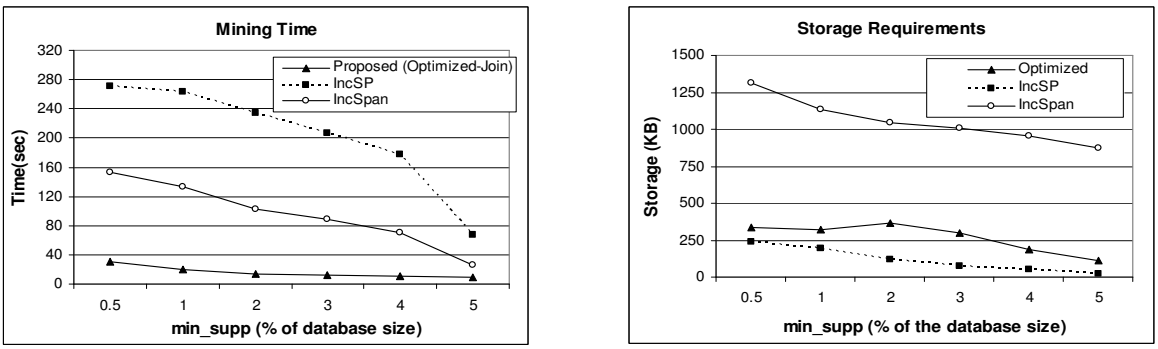
Enhanced-Join and *Normal-Join* approaches.

In Figure 13, we study the effect of F on the algorithm performance. We set *min_supp* = 1% and $g = 3$. In Figure 13(a), we present the distribution of the values of F over the database batches, i.e., we count the percentage of batches in which F is assigned a certain value. The figure shows that in most of the batches F has the values 8 or 9. This means that the heuristic finds the initial value of F (which is 20) large and decrements F until F reaches a value that suits the data distribution. The smallest value assigned to F is 6, which means that when $F = 6$ the heuristic detects a delay in detecting the frequent patterns and hence increments F again.

To check how well the heuristic works, we repeat the same experiment while fixing F to

a certain value, i.e., F does not change during the algorithm execution (Figure 13(b)). We define the *relative accuracy* of the algorithm as the number of discovered frequent patterns using the *Candidate Storage Compaction* optimization divided by the number of discovered frequent patterns without using the optimization. The *relative time* is defined as the time using the *Candidate Storage Compaction* optimization divided by the time without using the optimization. Similarly, the *relative storage* is defined as the CFP size using the *Candidate Storage Compaction* optimization divided by the CFP size without using the optimization. The figure illustrates that the algorithm reaches 100% relative accuracy with 20% storage saving at $F = 7$. If $F > 7$, the algorithm pays extra processing time without gaining more accuracy due to considering many CFP entries *UnSafe* and deleting them although the patterns' support in these entries is low. If $F < 7$, the accuracy of the algorithm as well as the mining time and the storage requirements decrease. This is because most of the CFP entries will be considered *Safe* although they may contain frequent patterns. As a result, CFP will be very compacted (entries can merge together) and the number of join operations will be small. From Figures 13(a) and 13(b) we conclude that the heuristic performs very well by assigning F to values 8 and 9 most of the time which is very close to the optimal value 7 (a bit more pessimistic).

We now compare our Optimized-Join technique against two incremental techniques; IncSpan [7] and IncSP [13]. IncSpan and IncSP are shown to outperform ISM [14], GSP [19], and PrefixSpan [16]. IncSpan and IncSP assume a database that consists of sequences of itemsets. The support of a pattern P is computed as the percent of sequences that contain P. We slightly modified IncSpan and IncSP to make them comparable with our technique as follows. We set the itemset size to 1 such that the database sequences consist of singleton items instead of itemsets. The support definition is updated to the definition introduced in Section 3. The incremental update for the mining results is performed with the arrival of every new item instead of a batch update.



