Università di Pisa

Data Mining and Machine Learning

Bioinspired computational methods

Biological data mining

Sequential Pattern Mining

Francesco Marcelloni

Department of Information Engineering
University of Pisa
ITALY

Some slides belong to the collection

Jiawei Han, Micheline Kamber, and Jian Pei
University of Illinois at Urbana-Champaign
Simon Fraser University

Introduction

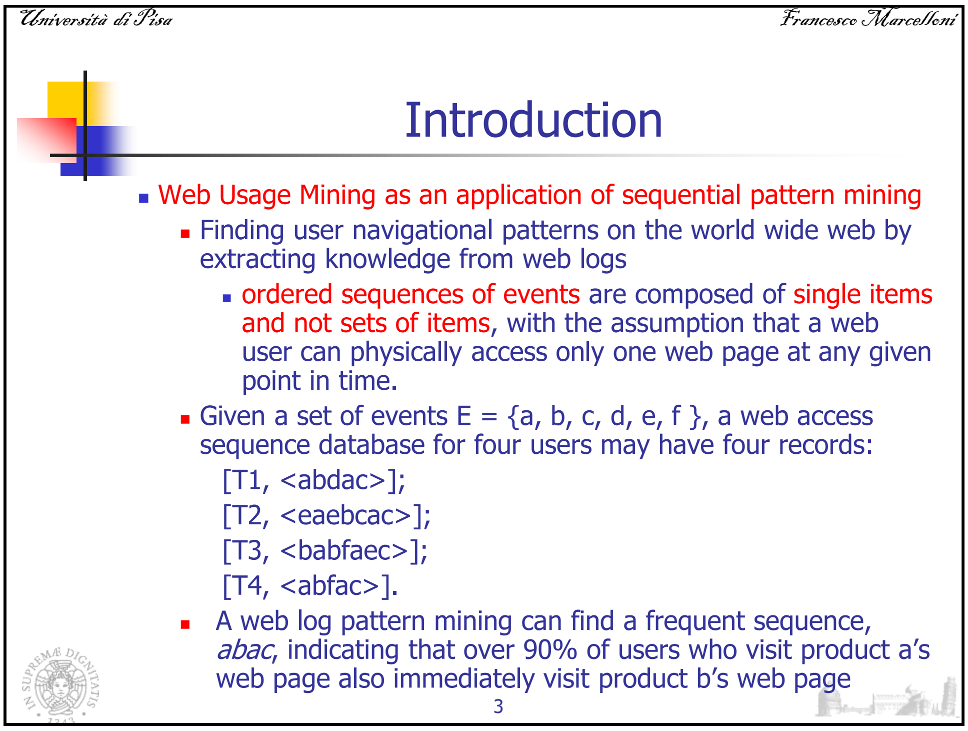
- Sequential pattern mining discovers frequent subsequences as patterns in a sequence database.
- A sequence database stores a number of records, where all records are sequences of ordered events, with or without concrete notions of time.
  - An example sequence database is retail customer transactions or purchase sequences in a grocery store showing, for each customer, the collection of store items they purchased every week for one month.
  - Records are stored as follows:

  [Transaction/Customer ID, <Ordered Sequence Events>]

  Examples:

  [T1, <(bread, milk), (bread, milk, sugar), (milk), (tea, sugar)>];

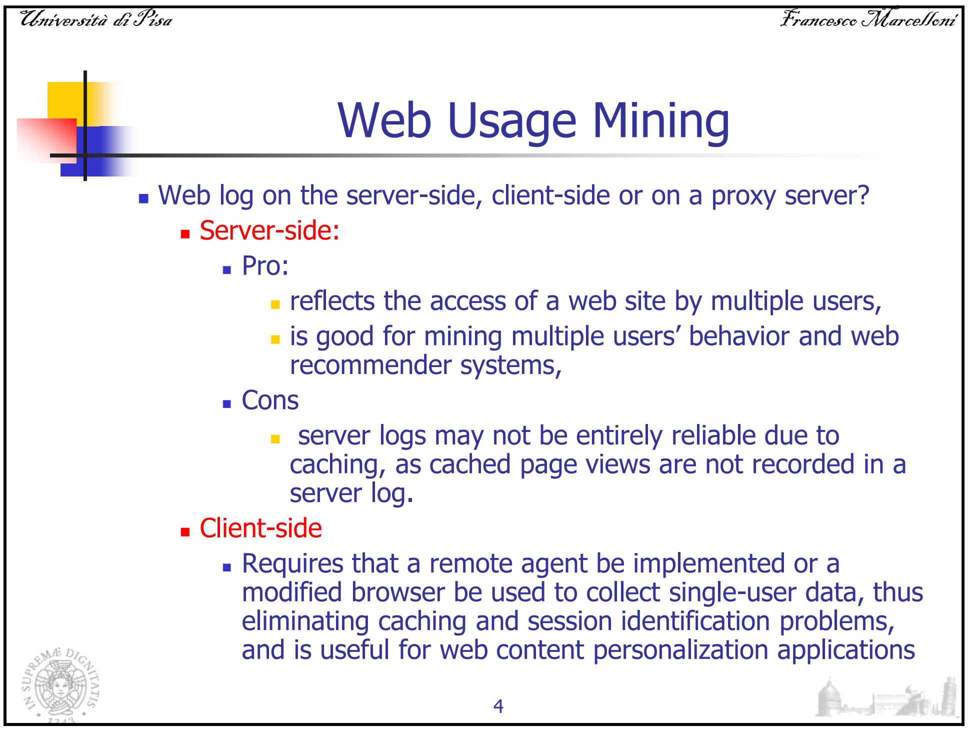
  [T2, <(bread), (sugar, tea)>]



Introduction

- Web Usage Mining as an application of sequential pattern mining
  - Finding user navigational patterns on the world wide web by extracting knowledge from web logs
    - ordered sequences of events are composed of single items and not sets of items, with the assumption that a web user can physically access only one web page at any given point in time.
  - Given a set of events E = {a, b, c, d, e, f }, a web access sequence database for four users may have four records:

    [T1, <abdac>];
    [T2, <eaebcac>];
    [T3, <babfaec>];
    [T4, <abfac>].
  - A web log pattern mining can find a frequent sequence, *abac*, indicating that over 90% of users who visit product a's web page also immediately visit product b's web page



Web Usage Mining

- Web log on the server-side, client-side or on a proxy server?
  - Server-side:
    - Pro:
      - reflects the access of a web site by multiple users,
      - is good for mining multiple users' behavior and web recommender systems,
    - Cons
      - server logs may not be entirely reliable due to caching, as cached page views are not recorded in a server log.
  - Client-side
    - Requires that a remote agent be implemented or a modified browser be used to collect single-user data, thus eliminating caching and session identification problems, and is useful for web content personalization applications

Web Usage Mining

- Web log on the server-side, client-side or on a proxy server?
  - Proxy server:
    - reveal the actual HTTP requests from multiple clients to multiple web servers, thus characterizing the browsing behavior of a group of anonymous users sharing a common server

Web Usage Mining

- Data Format
  - 137.207.76.120 - [30/Aug/2009:12:03:24 -0500] "GET /jdk1.3/docs/relnotes/deprecatedlist.html HTTP/1.0" 200 2781.

Anonymous user

- Other techniques such as cookie and sniffer may be needed.
- In most cases, researchers assume that user web visit information is completely recorded in the web server log, which is preprocessed to obtain the transaction database to be mined for sequences.

