

# CISpan: Comprehensive Incremental Mining Algorithms of Closed Sequential Patterns for Multi-Versional Software Mining

Ding Yuan<sup>†</sup>, Kyuhyung Lee<sup>†</sup>, Hong Cheng<sup>†</sup>, Gopal Krishna<sup>†</sup>, Zhenmin Li<sup>‡</sup>,  
Xiao Ma<sup>†</sup>, Yuanyuan Zhou<sup>†‡</sup> and Jiawei Han<sup>†</sup>

<sup>†</sup>University of Illinois at Urbana-Champaign, Urbana, Illinois, USA

<sup>‡</sup>CleanMake Inc., Urbana, Illinois, USA

{dyuan3, kyuhlee, hcheng3, gkrishn2, zli4, xiaoma2, yzhou, hanj}@cs.uiuc.edu

## Abstract

Recently, frequent sequential pattern mining algorithms have been widely used in software engineering field to mine various source code or specification patterns. In practice, software evolves from one version to another in its life span. The effort of mining frequent sequential patterns across multiple versions of a software can be substantially reduced by efficient incremental mining. This problem is challenging in this domain since the databases are usually updated in all kinds of manners including insertion, various modifications as well as removal of sequences. Also, different mining tools may have various mining constraints, such as low minimum support. None of the existing work can be applied effectively due to various limitations of such work. For example, our recent work, IncSpan, failed solving the problem because it could neither handle low minimum support nor removal of sequences from database.

In this paper, we propose a novel, comprehensive incremental mining algorithm for frequent sequential pattern, CISpan (Comprehensive Incremental Sequential Pattern mining). CISpan supports both closed and complete incremental frequent sequence mining, with all kinds of updates to the database. Compared to IncSpan, CISpan tolerates a wide range for minimum support threshold (as low as 2). Our performance study shows that in addition to handling more test cases on which IncSpan fails, CISpan outperforms IncSpan in all test cases which IncSpan could handle, including various sequence length, number of sequences, modification ratio, etc., with an average of 3.4 times speedup. We also tested CISpan's performance on databases transformed from 20 consecutive versions of Linux Kernel source code. On average, CISpan outperforms the non-incremental CloSpan by 42 times.

**Keywords:** Incremental mining, Software Engineering, Cross Module Mining, Frequent pattern

## 1 Introduction

**1.1 Motivation** Frequent sequential pattern mining [16, 13, 12, 15] is an important and active research topic in data mining with broad applications, including mining web logs, customer shopping transaction analysis, and DNA sequences, etc.

These years also saw an increasing trend of utilizing frequent pattern mining in source code mining [5, 7, 14, 1, 9, 6] and software specification mining [8]. These tools tokenize the source code in certain ways into a sequence database representation, and mine the frequent patterns in order to extract various information such as copy-pasted code segments [5], API usage [14, 1], programming rules [6], etc. For example, CP-Miner [5] is a tool to effectively detect copy-pasted code segments and copy-paste related bugs from source code. It first tokenizes each statement of source code into an integer, and maps the code into a sequential database, where each sequence corresponds to a basic block of the source code. By mining the closed frequent sequences with the minimum support,  $min\_sup$ , of 2, which suggests that the code segment was at least repeated once, the tool detects all the copy-pasted code segments within the source code.

In reality, database evolves in an incremental manner. For example, the latest major version of Linux Kernel, 2.6.X series, consists of more than 150 versions in total, which would be transformed into more than 150 similar databases. The difference between these databases could involve removal, various kinds of modifications and insertion of sequences, corresponding to removal, modification and insertion of code segments into the source code. If we mine each updated database from scratch, the majority of time and space would be spent on repetition of the previous mining process, which is something we should and could optimize. In our result, we will demonstrate that by using our CISpan algorithm, we can reduce the mining time of the updated version database transformed from Linux Kernel source code by 42 times on average.

Because of the significance of this problem, many studies have contributed to making sequential pattern mining incremental [4, 10, 17, 3]. Our recent work, IncSpan [4], demonstrated it significantly outperforms the non-incremental sequential mining algorithm [12] and the previously proposed incremental algorithm [10], and thus is considered the state-of-the-art of incremental sequential pattern mining. When mining the original database, IncSpan not only buffers the frequent sequences, but also sequences that are *semi-frequent*, which are likely to become frequent in the new version. Later when mining the new version, only sequences with support over a certain threshold will likely become frequent out of un-buffered sequences.

However, during our attempts to apply IncSpan into the code mining tools, we found that IncSpan is neither general enough to handle real life database evolution nor real life mining requirement (for example, CP-Miner requires minimum support threshold to be 2). In addition, the performance of IncSpan is far from optimal. The limitation and cost of IncSpan can be summarized as follows.

- **Failure to handle removal of sequences.** IncSpan could only handle insertion of sequences or appending items into the tail of each sequence. The databases in real world would evolve in far more varied ways. For example, the source code database we encountered would involve removal, various kinds of modifications such as adding or removing items, as well as insertion of the sequences.
- **Difficulties for mining frequent sequences with low minimum support.** IncSpan buffers *semi-frequent* sequences, sequences with support between  $[\mu \times min\_sup, min\_sup)$  where  $\mu$  is the buffer ratio between 0 and 1. When the *min\_sup* itself is small, IncSpan would end up buffering huge amount of semi-frequent sequences, and significantly degrade the performance. In the CP-Miner context, the *min\_sup* is 2. In this extreme case, IncSpan will not be effective at all, since it has to buffer semi-frequent sequences with support 1, that are all the subsequences in the database (exponential of sequence number in the database).
- **Unable to mine closed sequences.** In many context, we only care about closed sequences. Given the closed frequent sequences, we can always derive the set of complete frequent sequences. In CP-Miner’s context, it reports only the closed frequent sequences, which represent the largest copy-pasted code region.
- **Require difficult parameter tuning.** In the IncSpan algorithm, user has to specify the semi-frequent buffer ratio,  $\mu$ . However, without the knowledge of the details of the algorithm, it is difficult for users to tune this to an optimal buffer ratio.

Our initial motivation comes from the effort to optimize CP-Miner [5] to work in an incremental manner to analyze software evolution patterns. In CP-Miner, after tokenizing the program into a sequential database, it applies CloSpan [15] to mine all the closed frequent sequences out from the database. The evolution of software results into removal, modification and insertion of sequences in our database, and the *min\_sup* for CP-Miner is 2. After a careful survey of the previous work, we concluded that none of existing work, to the best of our knowledge, could be applied to solve our problem effectively. Bearing this motivation in mind, we set our foot to develop a comprehensive solution to incremental frequent sequential pattern mining problem.

**1.2 Challenge** Any update to a database can be modeled as removals and insertions of sequences. If a sequence  $\alpha$  in old database is modified to  $\alpha'$  in the new database, it could be treated as  $\alpha$  removed from old database and  $\alpha'$  is inserted into the new database. Thus from now on we will simply refer to all kinds of updates to removal and insertion of sequences.

The reason for the almost non-existence of work for a comprehensive and efficient solution to incremental sequential mining is the complexity of this problem. In the following we will use examples to show why removal and insertion of sequences are hard.

The immediate intuition to handle the removal case would be simply decrease the support of each frequent sequence that has support from a removed sequence in the result. If the updated support falls below the threshold, remove that sequence from the result. In complete frequent sequence mining, this would generate correct result. However, doing so is quite expensive, since the number of complete frequent sequences would be exponential. Things become more complex in case of closed frequent sequence mining, that this approach couldn’t even guarantee the correctness of the result.