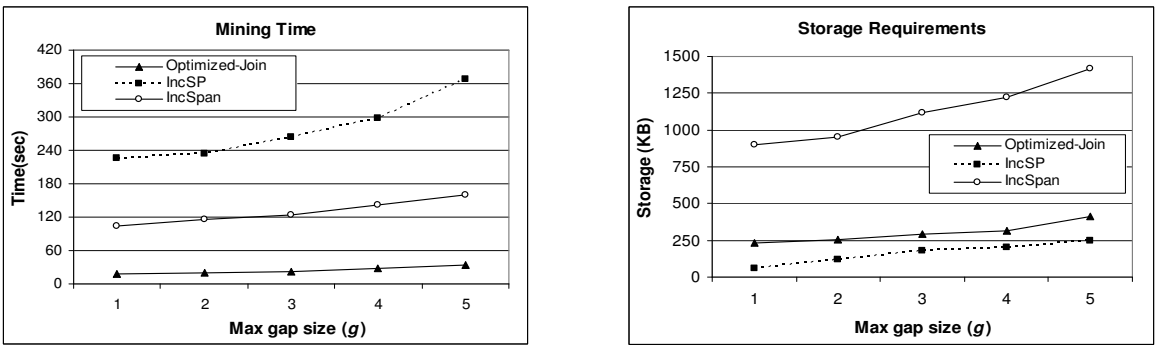
(a) Mining time

(b) Storage requirements

Figure 14: The effect of varying *min_supp*

We first compare the three algorithms by fixing the gap size and varying *min_supp*. Figure 14(a) compares the mining time taken by IncSpan, IncSP, and Optimized-Join. For IncSpan, we set the buffer ratio μ to 0.8 as suggested by [7]. We set $g = 3$ and vary *min_supp* from 0.5% to 5%. The figure illustrates that Optimized-Join is much faster than IncSpan and IncSP. IncSP is the slowest because the candidate generation phase is performed on multiple phases. Additionally, scanning the database multiple times to get the support of the new candidate patterns is a very expensive operation. IncSpan has a better mining time than IncSP due to buffering the semi-frequent patterns which saves many database projections. However, checking every pattern in the frequent and semi-frequent lists with every update in addition to the database projection are still a significant overhead. *Optimized-Join* is the fastest because the LDS data structures allow fast candidate generation. The CFP list saves many database scans, and if a database scan is necessary, the *Counting* procedure performs the scan as a fast join operation between two lists.

With respect to storage requirements, we focus on the temporary storage required by each technique, i.e., storage other than the input database and the output frequent patterns. The temporary storage of Optimized-Join consists of the LDS data structures and the CFP list. The temporary storage of IncSpan consists of the semi-frequent patterns buffer (SFS) and the projected databases. The temporary storage for IncSP consists of the candidate