Sequential pattern mining

- An itemset is a set drawn from items in I, and denoted $(i_1, i_2, . . . , i_k)$, where i_j is an item or event.

- Problem definition
  - **Given**
    - A set of sequential records (called sequences) representing a sequential database D
    - A minimum support threshold called min_sup
    - A set of *k* unique items or events $I=\{i_1,i_2,,\ldots i_k\}$

  - **Find**
    - The set of all frequent sequences S in the given sequence database D of items I at the given min_sup.

Lexicographic order

- A sequence S is denoted as a sequence of elements $<e_1e_2e_3. . . e_q>$, where the sequence element e_j is an itemset (e.g., (be) in <a(be)c(ad)>) that might contain only one item (which is also referred to as 1-itemset).
- A sequence element is a lexicographically ordered list of items
  - Assume an itemset t of distinct items $t = \{i_1, i_2,. . . , i_k\}$, and another itemset of distinct items also $t' = \{ j_1, j_2,. . . , j_l\}$, where $i_1 \leq i_2 \leq \cdots \leq i_k$ and $j_1 \leq j_2 \leq \cdots \leq j_l$, such that ≤ indicates “occurs before” relationship. Then, for itemsets, t < t′ (t is lexicographically less than t) iff either of the following is true:

  1) for some integer h, $0 \leq h \leq \min\{k, l\}$, we have $i_r = j_r$ for $r< h$, and $i_h < j_h$, or
  2) $k < l$, and $i_1 = j_1, i_2 = j_2,\ldots, i_k = j_k$.

  Example (1): (abc) < (abec) and (af) < (bf)
  Example (2): (ab) < (abc)

Sequences

- A sequence with *k* elements is called a *k*-sequence. An item can occur only once in an itemset, but it can occur several times in different itemsets of a sequence.
- A sequence $\alpha = < e_{i_1} e_{i_2} e_{i_3}, \ldots, e_{i_m} >$ is a subsequence of another sequence $\beta = < e_1 e_2 e_3 \ldots e_n >$, denoted $\alpha \preccurlyeq \beta$, if there exist integers $i_1 < i_2 < \ldots < i_m$ and all events $e_{i_j} \in \alpha$ and $e_i \in \beta$ and $i_1 \leq 1$ and $i_m \leq n$, such that $e_{i_j} \subseteq e_i$.
- A sequential pattern is maximal if it is not a subsequence of any other sequential pattern.

Example of sequence

A ***sequence database***

Seq. ID	Sequence
10	<(**bd**)**cb**(ac)>
20	<(bf)(ce)b(fg)>
30	<(ah)(bf)abf>
40	<(be)(ce)d>
50	<a(**bd**)b**cb**(ade)>

A ***sequence***: <$\boxed{\text{(bd)}}\,\boxed{\text{c}}\,\boxed{\text{b}}\,\boxed{\text{(ac)}}$>

Elements

<ad(ae)> is a ***subsequence*** of <**a**(b**d**)bcb(**a**d**e**)>

Given ***support threshold*** *min_sup* =2, <(bd)cb> is a ***sequential pattern***

Support of a sequence

- The frequency or support of a sequence (or subsequence) S, denoted σ(S) is the total number of sequences of which S is a subsequence divided by the total number of sequences in the database D, whereas the absolute support (or support count) of a sequence (or subsequence) S is the total number of sequences in D of which S is a subsequence.

- A sequence is called frequent if its frequency is not less than a user-specified threshold, called minimum support, denoted min sup or the greek letter ξ .

Lexicographic order

- A frequent sequence S_α is called a frequent closed sequence if there exists no proper supersequence of S_α with the same support, that is, S_β such that $S_\alpha \preccurlyeq S_\beta$ and $\sigma(S_\alpha) = \sigma(S_\beta)$; otherwise it is said that sequence S_α is absorbed by S_β
  - Assume the frequent sequence S_β =< beadc > is the only superset of the frequent sequence S_α =< bea >, if, $\sigma(S_\alpha) = \sigma(S_\beta)$, then S_α is not a frequent closed sequence; on the other hand, if $\sigma(S_\alpha) > \sigma(S_\beta)$, then S_α is a frequent closed sequence. Notice that $\sigma(S_\beta)$ cannot be greater than $\sigma(S_\alpha)$, because $S_\alpha \preccurlyeq S_\beta$.

Example

Customer ID	Transaction Time	Items Bought
1	June 25 '93	30
1	June 30 '93	90
2	June 10 '93	10,20
2	June 15 '93	30
2	June 20 '93	40,60,70
3	June 25 '93	30,50,70
4	June 25 '93	30
4	June 30 '93	40,70
4	July 25 '93	90
5	June 12 '93	90

Customer ID	Customer Sequence
1	{ (30) (90) }
2	{ (10 20) (30) (40 60 70) }
3	{ (30 50 70) }
4	{ (30) (40 70) (90) }
5	{ (90) }

Maximal sequences with support > 25%
{ (30) (90) }
{ (30) (40 70) }

Note: Use Minsup of 25%
{ (10 20) (30) } Does not have minsup (Only supported by Cust. 2)
{ (30) }, { (70) }, { (30) (40) } … are not maximal.

The Algorithms

- Sort Phase
- Litemset Phase
- Transformation Phase
- Sequence Phase
- Maximal Phase

Sort Phase

- Sort the database:
  - Customer ID as the major key.
  - Transaction-Time as the minor key.
- Converts the original transaction database into a database of customer sequences

Customer ID	Transaction Time	Items Bought
1	June 25 '93	30
1	June 30 '93	90
2	June 10 '93	10,20
2	June 15 '93	30
2	June 20 '93	40,60,70
3	June 25 '93	30,50,70
4	June 25 '93	30
4	June 30 '93	40,70
4	July 25 '93	90
5	June 12 '93	90

Litemset Phase (1)

- Litemset (Large Itemset):
  - Supported by fraction of customers larger than minsup.
- Recall: each itemset in a large sequence has to be a large itemset
- Support counting: measured by fraction of customers

Customer ID	Transaction Time	Items Bought
1	June 25 '93	30
1	June 30 '93	90
2	June 10 '93	10,20
2	June 15 '93	30
2	June 20 '93	40,60,70
3	June 25 '93	30,50,70
4	June 25 '93	30
4	June 30 '93	40,70
4	July 25 '93	90
5	June 12 '93	90

Litemset Phase (2)

- Each large itemset is then mapped to a set of contiguous integers
  - Used to compare large itemsets in constant time and reduce the time required to check if a sequence is contained in a customer sequence.