*Example.* Table 1 is a sample sequence database, referred as  $D$  in this paper when the context is clear. If *min\_sup* = 2 (taken as default in this paper), closed frequent sequences in  $D$  are in table 2. Now in the new database, if sequence 0 of  $D$  is removed, then simply updating of the result would yield the result in table 3. The result in table 3 is incorrect, since pattern:  $\langle (ab) \rangle: 2$  is no more closed. The correct closed frequent sequences are shown in table 4

Complexity for the insertion case becomes even worse than the removal case. The main challenge is that those originally infrequent sequences may now become frequent, of which we have no clue of from the original result. For example, if we are going to insert a new sequence,  $\langle (f) \rangle$ , into  $D$ , it will turn the original infrequent item  $(f)$  into a frequent item, since in original database  $D$ ,  $(f)$  already

SeqID	Sequence
0	$\langle (ab)(d) \rangle$
1	$\langle (ab)(c)(d) \rangle$
2	$\langle (ab)(ce)(f) \rangle$

Table 1: A Sample Sequence Database  $D$

Pattern	Support	Support List
$\langle (ab) \rangle$	3	0, 1, 2
$\langle (ab)(c) \rangle$	2	1, 2
$\langle (ab)(d) \rangle$	2	0, 1

Table 2: Closed frequent sequences for  $D$

has a support of 1. However, this information don’t appear either in the final mining result nor some intermediate data structure, e.g. prefix tree.

Rather than mining everything from scratch, IncSpan solves the insertion problem by buffering some infrequent sequences with relatively high probability to become frequent in the updated database, and thus reduces some search space in incremental mining. Despite the fact that IncSpan can only handle the removal case, it also suffers from other defects that are fatal in certain circumstances. Since IncSpan doesn’t maintain any tree-like data structure, it will be very expensive to perform closeness checking. It is necessary to design a more general and effective solution to the incremental mining problem.

**1.3 Our Contribution** In this paper, we propose CISpan, a Comprehensive Incremental mining algorithm of Closed Sequential Pattern. We attack this problem in a divide and conquer manner, separating insertion case and removal case apart from each other. The key idea behind CISpan is to build a tree-like data structure named *incremental lattice* to store all the frequent sequences appearing in the inserted sequences, while for removal case directly update the intermediate mining data structure, namely prefix lattice, of the original database. Both of these two operations are very cheap. For *incremental lattice*, the number of nodes is limited by involving only items within the inserted sequences. Handling the removed sequences is even cheaper since it will only be a side effect of recovering the original prefix lattice, which is a compact representation of all the frequent sequences within the database. Then we merge the original prefix lattice with the incremental lattice, apply closed checking algorithm to mine closed sequences, and finally output the frequent sequences. By disabling the closed checking step, we can also output the complete set of frequent sequences. The final merged lattice in CISpan is exactly the same as the one mining from scratch, which

Pattern	Support	Support List
$\langle (ab) \rangle$	2	1, 2
$\langle (ab)(c) \rangle$	2	1, 2

Table 3: Incorrect closed sequences for  $D$  after seq 0 is deleted by simply updating the result

Pattern	Support	Support List
$\langle (ab)(c) \rangle$	2	1, 2

Table 4: Correct closed sequences for  $D$  after seq 0 is deleted.

means the performance will not degrade by accumulatively applying CISpan on a series of database.

Our CISpan is a comprehensive algorithm for incremental sequential pattern mining. Our contributions are summarized as following:

- CISpan can handle all kinds of updates to the database, not limited only to insertion and appendage.
- CISpan can mine both closed and complete frequent sequences. We have both versions implemented.
- CISpan can effectively mine frequent sequences with all possible minimum support threshold, even as low as 2. Our evaluation on real database transformed from Linux Kernel source code is based on the minimum support of 2, in which case IncSpan simply becomes non-incremental.
- CISpan performs linearly to a wide range of various support threshold, sequence length, sequence number and modification rate.
- We provide an effective algorithm for software evolution study. Tools using sequential mining algorithms can now be extended with CISpan to analyze software evolution patterns. We also demonstrated the effect of applying CISpan into one of the code mining tools, CP-Miner, as a case study.

We compare CISpan’s complete frequent sequence mining version with IncSpan, since IncSpan cannot mine closed frequent sequences. For every test case which IncSpan can finish mining, CISpan could also successfully mine, with a better performance. For all these test cases, CISpan outperforms IncSpan by an average 3.4 times speedup.

We also test CISpan’s performance with original CloSpan on our real database from source code. IncSpan simply cannot handle this case at all. On average, we see 42 times speedup of incremental mining compared

to CloSpan [15] on the real source code database transformed from Linux kernel. We also evaluate CISpan’s performance on multiple increments of source code database, which shows that even for 20 versions’ difference, CISpan still performs 4.5 times faster than mining from scratch using CloSpan.

## 2 Preliminary Concept

Let  $IT = \{i_1, i_2, \dots, i_k\}$  be a set of all items. A subset of  $IT$  is called an *itemset*. A *sequence*  $s = \langle t_1, t_2, \dots, t_m \rangle$  ( $t_i \subseteq IT$ ) is an ordered list. The *length*,  $l(s)$ , is the total number of items in the sequence, i.e.,  $l(s) = \sum_{i=1}^m |t_i|$ . A sequence  $\alpha = \langle a_1, a_2, \dots, a_m \rangle$  is a *sub-sequence* of another sequence  $\beta = \langle b_1, b_2, \dots, b_n \rangle$ , denoted as  $\alpha \sqsubseteq \beta$ , if and only if  $\exists i_1, i_2, \dots, i_m$ , such that  $1 \leq i_1 < i_2 < \dots < i_m \leq n$  and  $a_1 \subseteq b_{i_1}, a_2 \subseteq b_{i_2}, \dots$ , and  $a_m \subseteq b_{i_m}$ .

A sequence database,  $D = \{s_1, s_2, \dots, s_n\}$ , is a set of sequences. The *support* of a sequence  $\alpha$  in  $D$  is the number of sequences in  $D$  which contain  $\alpha$ ,  $support(\alpha) = |\{s | s \in D \text{ and } \alpha \sqsubseteq s\}|$ . The *support list* of a sequence  $\alpha$  in  $D$  is a list of sequence IDs containing  $\alpha$ . So the length of *support list* for  $\alpha$  is the *support* of  $\alpha$ . Given a minimum support threshold,  $min\_sup$ , a sequence is *frequent* if its support is no less than  $min\_sup$ . The set of *frequent sequential patterns*,  $FS$ , includes all the frequent sequences. The set of *closed frequent sequential patterns*, is defined as follows,  $CS = \{\alpha | \alpha \in FS \text{ and } \nexists \beta \in FS \text{ such that } \alpha \sqsubseteq \beta \text{ and } support(\alpha) = support(\beta)\}$ . Since  $CS$  includes no sequence which has a super-sequence with the same support, we have  $CS \subseteq FS$ . The problem of *closed sequence mining* is to find  $CS$  above a minimum support threshold. Table 2 shows the closed frequent sequences for  $D$  in table 1.