(a) Mining time

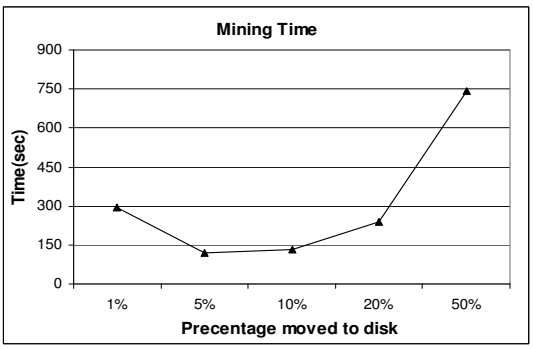
(b) Storage requirements

Figure 15: The effect of varying gap size g

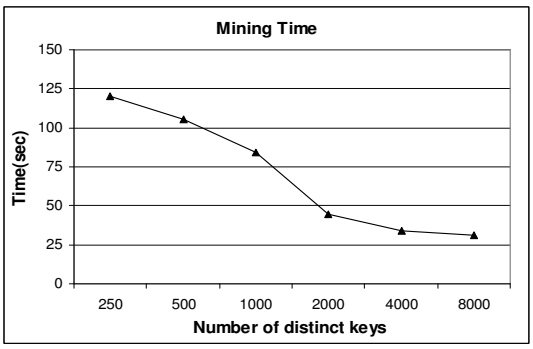
hash-tree. Figure 14(b) illustrates that IncSP has the lowest storage overhead while IncSpan is the highest. The reason is that IncSpan stores all the patterns with support between *min_supp* and μ*min_supp, where μ is set to 0.8, in the SFS list. The number of these semi-frequent patterns is large. Also, the size of the projected databases adds additional storage overhead. The *Optimized-Join* requires a little bit more storage than IncSP because of the CFP list. This can be justified by the efficient mining time our technique has. However, this storage overhead is much less than IncSpan due to applying the storage optimization techniques discussed in Section 5.2.3.

In Figure 15, we illustrate the effect of varying g from 1 to 5 and fixing *min_supp* to 1%. The experiment yields results similar to the previous comparison. Figure 15(a) illustrates that Optimized-Join has the least mining time, while IncSP has the worst mining time. This is because IncSP performs several phases to generate the candidates and several database scans to get their support. Figure 15(b) illustrates that IncSpan has a significant storage overhead compared to Optimized-Join and IncSP. The is because the size of the semi-frequent patterns list grows dramatically with the increase of g.

To evaluate the disk-based performance of the algorithm we used PostgreSQL [20] to facilitate the disk-based processing. We implemented GDS_Q and GDS_H as tables inside the database and built the B-tree indexes over the tables as described in Section 6. The update



(a) Varying the subset size moved to disk



(b) Varying the number of distinct keys

Figure 16: Disk-based Processing

and search operations over GDS structures are implemented as database calls. We did not re-evaluate the relative performance for the *Optimized-Join*, *Enhanced-Join*, and *Normal-Join* approaches because it depends mainly on the CFP list which still resides entirely in the main memory. The disk-based performance comparison against IncSpan [7] and IncSP [13] is not possible because both algorithms do not provide a mechanism for the disk-based processing.

We present the disk-based performance of the proposed algorithm in Figure 16. In Figure 16(a), we measured the mining time while varying the size of the subset moved to the disk when a memory partition is full. The subset size is a percentage of the partition size. We set $g = 3$ and $min_supp = 1\%$. The figure illustrates that if the subset percentage is very small, e.g., 1%, or very large, e.g., 50%, then the algorithm performs many database calls and takes more mining time than needed. Indeed, in the former case, the memory fills up very rapidly and many database calls are performed to free part of the memory. However, in the latter case many of the entries moved to the disk are soon referenced and retrieved again from disk, therefore many unnecessary database calls are performed.

In Figure 16(b), we test the scalability of the algorithm by varying the number of distinct keys in the database. We set $g = 3$, $min_supp = 1\%$, and the subset percentage moved to disk = 5%. The figure illustrates that as the number of distinct keys increases the mining time decreases. The reason is that with the increase in the number of distinct keys the

frequency of each key in the database decreases, and hence the number of frequent patterns significantly decreases.

8 Conclusion

We presented a new incremental mining algorithm for discovering the complete set of frequent patterns with a user-defined maximum gap constraint in time series databases. The incremental nature of the proposed algorithm makes it efficient and scalable for mining continuously updated data, where the arrival of each new data item triggers certain events that update the existing mining results. We introduced several optimization techniques to enhance both the processing time and storage requirements of the proposed algorithm. In particular, we proposed three approaches, namely *Normal-Join*, *Enhanced-Join*, and *Optimized-Join*, for efficient implementation of the proposed algorithm. The *Optimized-Join* approach shows significant improvement over the other two approaches. We presented a disk-based mechanism that allows the proposed technique to scale efficiently with the increase in the database size. We compared the *Optimized-Join* approach with two recent incremental techniques IncSpan and IncSP. The experiments illustrate that our approach outperforms both techniques in terms of the mining time and requires a little bit more storage than IncSP.

References

[1] GenData from Dataset Generator: http://www.datasetgenerator.com/.

[2] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Very Large Data Bases Conference*, 1994.