Large Itemsets	Mapped To
(30)	1
(40)	2
(70)	3
(40 70)	4
(90)	5

Transformation Phase

- Need to repeatedly determine which of a given set of large sequences are contained in a customer sequence. To make this fast:
  - Replace each transaction with all litemsets contained in the transaction.
  - Transactions with no litemsets are dropped. (still considered for support counts)

Customer ID	Original Customer Sequence	Transformed Custumer Sequence	After Mapping
1	{ (30) (90) }	<{(30)} {(90)}>	<{1} {5}>
2	{ (10 20) (30) (40 60 70) }	<{(30)} {(40),(70),(40 70)}>	<{1} {2,3,4}>
3	{ (30) (50) (70) }	<{(30),(70)}>	<{1,3}>
4	{ (30) (40 70) (90) }	<{(30)} {(40),(70),(40 70)} {(90)}>	<{1} {2,3,4} {5}>
5	{ (90) }	<{(90)}>	<{5}>

Note: (10 20) dropped because of lack of support. (40 60 70) replaced with set of litemsets {(40),(70),(40 70)} (60 does not have min-sup)

Sequence Phase

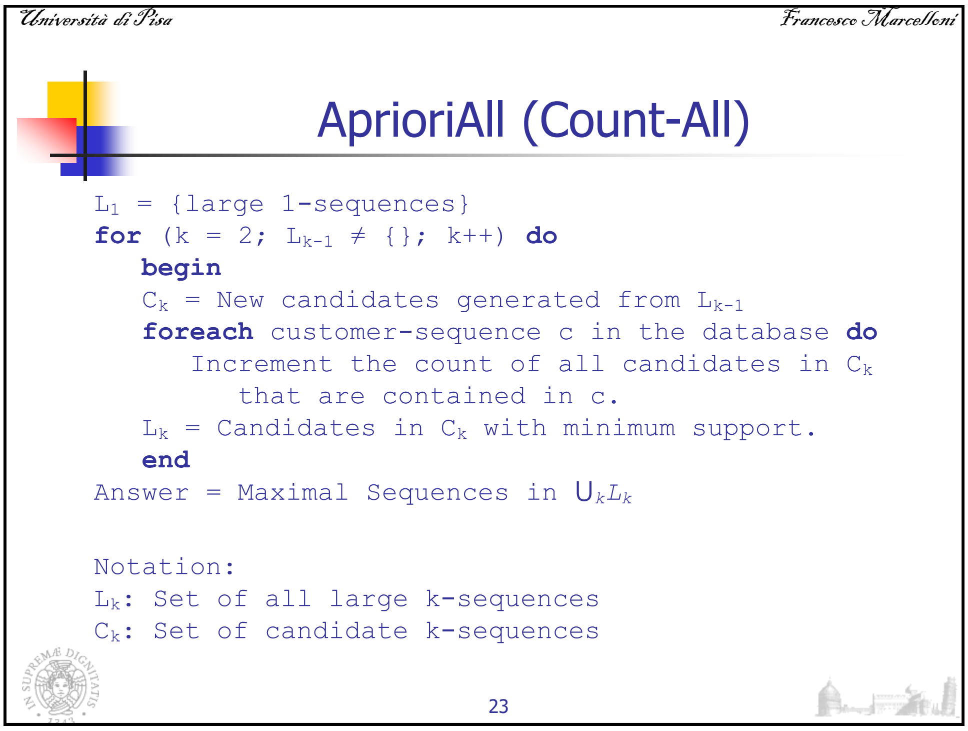
- Use the set of large itemsets to find the desired sequences.
- Similar structure to Apriori algorithms used to find large itemsets.
  - Use seed set to generate candidate sequences.
  - Count support for each candidate.
  - Eliminate candidate sequences which are not large.
- Two families of algorithms:
  - Count-all: count all large sequences including non-maximal sequences (it is careful with respect to the minimum support)
    - AprioriAll
  - Count-some: try to avoid counting non-maximal sequences by counting longer sequences first (it is careful with respect to maximality)
    - AprioriSome
    - DynamicSome

Maximal Phase

- Find maximal sequences among large sequences.
- k-sequence: sequence of length k
- S set of all large sequences

```
for (k=n; k>1; k--) do
  foreach k-sequence $s_k$ do
    delete from S all subsequences of $s_k$
```

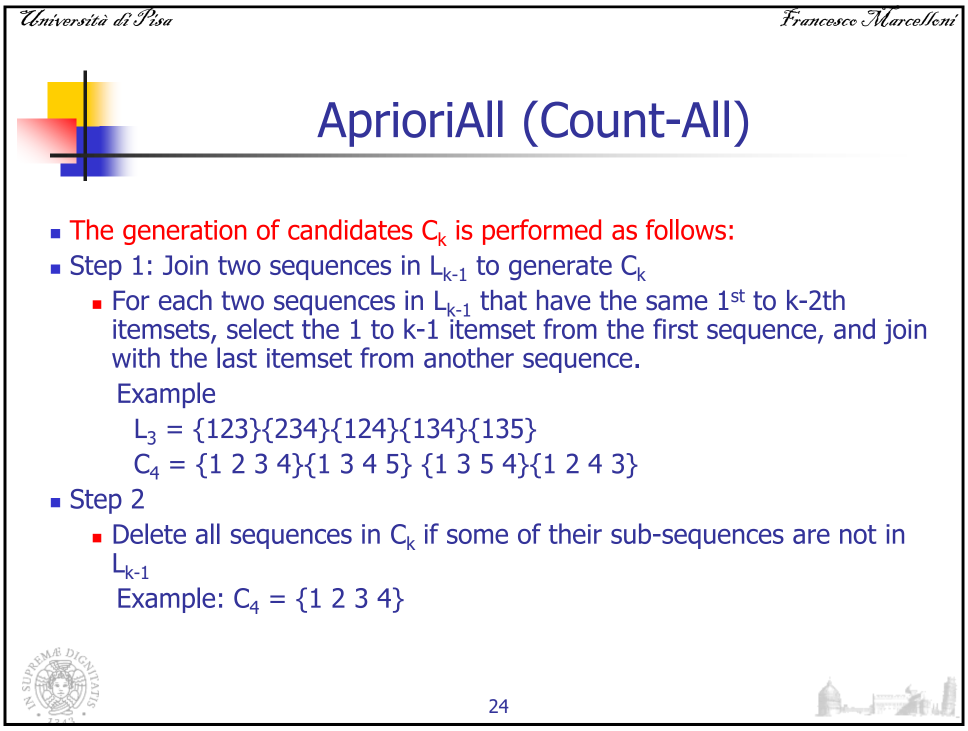
- Data-structures and an algorithm exist to do this efficiently. (hash trees)



AprioriAll (Count-All)

```
L_1 = {large 1-sequences}
for (k = 2; L_{k-1} ≠ {}; k++) do
   begin
   C_k = New candidates generated from L_{k-1}
   foreach customer-sequence c in the database do
      Increment the count of all candidates in C_k
         that are contained in c.
   L_k = Candidates in C_k with minimum support.
   end
Answer = Maximal Sequences in $\cup_k L_k$

Notation:
L_k: Set of all large k-sequences
C_k: Set of candidate k-sequences
```



AprioriAll (Count-All)

- The generation of candidates C_k is performed as follows:
- Step 1: Join two sequences in L_{k-1} to generate C_k
  - For each two sequences in L_{k-1} that have the same 1st to k-2th itemsets, select the 1 to k-1 itemset from the first sequence, and join with the last itemset from another sequence.