The projected database of a sequence  $\alpha$  consists of all the suffix of sequences that contain  $\alpha$  (formal definition in CloSpan [15]). For example, the projected database for  $\langle (b) \rangle$  in  $D$  is  $D_{\langle (b) \rangle} = \{\langle (d) \rangle, \langle (c)(d) \rangle, \langle (ce)(f) \rangle\}$ ; and  $D_{\langle (a)(c) \rangle} = \{\langle (d) \rangle, \langle (e)(f) \rangle\}$ .

A *prefix tree*  $T$  is a tree that represents the set of frequent subsequences in a database. Each node  $p$  in  $T$  has a tag labelled with  $s$  or  $i$ .  $s$  means the node is a starting item in an itemset;  $i$  means the node is an intermediate item in an itemset. Figure 1 shows the prefix tree for  $D$ . The representation for each node follows the form:  $\langle item : support \rangle$ . Each circle in the figure represents a *children pointer* of each node, which points to a vector filled by pointers to all the children nodes.

A closer examination at Figure 1 would reveal the fact that the subtrees of the two nodes  $\langle b_s \rangle : 3$  and  $\langle b_i \rangle : 3$  are exactly the same. This is because the projected databases  $D_{\langle (b) \rangle}$  and  $D_{\langle (ab) \rangle}$  are exactly the same. It would be a waste if we produce and store two copies of the subtrees in this case. CloSpan [15] optimizes the prefix tree by making nodes with same projected database share the same children.

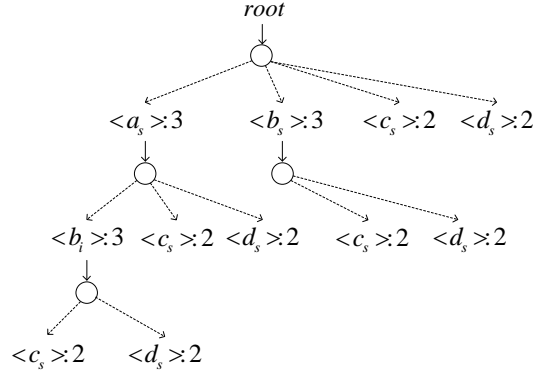


Figure 1: Prefix tree of  $D$ .

Figure 2 is the data structure in CloSpan, namely *prefix sequence lattice*. Subtree sharing of two nodes is simply done by pointing to the same *children pointer*. In this case, node  $\langle b_s \rangle : 3$  and  $\langle b_i \rangle : 3$  share the same subtree, thus we only need to produce and store the subtree once.

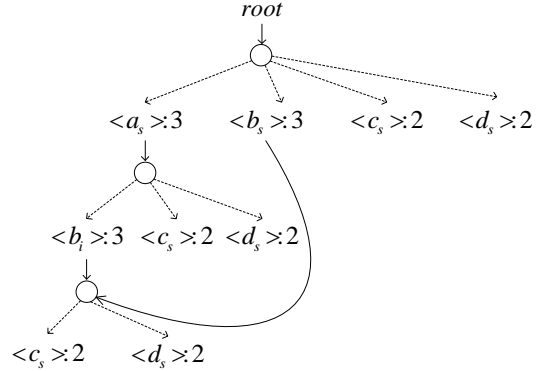


Figure 2: Prefix sequence lattice of  $D$ .

We define that a sequence  $\alpha$  *appears* in a database  $D$  if  $\alpha$  is a subsequence of one of  $D$ ’s sequences. Formally,  $\alpha$  *appears* in  $D$  if and only if  $\exists s$  such that  $s \in D$  and  $\alpha \sqsubseteq s$ , or simply  $support(\alpha) \geq 1$  in  $D$ .

## 3 CISpan Overview

Given the original database  $D$  and an updated database,  $D'$ , CISpan models all updates as removal of sequences from  $D$  and insertion of sequences to  $D'$ . For example, table 5 shows the updated database  $D'$  after modification of sequence 0 in the example database  $D$ . CISpan will model this update to a removal of sequence 0 in  $D$  and an insertion of sequence 0 in  $D'$ . Figure 4 illustrates the update model from  $D$  to  $D'$ . We use  $U$  to represent the unchanged sequence set in  $D$  and  $D'$ ,  $R$  for the removed sequence set in  $D$ , and  $I$  for the inserted sequence set in  $D'$ . All kinds of

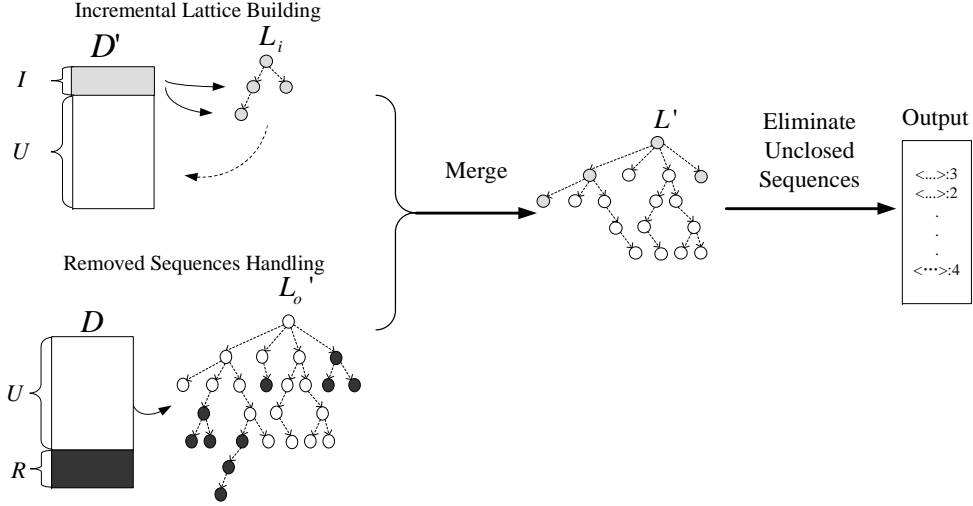


Figure 3: Overview of CISpan.  $L_i$  is the incremental lattice, and  $L'_o$  is the original prefix lattice after being updated for removed sequences.  $L'$  is the merged lattice. Arrows from the database to the lattices indicate where the nodes in the lattices come from. Each node in  $L_i$  corresponds to an item appears in  $I$ , while nodes in  $L'_o$  are items appear in  $U$ . Arrow from lattice  $L_i$  to database  $D'$  indicates that we calculate the support of sequences in  $L_i$  from  $D'$  as a whole, not only in  $I$ .

possible updates for  $D$  are covered in this simple model.

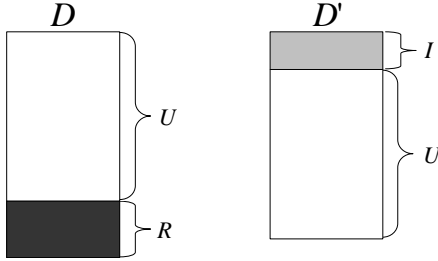


Figure 4: Database update model.

SeqID	Sequence
0	$\langle (a)(f) \rangle$
1	$\langle (ab)(c)(d) \rangle$
2	$\langle (ab)(ce)(f) \rangle$

Table 5: Sample Sequence Database  $D'$

While mining  $D$  from scratch using CloSpan, we store the intermediate prefix lattice  $L_o$ . Because  $L_o$  is an acyclic graph rather than a tree, simple recursive traversal is not enough to store the subtree sharing information. CISpan solves this by numbering each node with a unique node ID in the lattice, so for each node we can exactly identify every child node through the node ID.