[3] R. Agrawal and R. Srikant. Mining sequential patterns. In *International Conference on Data Engineering*, pages 3–14, 1995.

[4] C. Antunes and A. L. Oliveira. Generalization of pattern-growth methods for sequential pattern mining with gap constraints. In *Machine Learning and Data Mining in Pattern Recognition*, pages 239–251, 2003.

[5] W. G. Aref, M. G. Elfeky, and A. K. Elmagarmid. Incremental online and merge mining of partial periodic patterns in time-series databases. In *TKDE*, volume 16, pages 332–342, 2004.

[6] K. Chakrabarti, E. Keogh, S. Mehrotra, and M. Pazzani. Locally adaptive dimensionality reduction for indexing large time series databases. In *ACM Transactions on Database Systems*, 27(2):188–228, 2002.

[7] H. Cheng, X. Yan, and J. Han. Incspan: incremental mining of sequential patterns in large database. In *Knowledge Discovery and Data Mining*, pages 527–532, 2004.

[8] C. Faloutsos, M. Ranganathan, and Y. Manolopoulos. Fast subsequence matching in time-series databases. In *Special Interest Group on Management Of Data Conference*, pages 419–429, 1994.

[9] M. N. Garofalakis, R. Rastogi, and K. Shim. SPIRIT: Sequential pattern mining with regular expression constraints. In *Very Large Data Bases Journal*, pages 223–234, 1999.

[10] J. Han, G. Dong, and Y. Yin. Efficient mining of partial periodic patterns in time series database. In *International Conference on Data Engineering*, pages 106–115, 1999.

[11] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *Special Interest Group on Management Of Data Conference*, pages 1–12, 2000.

[12] P. Indyk, N. Koudas, and S. Muthukrishnan. Identifying representative trends in massive time series data sets using sketches. In *Very Large Data Bases Conference*, pages 363–372, 2000.

[13] M. Y. Lin and S. Y. Lee. Incremental update on sequential patterns in large databases by implicit merging and efficient counting. In *Information Systems*, 29(5):385–404, 2004.

[14] S. Parthasarathy, M. J. Zaki, M. Ogihara, and S. Dwarkadas. Incremental and interactive sequence mining. In *ACM Conference on Information and Knowledge Management* , pages 251–258, 1999.

[15] P. Patel, E. Keogh, J. Lin, and S. Lonardi. Mining motifs in massive time series databases. In *International Conference on Data Mining*, 2002.

[16] J. Pei, J. Han, B. Mortazavi-Asl, H. Pinto, Q. Chen, U. Dayal, and M.-C. Hsu. Prefixspan: Mining sequential patterns efficiently by prefix-projected pattern growth. In *ICDE*, pages 215–224, 2001.

[17] Jian Pei, Runying Mao, Kan Hu, and Hua Zhu. Towards data mining benchmarking: a test bed for performance study of frequent pattern mining. In *SIGMOD*, pages 592–592, 2000.

[18] B. K. Pratt and E. Fink. Search for patterns in compressed time series. In *International Journal of Image and Graphics*, 2002.

[19] R. Srikant and R. Agrawal. Mining sequential patterns: Generalizations and performance improvements. In *International Conference on Extending Database Technology*, pages 3–17, 1996.

[20] M. Stonebraker and G. Kemnitz. The postgres next generation database management system. *Commun. ACM*, 34(10):78–92, 1991.

[21] Ajumobi Udechukwu, Ken Barker, and Reda Alhajj. Discovering all frequent trends in time series. In *Winter International Symposium on Information and Communication Technologies*, pages 1–6, 2004.

[22] C. Wang and X. S. Wang. Supporting content-based searches on time series via approximation. In *International Conference on Scientific and Statistical Database Management*, pages 69–81, 2000.

[23] Ke Wang. Discovering patterns from large and dynamic sequential data. *Journal of Intelligent Information Systems*, 9(1):33–56, 1997.

[24] M. J. Zaki. Efficient enumeration of frequent sequences. In *ACM Conference on Information and Knowledge Management* , pages 68–75, 1998.