    Example

    L_3 = {123}{234}{124}{134}{135}

    C_4 = {1 2 3 4}{1 3 4 5} {1 3 5 4}{1 2 4 3}
- Step 2
  - Delete all sequences in C_k if some of their sub-sequences are not in L_{k-1}

    Example: C_4 = {1 2 3 4}

AprioriAll (Example)

Customer Sequences
<{1 5} {2} {3} {4} >
<{1} {3} {4} {3 5}>
<{1} {2} {3} {4}>
<{1} {3} {5}>
<{4} {5}>

L1

1-Seq	Support
<1>	4
<2>	2
<3>	4
<4>	4
<5>	4

L2

2-Seq	Support
<1 2>	2
<1 3>	4
<1 4>	3
<1 5>	3
<2 3>	2
<2 4>	2
<3 4>	3
<3 5>	2
<4 5>	2
<2 5>	0

L3

3-Seq	Support
<1 2 3>	2
<1 2 4>	2
<1 3 4>	3
<1 3 5>	2
<2 3 4>	2
<3 4 5>	1

L4

4-Seq	Support
<1 2 3 4>	2

Answer: <1 2 3 4>, <1 3 5>, <4 5>

Minisup = 25%

AprioriSome(1)

- Try to avoid counting non-maximal sequences by counting longer sequences first.
- 2 phases:
  - Forward Phase – find all large sequences of certain lengths.
  - Backward Phase – find all remaining large sequences.
    - For example, we might count sequences of length 1, 2, 4 and 6 in the forward phase and count sequences of length 3 and 5 in the backward phase
- Determines which lengths to count using next() function.
- next() takes in as a parameter the length of the sequence counted in the last pass.
  - next(k) = k + 1 - Same as AprioriAll
- Balances tradeoff between:
  - Counting non-maximal sequences
  - Counting extensions of small candidate sequences

AprioriSome(2)

```
function next(k: integer)
begin
    if (hit_k < 0.666)    return k + 1;
    elsif (hit_k < 0.75)  return k + 2;
    elsif (hit_k < 0.80)  return k + 3;
    elsif (hit_k < 0.85)  return k + 4;
    else                  return k + 5;
end
```

- $hit_k = |L_k|/|C_k|$
- Intuition: As hit_k increases, the time wasted by counting extensions of small candidates decreases.

AprioriSome (Forward Phase)

```
L_1 = {large 1-sequences}
C_1 = L_1
last = 1
for (k = 2; C_{k-1} ≠ {} and L_last ≠ {}; k++) do
   begin
   if (L_{k-1} known) then
      C_k = New candidates generated from L_{k-1}
    else
      C_k = New candidates generated from C_{k-1}
if (k==next(last)) then begin // (next k to count?)
   foreach customer-sequence c in the database do
      Increment the count of all candidates in C_k
         that are contained in c.
   L_k = Candidates in C_k with minimum support.
   last = k;
   end
end
```

AprioriSome (Forward Phase)

```
L_1 = {large 1-sequences}
C_1 = L_1
last = 1
for (k = 2; C_{k-1} ≠ {} and L_last ≠ {}; k++) do
   begin
   if (L_{k-1} known) then
      C_k = New candidates generated from L_{k-1}
   else
      C_k = New candidates generated from C_{k-1}
```

In the candidate generation, if the large sequence set L_{k-1} is not available, we use the candidate set C_{k-1} to generate C_k.
Correctness is maintained because $L_{k-1} \subseteq C_{k-1}$.

```
      L_k = Candidates in C_k with minimum support.
      last = k;
      end
end
```

AprioriSome(3)

Backward Phase:

- For all lengths which we skipped:
  - Delete sequences in candidate set which are contained in some large sequence.
  - Count remaining candidates and find all sequences with min. support.
- Also delete large sequences found in forward phase which are non-maximal.

AprioriSome (Backward Phase)

```
for (k--; k>=1; k--) do
   if ($L_k$ not found in forward phase) then begin
      Delete all sequences in $C_k$ contained in
         some $L_i$ i>k;
      foreach customer-sequence c in $D_T$ do
         Increment the count of all candidates in $C_k$
            that are contained in c
      $L_k$ = Candidates in $C_k$ with minimum support
      end
   else // $l_k$ already known
      Delete all sequences in $C_k$ contained in
         some $L_i$ i>k;
Answer = $\cup_k L_k$ //(Maximal Phase not Needed)

Notation: $D_T$; Transformed database
```

AprioriSome (Example)

⟨{1 5} {2} {3} {4}⟩
⟨{1} {3} {4} {3 5}⟩
⟨{1} {2} {3} {4}⟩
⟨{1} {3} {5}⟩
⟨{4} {5}⟩

Figure 8: Customer Sequences

Forward Phase:

next(k) = 2k
minsup = 2

L_1

1-Sequences	Support
⟨1⟩	4
⟨2⟩	2
⟨3⟩	4
⟨4⟩	4
⟨5⟩	4

L_2

2-Sequences	Support
⟨1 2⟩	2
⟨1 3⟩	4
⟨1 4⟩	3
⟨1 5⟩	3
⟨2 3⟩	2
⟨2 4⟩	2
⟨3 4⟩	3
⟨3 5⟩	2
⟨4 5⟩	2

C_3

3-Sequences
⟨1 2 3⟩
⟨1 2 4⟩
⟨1 2 5⟩
⟨1 3 4⟩
⟨1 3 5⟩
⟨2 3 4⟩
⟨3 4 5⟩

L_4

4-Sequences	Support
⟨1 2 3 4⟩	2

AprioriSome (Example)

$\langle \{1\ 5\}\ \{2\}\ \{3\}\ \{4\} \rangle$
$\langle \{1\}\ \{3\}\ \{4\}\ \{3\ 5\} \rangle$
$\langle \{1\}\ \{2\}\ \{3\}\ \{4\} \rangle$
$\langle \{1\}\ \{3\}\ \{5\} \rangle$
$\langle \{4\}\ \{5\} \rangle$

Figure 8: Customer Sequences

Backward Phase: next(k) = 2k
minsup = 2

L_4

4-Sequences	Support
⟨1 2 3 4⟩	2

C_3

3-Sequences
~~⟨1 2 3⟩~~
~~⟨1 2 4⟩~~
~~⟨1 2 5⟩~~
~~⟨1 3 4⟩~~
⟨1 3 5⟩
~~⟨2 3 4⟩~~
~~⟨3 4 5⟩~~

L_2

2-Sequences	Support
~~⟨1 2⟩~~	2
~~⟨1 3⟩~~	4
~~⟨1 4⟩~~	3
~~⟨1 5⟩~~	3
~~⟨2 3⟩~~	2
~~⟨2 4⟩~~	2
~~⟨3 4⟩~~	3
~~⟨3 5⟩~~	2
⟨4 5⟩	2

L_1

1-Sequences	Support
~~⟨1⟩~~	4
~~⟨2⟩~~	2
~~⟨3⟩~~	4
~~⟨4⟩~~	4
~~⟨5⟩~~	4

AprioriDynamicSome(1)