When mining  $D'$  incrementally, CISpan algorithm follows a divide and conquer strategy. Figure 3 shows the

overview of CISpan. We handle the removal and insertion of sequences separately. For the insertion case, we build a small lattice called *incremental prefix lattice*,  $L_i$ , that contains only the frequent sequences *appearing* in  $I$ . For the removal case, we update the nodes correspondingly when retrieving the lattice  $L_o$ . We refer to this updated lattice  $L'_o$ . Then we merge updated  $L'_o$  with  $L_i$ , and form a lattice  $L'$  for  $D'$ . Our merging algorithm guarantees that this  $L'$  is exactly the same as the one if we mine  $D'$  from scratch using CloSpan. Finally, we output all the frequent sequences based on  $L'$ . To output only the closed frequent sequences, we apply the non-closed sequence elimination algorithm in CloSpan [15] to  $L'$  and output only the closed frequent sequence set.

We define  $IS$  as all the frequent sequences in  $D'$  that appear in  $I$ . More formally,  $IS = \{s | support(s) \text{ in } D' \geq min\_sup \text{ and } \exists s' \text{ such that } s' \in I \text{ and } s \sqsubseteq s'\}$ . The incremental lattice  $L_i$  contains all the sequences in  $IS$ . We also define  $US$  as sequences that are frequent in  $U$ . More formally,  $US = \{s | support(s) \text{ in } U \geq min\_sup\}$ .  $L'_o$  contains all the sequences in  $US$ . Lemma 3.1 forms the fundamental of our divide and conquer idea.

LEMMA 3.1.  $FS = IS \cup US$

*Proof.* Based on the definition,  $IS \subseteq FS$  and  $US \subseteq FS$ , so  $IS \cup US \subseteq FS$ . Assume a sequence  $\alpha \in FS$  that  $\alpha \notin IS \cup US$ . Then  $\alpha \notin IS$  and  $\alpha \notin US$ . Since  $\alpha \notin IS$ , then all of  $\alpha$ 's support must be from  $US$ ; and since  $\alpha \notin US$ , then  $\alpha$  must not be frequent in  $US$ , which means  $\alpha$  isn't frequent in  $D'$ . This is contradictory to the assumption. Thus we have:  $FS = IS \cup US$ .  $\square$

#### 4 Incremental Lattice Building

As shown in section 1, inserted sequences may introduce new frequent items. Following is another example to further reveal that the complexity of inserted sequences is not only limited to introducing new frequent *items*.

*Example.* Figure 5 shows a prefix tree for different databases  $D_1 = \{\langle (a)(b) \rangle, \langle (a)(c) \rangle, \langle (b)(c) \rangle\}$ ;  $D_2 = \{\langle (b)(a) \rangle, \langle (c)(a) \rangle, \langle (b)(c) \rangle\}$ ;  $D_3 = \{\langle (b)(a) \rangle, \langle (a)(c) \rangle, \langle (c)(b) \rangle\}$ , etc. All these databases have the same prefix tree, and they share the same frequent sequences:  $FS = \{\langle (a) \rangle: 2, \langle (b) \rangle: 2, \langle (c) \rangle: 2\}$ . Now if we insert a new sequence,  $\alpha = \langle (a)(b)(c) \rangle$ , to each of these databases, different databases will yield different frequent sequences. Note in  $\alpha$ , all of the items were frequent in original databases. Now given that all items in inserted sequences were frequent in original database, and also given the original mining data structure, the prefix tree in Figure 5, we still can't tell the final result without looking into the whole database.

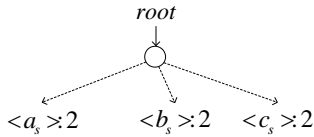


Figure 5: Prefix tree for various possible databases.

The fundamental reason for the complexity of insertion is that by only buffering information about frequent sequences, inserted sequences may always turn some original un-buffered sequences into frequent ones. In other words, by buffering only frequent sequences (or their tree-like representation, e.g., prefix lattice), we cannot avoid scanning the whole database in order to mine the final frequent sequences.

IncSpan solves the insertion problem by buffering additional semi-frequent sequences with relatively high probability to become frequent in the updated database, and thus reduces some search space in incremental mining. Even though, it still needs to scan the whole database for new items. Also when update percentage becomes relatively high, some originally infrequent sequences are likely to become frequent, in which case that buffering semi-frequent sequences cannot help much.

The bottleneck of prefix tree based algorithms [12, 15, 4] is multiple scans of the whole database. The number of scans of the database is proportional to the number of nodes in the prefix tree. After identifying this, our design philosophy is to build a concise prefix lattice with sufficient information we need to handle the inserted case. In practice, this *incremental prefix lattice*,  $L_i$ , contains only the frequent sequences in  $D'$  that appear in  $I$ . We name this process *Cross Module Mining*.

In cross module mining, the  $L_i$  is built in the CloSpan manner. The difference with CloSpan is that every sequence in  $L_i$  has to be not only frequent, but also has to satisfy the *cross module mining property*.

**PROPERTY 4.1.** *Cross Module Mining Property:* If sequence  $\alpha$  appears in  $I$ , then we say  $\alpha$  satisfies cross module mining property.

The reason we name this step *cross module mining* is that given  $D'$  is formed by 2 modules,  $I$  and  $U$ , we only mine the sequences appear in  $I$ , however we have to *cross the module* to count its support from  $U$  and  $I$  together.

Algorithm 1 illustrates the framework of cross module mining. We scan the database once, and find all the frequent items that appear in  $I$ , and the incremental lattice only contains these items. Note we still count the support of an item in the complete  $D'$ , not only in  $I$ . *IncCloSpan*, as shown in algorithm 2, is the recursive function that builds the incremental lattice,  $L_i$ . It follows the same behavior as original *CloSpan*, with only difference that in addition to checking the next level item is frequent or not, it further checks whether the corresponding sequence appears in  $I$  or not (line 7, first condition). This is simply done by checking whether the ID of each sequence containing  $i$  is within  $I$  or not. Every sequence in  $L_i$  must be frequent and must appear in  $I$ . Figure 6 is the incremental lattice for the inserted sequence in  $D'$ . In this case,  $I = \{\langle (a)(f) \rangle\}$ .

---

#### Algorithm 1 Cross Module Mining ( $D', min\_sup, L_i$ )

---

**Require:** A database  $D'$ , partitioned by  $I$  and  $U$ ; and  $min\_sup$ .

**Ensure:** The incremental prefix lattice  $L_i$

- 1: Scan  $D'$  once, find every item  $i$  such that
    - at least one sequence containing  $i$  in  $D'$  appears in  $I$ , and
    - $i$  is frequent in  $D'$ .
  - 2: **for** every such item  $i$  **do**
  - 3:    $s \leftarrow \langle i \rangle$ ;
  - 4:   IncCloSpan ( $s, D'_s, min\_sup, L_i$ )
  - 5: **end for**
- 

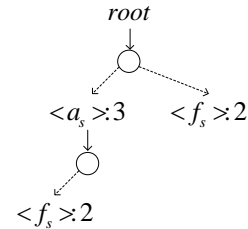


Figure 6: Incremental Prefix lattice for  $D'$ . Only sequence  $\langle (a)(f) \rangle$  is inserted.