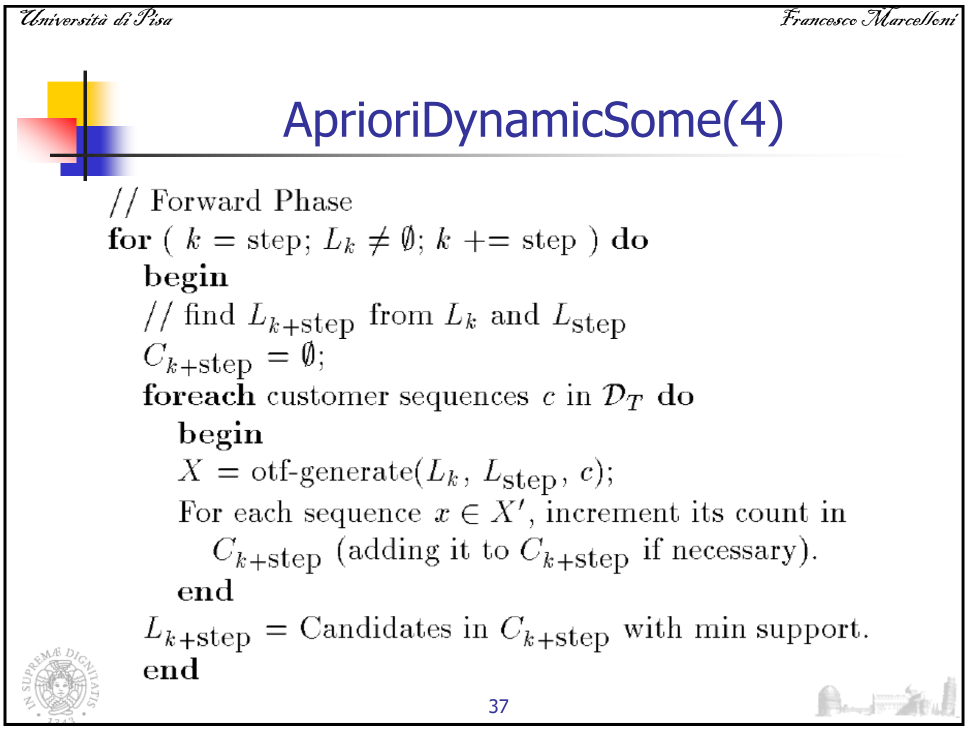
- Like AprioriSome, skip counting candidate sequences of certain lengths in the forward phase.
- The candidate sequences that are counted is determined by the variable step.
- Initialization phase:
  - all the candidate sequences of length upto and including step are counted.
- Forward phase, all sequences whose lengths are multiples of step are counted.
  - step = 3 -> will count sequences of lengths 1, 2, and 3 in the initialization phase, and 6, 9, 12,... in the forward phase.
  - We can generate sequences of length 6 by joining sequences of length 3. We can generate sequences of length 9 by joining sequences of length 6 with sequences of length 3, etc. However, to generate the sequences of length 3, we need sequences of lengths 1 and 2, and hence the initialization phase.

AprioriDynamicSome(2)

- Backward phase:
  - count sequences for the lengths we skipped over during the forward phase. However, unlike in AprioriSome, these candidate sequences were not generated in the forward phase.
  - The intermediate phase generates them.
  - For example, assume that we count L_3 and L_6, and L_9 turns out to be empty in the forward phase. We generate C_7 and C_8 (intermediate phase), and then count C_8 followed by C_7 after deleting non-maximal sequences (backward phase). This process is then repeated for C_4 and C_5.

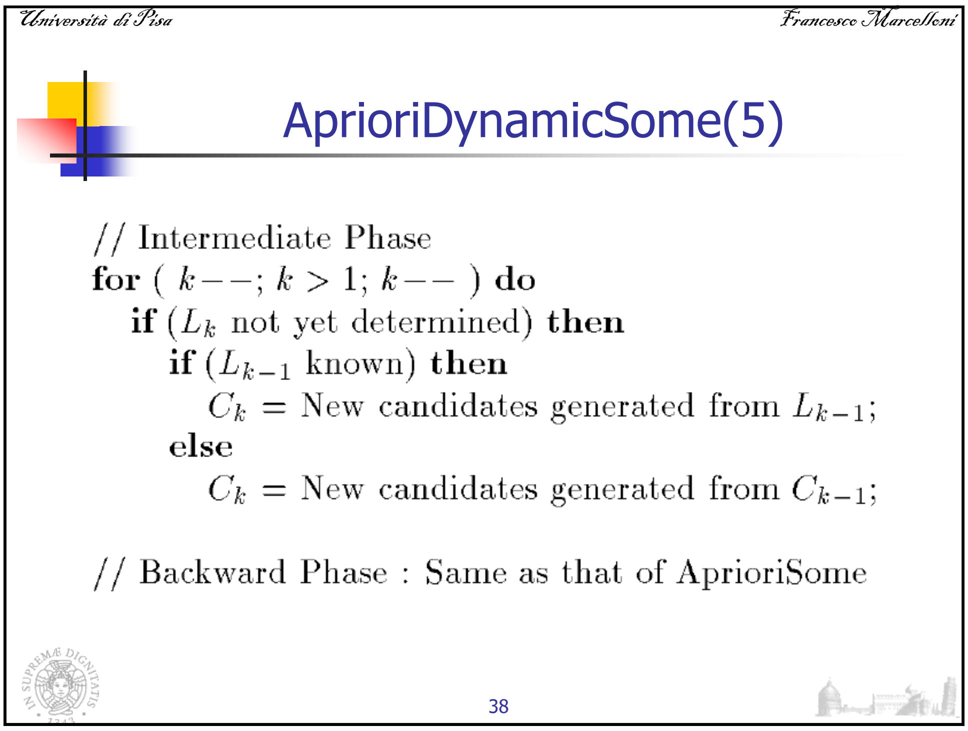
AprioriDynamicSome(3)

```
// *step* is an integer $\geq 1$
// Initialization Phase
$L_1$ = {large 1-sequences}; // Result of litemset phase
**for** ( $k = 2$; $k <=$ step **and** $L_{k-1} \neq \emptyset$; $k$++ ) **do**
    **begin**
    $C_k$ = New candidates generated from $L_{k-1}$;
    **foreach** customer-sequence $c$ **in** $\mathcal{D}_T$ **do**
        Increment the count of all candidates in $C_k$
            that are contained in $c$.
    $L_k$ = Candidates in $C_k$
            with minimum support.
    **end**
```



AprioriDynamicSome(4)

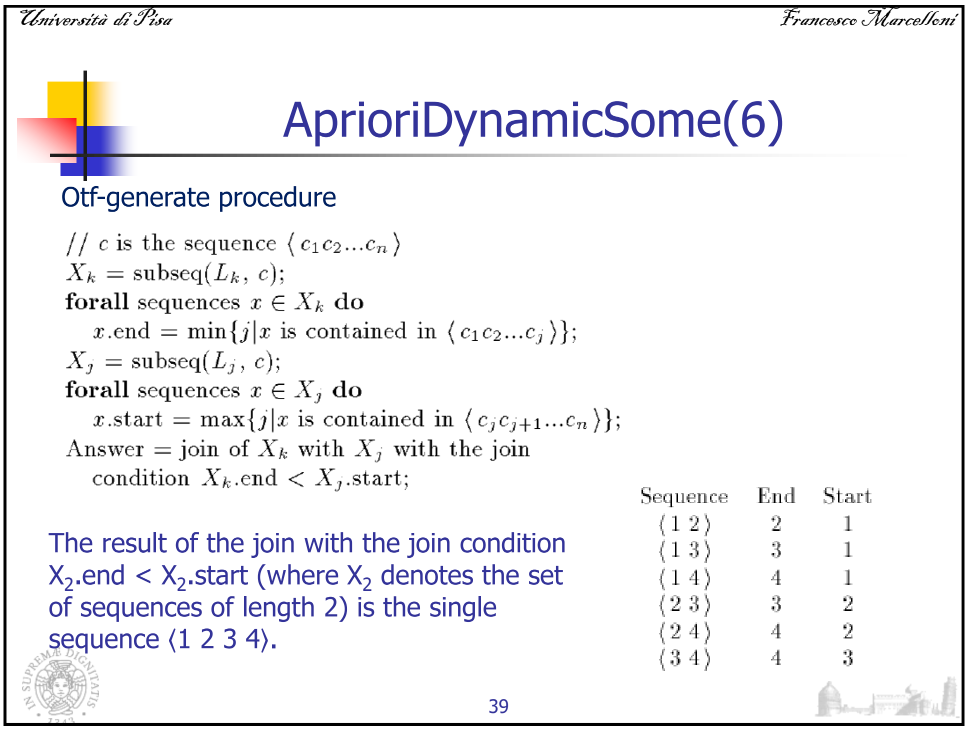
```
// Forward Phase
for ( $k$ = step; $L_k \neq \emptyset$; $k$ += step ) do
  begin
  // find $L_{k+\text{step}}$ from $L_k$ and $L_{\text{step}}$
  $C_{k+\text{step}} = \emptyset$;
  foreach customer sequences $c$ in $\mathcal{D}_T$ do
    begin
    $X$ = otf-generate($L_k$, $L_{\text{step}}$, $c$);
    For each sequence $x \in X'$, increment its count in
      $C_{k+\text{step}}$ (adding it to $C_{k+\text{step}}$ if necessary).
    end
  $L_{k+\text{step}}$ = Candidates in $C_{k+\text{step}}$ with min support.
  end
```