---

**Algorithm 2** IncCloSpan( $s, D'_s, min\_sup, L_i$ )

---

**Require:** A sequence  $s$ , a projected DB  $D'_s$ , and  $min\_sup$ **Ensure:** The incremental prefix lattice  $L_i$ .

- 1: Insert  $s$  into  $L_i$ ;
  - 2: Check whether a discovered sequence  $s'$  exists s.t.  $s$  and  $s'$  share the same projected database;
  - 3: **if** such  $s'$  exists **then**
  - 4:   Share the *children pointer* of  $s'$  with  $s$  in  $L_i$ ;
  - 5:   **return**;
  - 6: **end if**
  - 7: Scan  $D'_s$  once, find every item  $i$  such that
    - at least one sequence containing  $i$  in  $D'_s$  appears in  $I$ , and
    - $i$  is frequent in  $D'_s$ , and
    - $s$  can be extended to  $s \diamond_i i$  or  $s \diamond_s i$ ;
  - 8: **if** no valid  $i$  available **then**
  - 9:   **return**;
  - 10: **end if**
  - 11: **for** each valid  $i$  **do**
  - 12:   Call IncCloSpan( $s \diamond_i i, D'_{s \diamond_i i}, min\_sup, L_i$ );
  - 13: **end for**
  - 14: **for** each valid  $i$  **do**
  - 15:   Call IncCloSpan( $s \diamond_s i, D'_{s \diamond_s i}, min\_sup, L_i$ );
  - 16: **end for**
  - 17: **return**;
- 

LEMMA 4.1. Every sequence in  $IS$  must also appear in  $L_i$ , and its information (support, support list, etc) is the same as in case of mining  $D'$  from scratch.

Lemma 4.1 shows the nice property of  $L_i$ . Not only it completely solves all the complexities for the insertion case, it also retains the final prefix lattice structure as well as all information of these nodes appearing in  $I$ .

We claim that it is very cheap to build  $L_i$  given that the modification ratio is not large. Since  $I$  is very small compared to the whole database, there are only a small number of items within  $I$ , so our incremental prefix lattice is also small. So we only have a small number of scans of the database  $D'$ . For example, for the real database transformed from Linux kernel 2.6.20.1 source code, there will be only 206 nodes in the incremental prefix lattice (based on the result from 2.6.20), compared to 697430 nodes in original prefix lattice.

## 5 Handle the removed sequences

Removed sequences are simply handled as a side effect when we retrieve the original lattice,  $L_o$ . When we retrieve each node in  $L_o$ , we check each sequence ID in the support list of this node. If we find  $k$  of them are in  $R$ , then we decrease the support of this node by  $k$ . After decreasing support, if we find that the support of a node falls below  $min\_sup$ , then we simply remove this node and all its sub-lattice from  $L_o$ . After we are done with retrieval and removed sequences update,

we have a lattice,  $L'_o$ , containing all the frequent sequences in  $U$ . Figure 7 is the  $L'_o$  when we mine  $D'$  based on  $D$ .

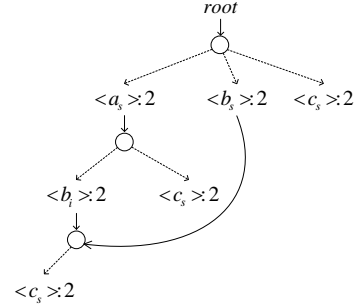


Figure 7: Retrieved Prefix lattice of  $D$ . Sequence  $\langle (ab)(d) \rangle$  is removed, causing support for  $\langle (a_s) \rangle$ ,  $\langle (b_s) \rangle$  and  $\langle (b_i) \rangle$  to decrease. Also the 3  $\langle (d_s) \rangle$  nodes are removed.

LEMMA 5.1. Every sequence in  $US$  must also appear in  $L'_o$ .

## 6 Merging the two lattices

With  $L_i$  and  $L'_o$  built up, we now can claim the following theorem based on lemmas 3.1, 4.1 and 5.1.

THEOREM 6.1. *Completeness:* Every sequence in  $FS$  is contained in  $L_i$  or  $L'_o$ , or both.

This theorem guarantees that we have the complete information about the  $FS$ . However, there may be competing information between  $L_i$  and  $L'_o$ , so the goal of the merging algorithm would be to eliminate all competing/redundant information, and finally generate a lattice,  $L'$ , that is the same as one mining from scratch.

The principle for the merging algorithm is based on lemma 4.1, that whenever there is competing information, we should follow the information in  $L_i$ . Since the majority of nodes will be in  $L'_o$ , so we should merge  $L_i$  into  $L'_o$  while walking through  $L_i$  and  $L'_o$  simultaneously.

In order to preserve the children-sharing property of the prefix lattice, we need to mark each node whether we have already visited it or not. When we reach two nodes with different values of this mark, it suggests we need to merge or split the nodes in  $L'_o$  in some way.

We start by calling Merge( $oRoot, iRoot$ ), and the algorithm will visit all the nodes in  $L_i$  and the corresponding ones in  $L'_o$ . We first replace the support and support list in  $oNode$  with the ones in  $iNode$  (line 1), ensuring that the information is updated according to  $L_i$ . Then for every child of  $iNode$ , namely  $iChild$ , we find the corresponding node in  $oNode$ , namely  $oChild$ . If there is no such child in  $oNode$ , we add a new one (line 10 to 12).

Based on the visit history of the two children (visited value), there could be totally four cases. Different strategies

---

**Algorithm 3** Merge (oNode, iNode)

---

**Require:** Two nodes in  $L'_o$  (oNode) and  $L_i$  (iNode).**Ensure:**  $L_i$  is merged into  $L'_o$ .

```
1: copy iNode's support and support list to oNode;
2: oNode.visited  $\Leftarrow$  true;
3: iNode.visited  $\Leftarrow$  true;
4: if iNode has no children then
5:   return;
6: end if
7: for iChild  $\Leftarrow$  every child of iNode do
8:   oChild  $\Leftarrow$  find_child_from(oNode, iChild);
9:   if oChild = EMPTY then
10:    oChild  $\Leftarrow$  new node same as iChild;
11:    add_child (oNode, iChild);
12:    oChild.visited  $\Leftarrow$  false;
13:   end if
14:   if (oChild.visited = false) AND (iChild.visited = false)
15:   then
16:     iChild.last_visit  $\Leftarrow$  oNode;
17:     MERGE(oChild, iChild);
18:   else if (oChild.visited = false) AND (iChild.visited = true)
19:   then
20:     oNode.Children  $\Leftarrow$  iChild.last_visit.Children;
21:     iChild.last_visit  $\Leftarrow$  oNode;
22:     return;
23:   else if (oChild.visited = true) AND (iChild.visited = false)
24:   then
25:     Make a new copy of oNode.Children and all the children
26:     nodes;
27:     oNode.Children  $\Leftarrow$  the new copy of previous step;
28:     oChild  $\Leftarrow$  find_child_from(oNode, iChild);
29:     iChild.last_visit  $\Leftarrow$  oNode;
30:     MERGE(oChild, iChild);
31:   else
32:     if oChild is not a child of iChild.last_visit then
33:       oNode.Children  $\Leftarrow$  iChild.last_visit.Children;
34:     end if
35:     iChild.last_visit  $\Leftarrow$  oNode;
36:     return;
37:   end if
38: end for
```

---

are applied to different cases.

**1. Case 1: Both unvisited**

Line 15 - 16 in algorithm 3 corresponds to this most trivial case. All we need to do is to recursively call Merge again.

**2. Case 2: Merge** Figure 8 illustrates the case iChild is visited while oChild is not. Line 18 - 20 in algorithm 3 corresponds to this case. This indicates we need to merge nodes in  $L'_o$ . In this case, we need the information about which node to merge with. This information is kept in a pointer last\_visited within every node in  $L'_i$ . Every time an iChild is vis-

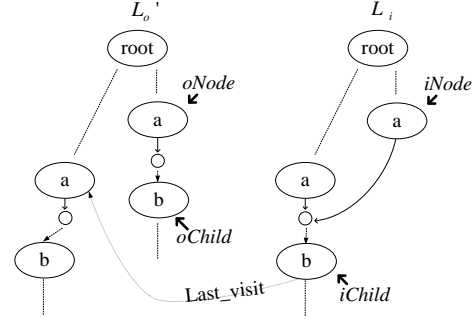


Figure 8: Example of the merge case.

ited, its last\_visited points to the current oNode. Thus we could do merging by simply modifying the Children pointer of oNode to point to the one in last\_visit (Line 18). After the merging of nodes, we don't need to further continue the recursion on this branch, since it has already been visited once previously.

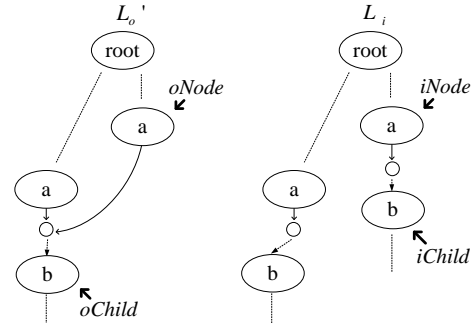
**3. Case 3: Split**

Figure 9: Example of the split case.

Figure 9 and line 22 - 26 in algorithm 3 correspond to the case when oChild is visited yet iChild is not. This indicates we need to split nodes in  $L'_o$ . In this case, we need to make a new copy of the Children pointer of oNode, and all the children nodes in it. Note we have to make a copy of Children from oNode rather than iNode, because the former may contain more children. Then we continue the recursion.

**4. Case 4: Both visited**

When both iChild and oChild are visited (line 28 - 32), it is not safe if we directly stop searching this branch. There may be a tricky case that the graph structure of  $L_i$  and  $L'_o$  are not corresponding to each other. Figure 10 illustrates one example. In figure 10, both iChild and oChild are visited. However, the structures of the two lattices are not the same. We

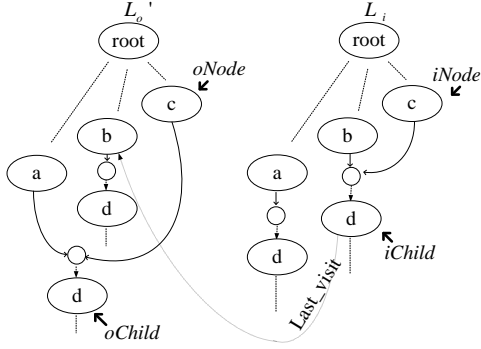


Figure 10: Example to show a case that both node are visited in  $L'_o$ .

can detect this by testing whether `oChild` is the child of `last_visit` (Line 28). If not, then we need to restructure  $L'_o$ , simply by modifying the `Children` pointer of `oNode` (Line 29).

The merged prefix lattice,  $L'$ , now contains the complete set of frequent sequences of  $D'$ . Figure 11 shows the merged prefix lattice of  $L'_i$  shown in Figure 6 and  $L'_o$  shown in Figure 7. Assume the prefix lattice we get in mining  $D'$  from scratch is named  $L'_{scratch}$ , we claim that  $L'$  is exactly the same as  $L'_{scratch}$ , which means after the merge,  $L'$  contains complete and non-redundant information.

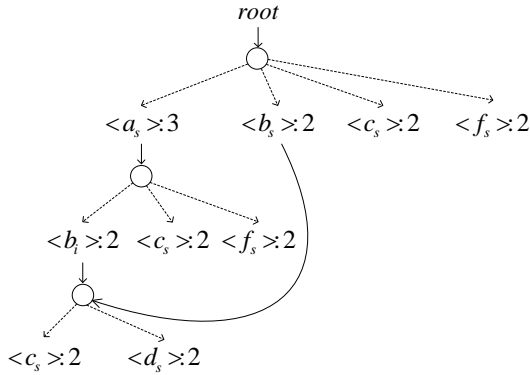


Figure 11: Merged  $L'$  for  $D'$ .

**THEOREM 6.2. Soundness and Completeness:**

$$L' = L'_{scratch}$$

*Proof.* Recall the properties of  $L'_{scratch}$ :

1. Contains all the frequent sequences in  $D'$ .
2. If two sequences have the same projection database, then they share the same sub-lattice.

We have to prove that  $L'$  also satisfies these 2 properties. The first property is simply proved in theorem 6.1. If two sequences,  $\alpha$  and  $\beta$  have the same projection database, then they must have the same support list in  $D'$ . Then this  $\alpha$  (or  $\beta$ ) could either *appear* in  $I$  or not. If so, they will share the same sub-lattice in  $L'_i$ , and by construction we retain this structure in  $L'$ . If not, then they share the same sub-lattice in the original lattice,  $L'_o$ , and we don't even visit these nodes during merging, so the structure is also retained.  $\square$

## 7 Evaluation

We evaluate the performance of CISpan from two aspects. First, we compare the performance of CISpan with IncSpan [4]. Because IncSpan can only handle *insertion* and *append* operation to database, so we only generate test cases that IncSpan can handle. On every test case which IncSpan can handle, our experimental result suggests CISpan can also handle that, with better performance. The performance gap between CISpan and IncSpan gets larger as performance degrades. Among all the test cases we used, CISpan outperformed IncSpan by an average of 3.4 times and maximum of 15 times.

Then we compare CISpan integrated in CP-Miner to CloSpan on real databases transformed from 20 consecutive versions of Linux Kernel source code. IncSpan cannot mine this kind of databases at all since it cannot handle removal of sequences, and also it becomes non-incremental when minimum support threshold is 2. On average, CISpan is 42 times (with a maximum 230 times) faster than mining from scratch using CloSpan.

Our test bed was equipped with Dual core Intel Pentium D 3.0GHz processor, 1024KBytes L2 cache and 2 GBytes main memory. CISpan, IncSpan and CloSpan algorithms are written in C++ and compiled with -O3 optimization.