AprioriDynamicSome(5)

```
// Intermediate Phase
for ( $k$--; $k > 1$; $k$-- ) do
  if ($L_k$ not yet determined) then
    if ($L_{k-1}$ known) then
      $C_k$ = New candidates generated from $L_{k-1}$;
    else
      $C_k$ = New candidates generated from $C_{k-1}$;

// Backward Phase : Same as that of AprioriSome
```



AprioriDynamicSome(6)

Otf-generate procedure

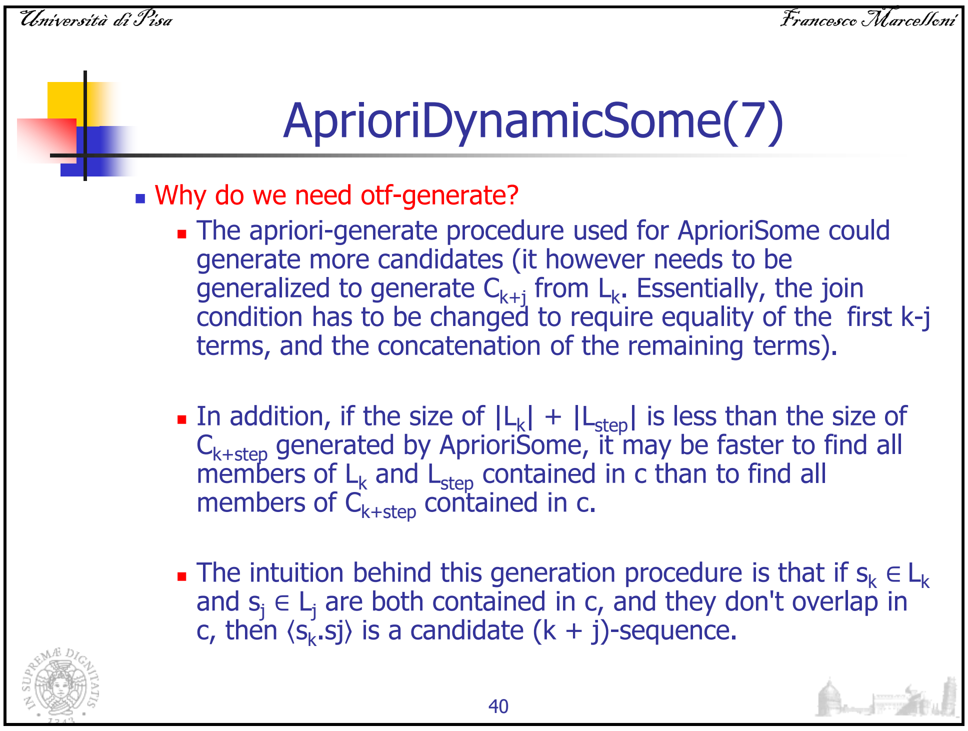
```
// $c$ is the sequence $\langle c_1 c_2 \ldots c_n \rangle$
$X_k$ = subseq($L_k$, $c$);
**forall** sequences $x \in X_k$ **do**
    $x$.end = min$\{j | x$ is contained in $\langle c_1 c_2 \ldots c_j \rangle\}$;
$X_j$ = subseq($L_j$, $c$);
**forall** sequences $x \in X_j$ **do**
    $x$.start = max$\{j | x$ is contained in $\langle c_j c_{j+1} \ldots c_n \rangle\}$;
Answer = join of $X_k$ with $X_j$ with the join
    condition $X_k$.end < $X_j$.start;
```

The result of the join with the join condition X_2.end < X_2.start (where X_2 denotes the set of sequences of length 2) is the single sequence ⟨1 2 3 4⟩.

Sequence	End	Start
⟨1 2⟩	2	1
⟨1 3⟩	3	1
⟨1 4⟩	4	1
⟨2 3⟩	3	2
⟨2 4⟩	4	2
⟨3 4⟩	4	3

39

39



AprioriDynamicSome(7)

- Why do we need otf-generate?
  - The apriori-generate procedure used for AprioriSome could generate more candidates (it however needs to be generalized to generate C_{k+j} from L_k. Essentially, the join condition has to be changed to require equality of the first k-j terms, and the concatenation of the remaining terms).
  - In addition, if the size of $|L_k| + |L_{step}|$ is less than the size of C_{k+step} generated by AprioriSome, it may be faster to find all members of L_k and L_{step} contained in c than to find all members of C_{k+step} contained in c.
  - The intuition behind this generation procedure is that if $s_k \in L_k$ and $s_j \in L_j$ are both contained in c, and they don't overlap in c, then $\langle s_k.sj \rangle$ is a candidate (k + j)-sequence.

40

40

AprioriDynamicSome(8)

- Example

Initialization:

L_1

1-Sequences	Support
$\langle 1 \rangle$	4
$\langle 2 \rangle$	2
$\langle 3 \rangle$	4
$\langle 4 \rangle$	4
$\langle 5 \rangle$	4

L_2

2-Sequences	Support
$\langle 1\ 2 \rangle$	2
$\langle 1\ 3 \rangle$	4
$\langle 1\ 4 \rangle$	3
$\langle 1\ 5 \rangle$	3
$\langle 2\ 3 \rangle$	2
$\langle 2\ 4 \rangle$	2
$\langle 3\ 4 \rangle$	3
$\langle 3\ 5 \rangle$	2
$\langle 4\ 5 \rangle$	2

Forward phase (step=2): C_4: $\langle 1\ 2\ 3\ 4 \rangle$ $\langle 1\ 3\ 4\ 5 \rangle$ L_4: $\langle 1\ 2\ 3\ 4 \rangle$ C_6: $\langle\ \rangle$

Intermediate phase: C_3: $\langle 1\ 2\ 3 \rangle$ $\langle 1\ 2\ 4 \rangle$ $\langle 1\ 2\ 5 \rangle$ $\langle 1\ 3\ 4 \rangle$ $\langle 1\ 3\ 5 \rangle$ $\langle 2\ 3\ 4 \rangle$ $\langle 3\ 4\ 5 \rangle$ C_5: $\langle\ \rangle$