**7.1 Performance Comparison with IncSpan** All the experiments in this section is based on synthetic dataset generated by IBM data generator tool. The synthetic dataset generator can be retrieved from an IBM website, <http://www.almaden.ibm.com/cs/quest>. The parameters we used in the experiment are as follows [2]:

- D - Number of sequences (in 1000s)
- C - Average itemset per sequence
- T - Average number of items in item set
- N - Number of different items (in 1000s)

In order to use IncSpan, only two models of updates, inserting new sequences and appending items to sequences are used. We first generate the dataset using the default parameters suggested by the provider to the tool, D100C10T2.5N10. Then we examine the effect of changing each parameter on

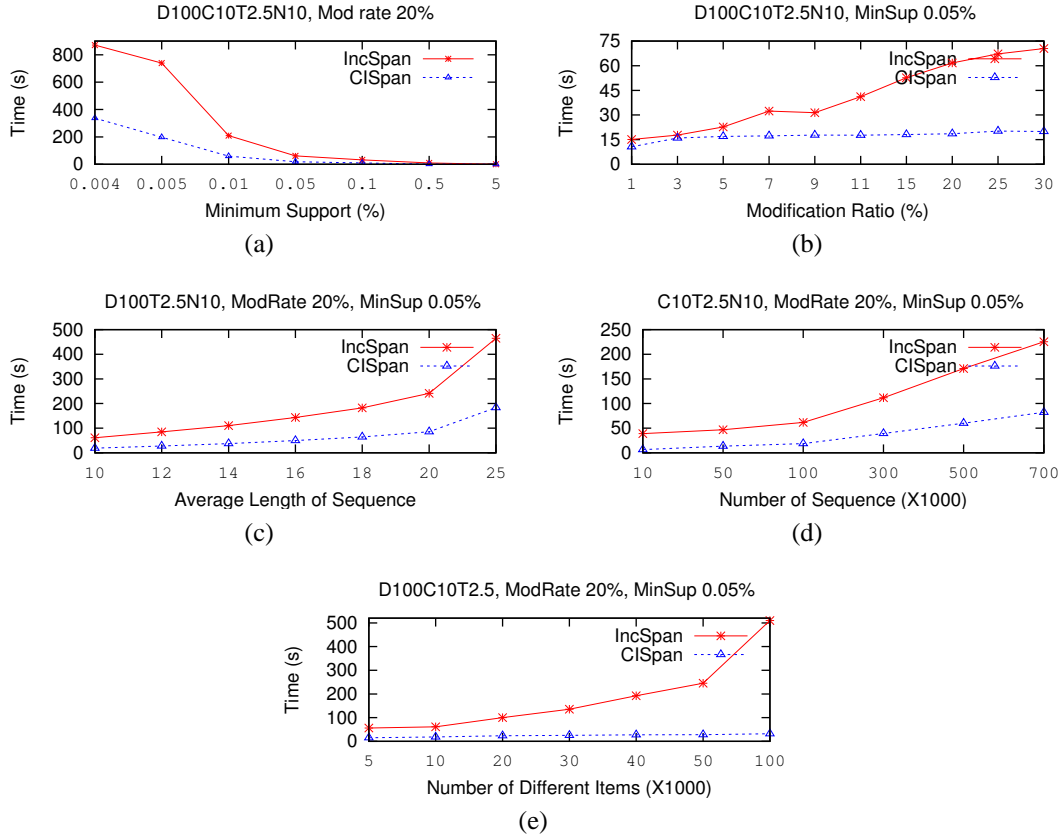


Figure 12: Performance results

the performances of CISpan and IncSpan. We set the semi-frequent buffer ratio in IncSpan as 0.8, as suggested by [4]. We also tested the effect of modification rate and minimum support rate in evaluation. Modification rate stands for percentage of updated sequence from original data set.

Figure 12(a) shows the performance of both CISpan and IncSpan when minimum support varies. We used default dataset D100C10T2.5N10 with modification rate 20%. When minimum support gets larger, both algorithms get better performance.

Figure 12(b) shows effect of modification rate. Modification rate is the percentage of updated sequences. For example, if data set has 100,000 sequences (D100) and 10,000 of them were updated, the modification rate is 10%. IncSpan’s performance is much more sensitive to modification rate change as compared to CISpan.

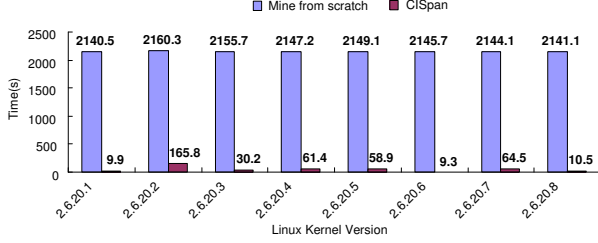
Figure 12(c) shows how each algorithm is affected by length of sequence (parameter  $C \times T$ ). For all the test cases where the average length varied from 10 to 25, CISpan outperforms IncSpan and the performance gap gets larger and larger.

In Figure 12(d), we evaluate both CISpan and IncSpan on various total number of sequences (parameter D).

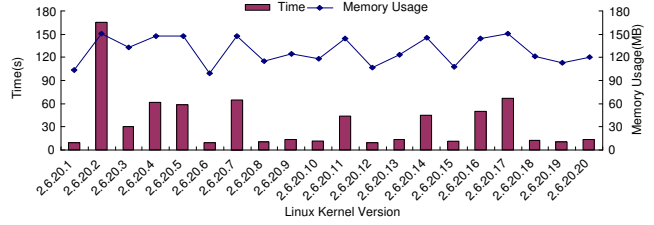
Figure 12(e) shows performance of each algorithm with various numbers of unique items ( $N$ ). For example, when  $N=10$ , data set will contains 10,000 unique items. When  $N=100$ , CISpan is 16 times faster than IncSpan. From the result we can see CISpan is almost not affected at all by the number of unique items, while the performance of IncSpan degrades as the number gets larger.

**7.2 Linux Kernel Mining** In this section, we use the real database transformed from Linux Kernel source code to evaluate CISpan. A recent major version of Linux Kernel, version 2.6.20.x is chosen, that contain totally 21 different Linux kernel code, from 2.6.20, 2.6.20.1, 2.6.20.2, ... to 2.6.20.20. We integrated CISpan algorithm within CP-Miner, with modifications to other steps to mine the source code incrementally. We first mine the 2.6.20 version from scratch using CP-Miner. After that, we apply incremental mining to later 20 versions. We first mine them in an iterative manner, which means incremental mining is based on the immediate previous version’s result. Then we test CISpan in an accumulative manner, that is, we mine every version based on the result of 2.6.20.

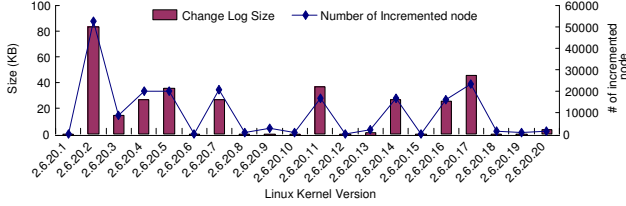
In figure 13(a), we first compare incremental mining



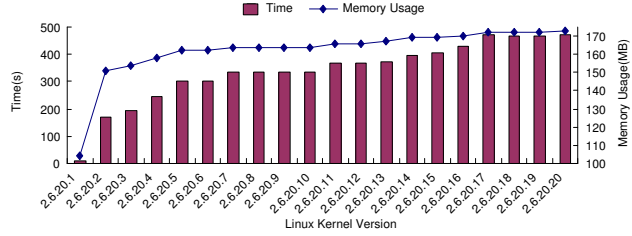
(a) Mining from scratch VS CISpan



(b) CISpan mining based on previous version



(c) Log size and # of incremented nodes for each version



(d) CISpan mining based on kernel 2.6.20

Figure 13: Incremental Mining of Linux Kernel 2.6.20.1 - 2.6.20.20

performance with mining from scratch. Since it takes around 40 minutes each time we mine the kernel source code from scratch, we only compared 8 versions (2.6.20.1 to 2.6.20.8) of kernel code mining from scratch to incremental mining. On average, CISpan outperforms CloSpan by 42 times.

We then incrementally mine all 20 versions one after another iteratively, e.g., mining 2.6.20.1 based on 2.6.20’s result, 2.6.20.2 based on 2.6.20.1’s result, etc. Figure 13(b) shows the time consumption and memory usage. On average, it takes 35 sec and 128.25 MBytes memory for each pass of incremental mining. Figure 13(c) shows the size of change log files and the size of incremental lattice,  $L_i$ , in number of nodes. Change log is a log or history of changes made to last version. Generally, big change log file suggests large amount of changes of source code from previous version, and correspondingly in CISpan it suggests large  $L_i$ . On average, the prefix lattice we get mining Linux Kernel code from scratch contains 701,339 number of nodes. This result shows in real life database, the performance of CISpan is sensitive to the change ratio of database. For example, incremental mining time for version 2.6.20.2 is significantly greater than other versions, due to the largest change log file size of that version.

Next, we incrementally mine the 20 kernel versions in an accumulative manner, that is, each time it is based on 2.6.20’s mining result. Figure 13(d) shows time and memory usage. We can see from the result that the performance increases steadily with the accumulative changes of versions. In the end, when mining the last version, 2.6.20.20, CISpan takes 470 seconds and 173 MB of memory, which is still 4.5 times faster than mining from scratch.

## 8 Related Work

Recently, several algorithms have been proposed for incremental mining and maintenance of sequential patterns, with two major approaches: (1) levelwise mining with candidate generate-and-test; and (2) incremental mining by keeping track of additional information.

[10] and [17] are two studies belonging to the first category. Both methods perform incremental mining with a candidate generate-and-test approach – the size- $(k + 1)$  candidates are generated from the size- $k$  frequent sequences. Although the number of database scans could be potentially reduced with some careful handling based on some mathematical deductions, this approach still involves multiple scans of the whole database, as well as the generation of a large number of candidates.

[3], [11] and [4] belong to the second category by keeping track of additional information for incremental mining. Chang et al. [3] proposed an incremental approach to buffer the CSTree of the old database. This approach is also capable of mining the closed frequent sequences.

Parthasarathy et al. [11] proposed ISM which is based on SPADE, by maintaining a sequence lattice of an old database. The sequence lattice includes the frequent sequences and a negative border which includes sequences which are infrequent but their subsequences are frequent.

[4] proposed IncSpan which is based on PrefixSpan by buffering a set of semi-frequent sequences. A semi-frequent sequence is an infrequent sequence but its frequency is no smaller than  $\mu * min\_sup$  where  $\mu \in [0, 1]$ . When a sequence database grows, the semi-frequent sequences have a higher probability to become frequent. Therefore, by buffering those patterns, the number of database scan and projection

operations could be effectively reduced.

When the existing methods are applied to the software source mining, there are three main limitations: (1) Cannot handle the removal of sequences; (2) Cannot handle a low minimum support as 2 which is meaningful in applications such as copy-pasted code detection; and (3) output too many frequent sequences which would slow down the incremental mining process. Our proposed CISpan algorithm could effectively handle these three challenges. It effectively reduces the number of mining results by generating closed sequential patterns in the incremental mining process.

## 9 Conclusions

Based on a real world problem, in this paper we proposed a comprehensive incremental mining algorithm for sequential database, that can be effectively applied to our software mining tool. CISpan could handle various mining requirements in real world database evolution, including general update model, closed sequence mining, low minimum support, etc. It outperforms the previously proposed incremental mining algorithm *IncSpan* by a wide margin. Integrated within our code mining tool, we show CISpan also outperforms the non-incremental method CloSpan by an average of 42 times on real Linux Kernel source code. Armed with this effective incremental mining algorithm, our immediate goal is to apply CP-Miner to investigate software evolution patterns, which is very significant for software engineering researchers. In the future, we will also apply CISpan into more software mining tools as well as utilizing these tools to mine software evolution patterns.

## 10 Acknowledgements

We thank the anonymous reviewers for useful feedback, the Opera group for useful discussions and paper proofreading. This research is supported by NSF CCF-0325603 grant, NSF CNS-0615372 grant, NSF CNS-0347854 (career award), and Motorola gift grants.

## References

- [1] M. Acharya, T. Xie, J. Pei, and J. Xu. Mining api patterns as partial orders from source code: from usage scenarios to specifications. In *ESEC-FSE '07*, pages 25–34, New York, NY, USA, 2007. ACM Press.
- [2] R. Agrawal and R. Srikant. Mining sequential patterns. In P. S. Yu and A. S. P. Chen, editors, *Eleventh International Conference on Data Engineering*, pages 3–14, Taipei, Taiwan, 1995. IEEE Computer Society Press.
- [3] L. Chang, D. Yang, T. Wang, and S. Tang. Imcs: Incremental mining of closed sequential patterns. In G. Dong, X. Lin, W. Wang, Y. Yang, and J. X. Yu, editors, *APWeb/WAIM*, volume 4505 of *Lecture Notes in Computer Science*, pages 50–61. Springer, 2007.
- [4] H. Cheng, X. Yan, and J. Han. Incspan: Incremental mining of sequential patterns in large database. In *Proc. 2004 Int. Conf. on Knowledge Discovery and Data Mining (KDD'04)*, 2004.
- [5] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: A tool for finding copy-paste and related bugs in operating system code. In *Proceedings of the Sixth Symposium on Operating System Design and Implementation (OSDI'04)*, pages 289–302, 2004.
- [6] Z. Li and Y. Zhou. PR-Miner: Automatically extracting implicit programming rules and detecting violations in large software code. In *13th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE'05)*, Sept 2005.
- [7] V. Livshits and T. Zimmermann. Dynamine: Finding common error patterns by mining software revision histories, 2005.
- [8] D. Lo, S.-C. Khoo, and C. Liu. Efficient mining of iterative patterns for software specification discovery. In *KDD '07: Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 460–469, New York, NY, USA, 2007. ACM Press.
- [9] S. Lu, S. Park, C. Hu, X. Ma, W. Jiang, Z. Li, R. A. Popa, and Y. Zhou. Muvi: Automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP07)*, October 2007.
- [10] F. Masseglia, P. Poncelet, and M. Teisseire. Incremental mining of sequential patterns in large databases. In *Data and Knowledge (DKE) Journal*, 2003.
- [11] S. Parthasarathy, M. J. Zaki, M. Ogihara, and S. Dwarkadas. Incremental and interactive sequence mining. In *CIKM*, pages 251–258, 1999.
- [12] 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. pages 215–226.
- [13] R. Srikant and R. Agrawal. Mining sequential patterns: Generalizations and performance improvements. In P. M. G. Apers, M. Bouzeghoub, and G. Gardarin, editors, *Proc. 5th Int. Conf. Extending Database Technology, EDBT*, volume 1057, pages 3–17. Springer-Verlag, 25–29 1996.
- [14] T. Xie and J. Pei. Mapo: mining api usages from open source repositories. In *MSR '06: Proceedings of the 2006 international workshop on Mining software repositories*, pages 54–57, New York, NY, USA, 2006. ACM Press.
- [15] X. Yan, J. Han, and R. Afshar. Clospan: Mining closed sequential patterns in large datasets, 2003.
- [16] M. J. Zaki. SPADE: An efficient algorithm for mining frequent sequences. *Machine Learning*, 42(1/2):31–60, 2001.
- [17] M. Zhang, B. Kao, D. W.-L. Cheung, and C. L. Yip. Efficient algorithms for incremental update of frequent sequences. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 186–197, 2002.