Backward phase: we count only C_3

Performance (1)

- Used generated datasets again
- Parameters for data:

$\lvert D\rvert$	Number of customers (= size of Database)
$\lvert C\rvert$	Average number of transactions per Customer
$\lvert T\rvert$	Average number of items per Transaction
$\lvert S\rvert$	Average length of maximal potentially large Sequences
$\lvert I\rvert$	Average size of Itemsets in maximal potentially large sequences
N_S	Number of maximal potentially large Sequences
N_I	Number of maximal potentially large Itemsets
N	Number of items

=

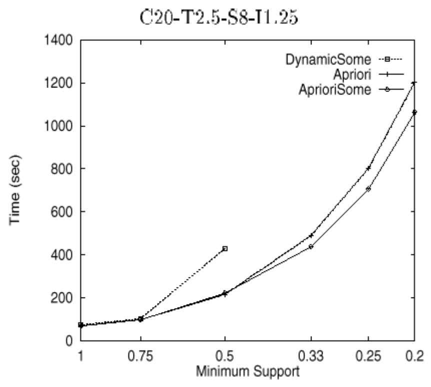
Name	$\lvert C\rvert$	$\lvert T\rvert$	$\lvert S\rvert$	$\lvert I\rvert$	Size (MB)
C10-T5-S4-I1.25	10	5	4	1.25	5.8
C10-T5-S4-I2.5	10	5	4	2.5	6.0
C20-T2.5-S4-I1.25	20	2.5	4	1.25	6.9
C20-T2.5-S8-I1.25	20	2.5	8	1.25	7.8

$N_S = 5000$, $N_I = 25000$ and $N = 10000$

$|D| = 250{,}000$

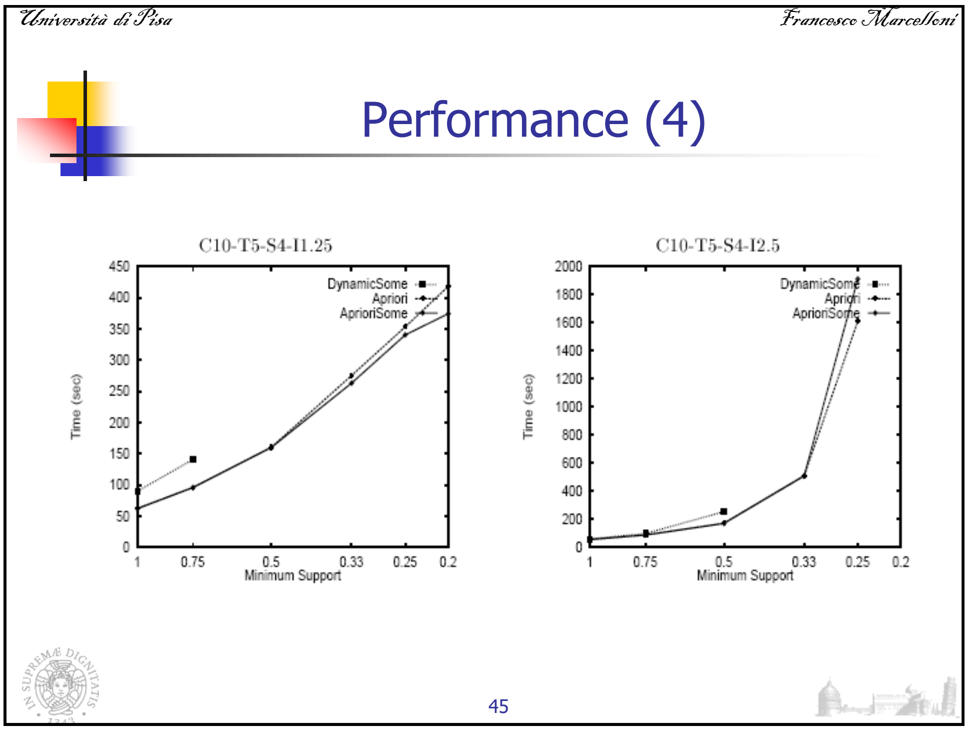
Performance (2)

- DynamicSome generates too many candidates
- AprioriSome does a little better than AprioriAll
  - It avoids counting many non-maximal sequences



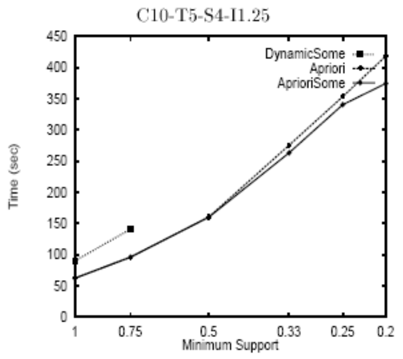
Performance (3)

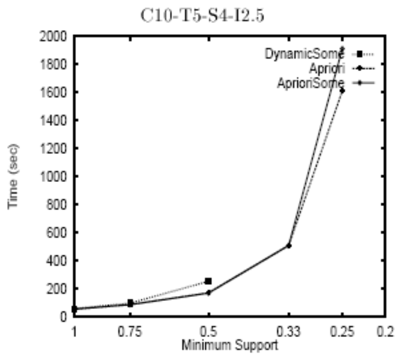
- Advantage of AprioriSome is reduced for 2 reasons:
  - AprioriSome generates more candidates.
  - Candidates remain memory resident even if skipped over.

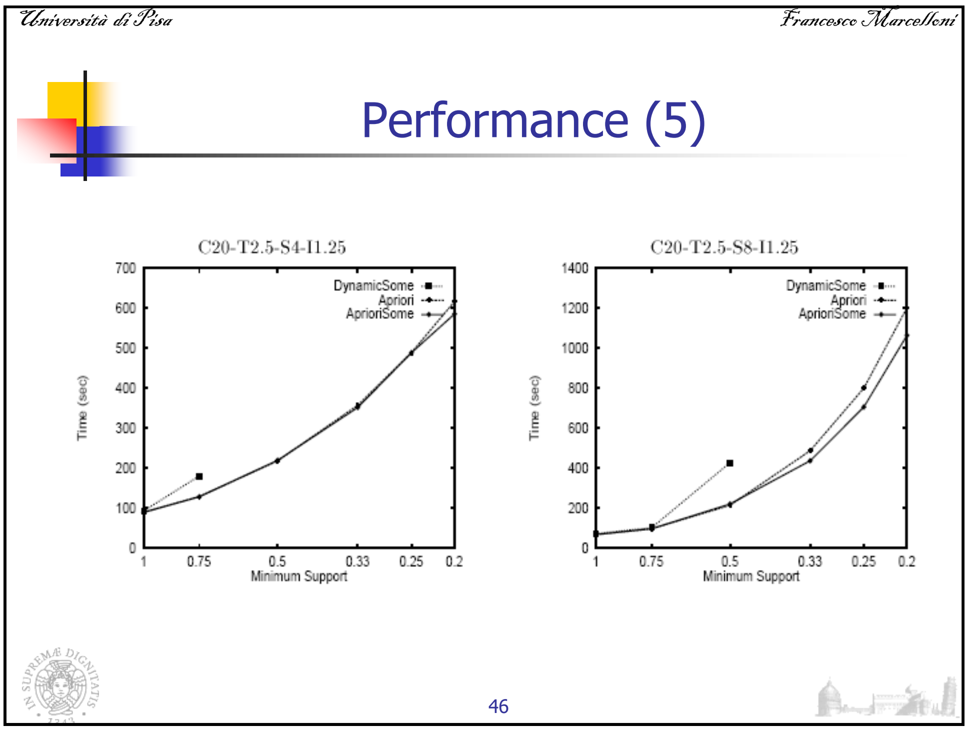


Università di Pisa Francesco Marcelloni

Performance (4)

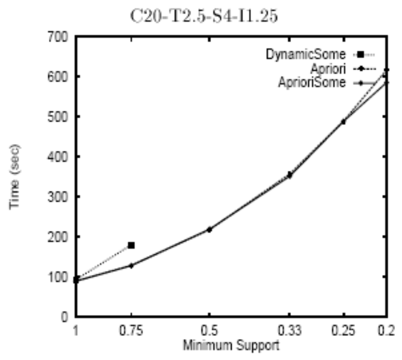


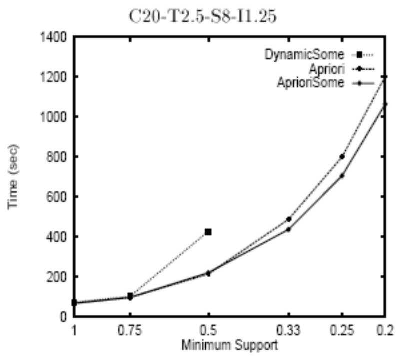




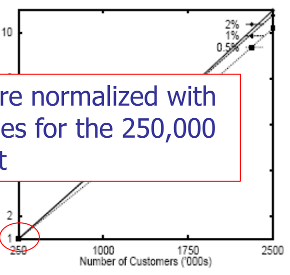
Università di Pisa Francesco Marcelloni

Performance (5)

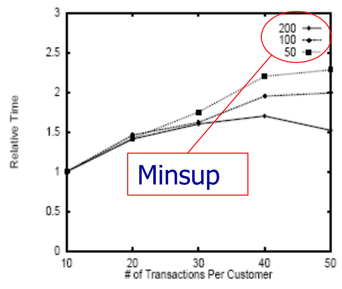




Performance (6) AprioriSome



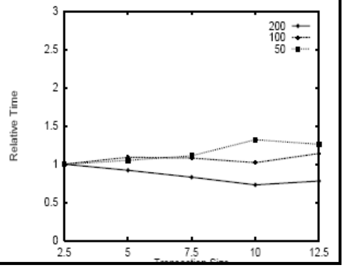
Execution times are normalized with respect to the times for the 250,000 customers dataset



Minsup

DataSet:C10-T2.5-S4-I1.25

In the experiments on the right, the size of the database was roughly constant by keeping the product of the average customer-sequence size and the number of customers constant.



Bottlenecks of Apriori-like methods

- A huge set of candidates could be generated
- Many scans of database in mining
- Encounter difficulty when mining long sequential patterns
  - Exponential number of short candidates
  - A length-100 sequential pattern needs $\sum_{i=1}^{100} \binom{100}{i} = 2^{100} - 1 \approx 10^{30}$

FreeSpan: FP-growth for sequential pattern mining

- Frequent pattern tree and FP-growth (SIGMOD'2000):
  - A successful algorithm for mining frequent (unordered) itemsets
- Can we extend FP-growth to sequential pattern mining?
  - A straightforward construction of sequential-pattern tree does not work well.
  - A level-by-level project does not achieve high performance either
  - An interesting method is to explore alternative-level projection
- J. Han, J. Pei, B. Mortazavi-Asl, Q. Chen, U. Dayal, M.-C. Hsu, "FreeSpan: Frequent Pattern-Projected Sequential Pattern Mining", Proc. 2000 Int. Conf. on Knowledge Discovery and Data Mining (KDD'00), August 2000.

FreeSpan: Mapping into projected sequence database

- Find frequent items from database
  - List of frequent items in support descending order is called f_list
- All sequential patterns can be divided into several subsets without overlap

Sequence Database *SDB*
< (bd) c b (ac) >
< (bf) (ce) b (fg) >
< (ah) (bf) a b f >
< (be) (ce) d >
< a (bd) b c b (ade) >

f_list: b:5, c:4, a:3, d:3, e:3, f:2

All seq. pat. can be divided into 6 subsets:
•Seq. pat. containing item *f*
•Those containing *e* but no *f*
•Those containing *d* but no *e* nor *f*
•Those containing *a* but no *d*, *e* or *f*
•Those containing *c* but no *a*, *d*, *e* or *f*
•Those containing only item *b*

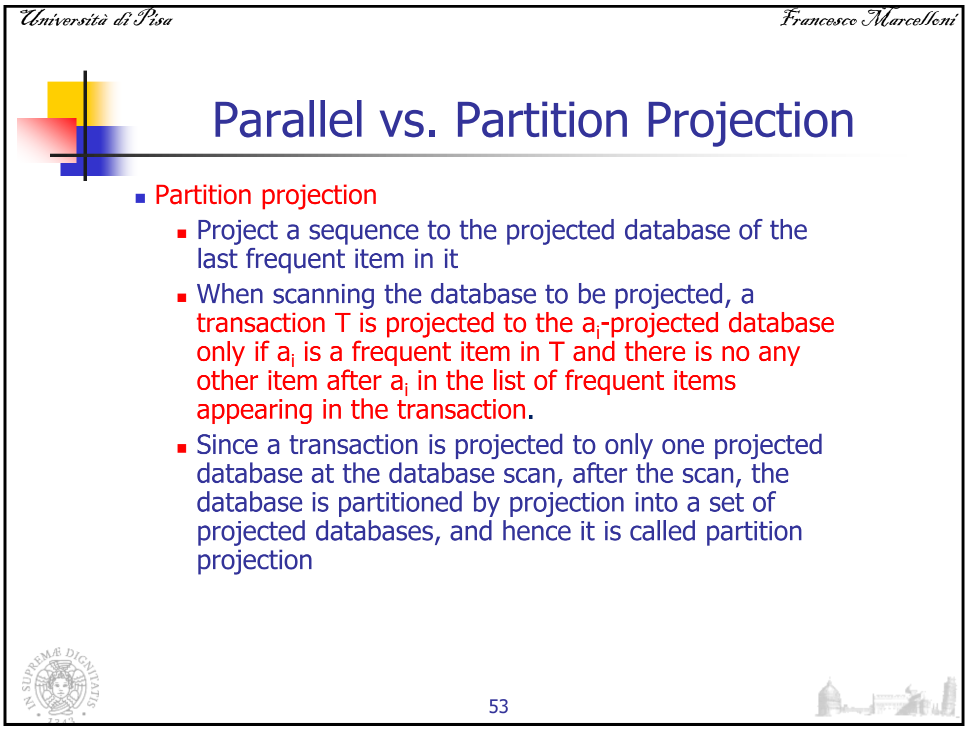
Mine Sequential Patterns Using Projected Databases

- The complete set of sequential patterns containing item i but no items following i in f_list can be found in the i-projected database
- A sequence s is projected as s_i to the i-projected database if there is at least an item i in s
- s_i is a copy of s by removing from s all the infrequent items and any frequent item j following i in f_list

- Example: <(ah)(bf)abf> is projected to *f*-projected database as <a(bf)abf>, and to *a*-projected database as <abab>, and to *b*-projected database as <bb>

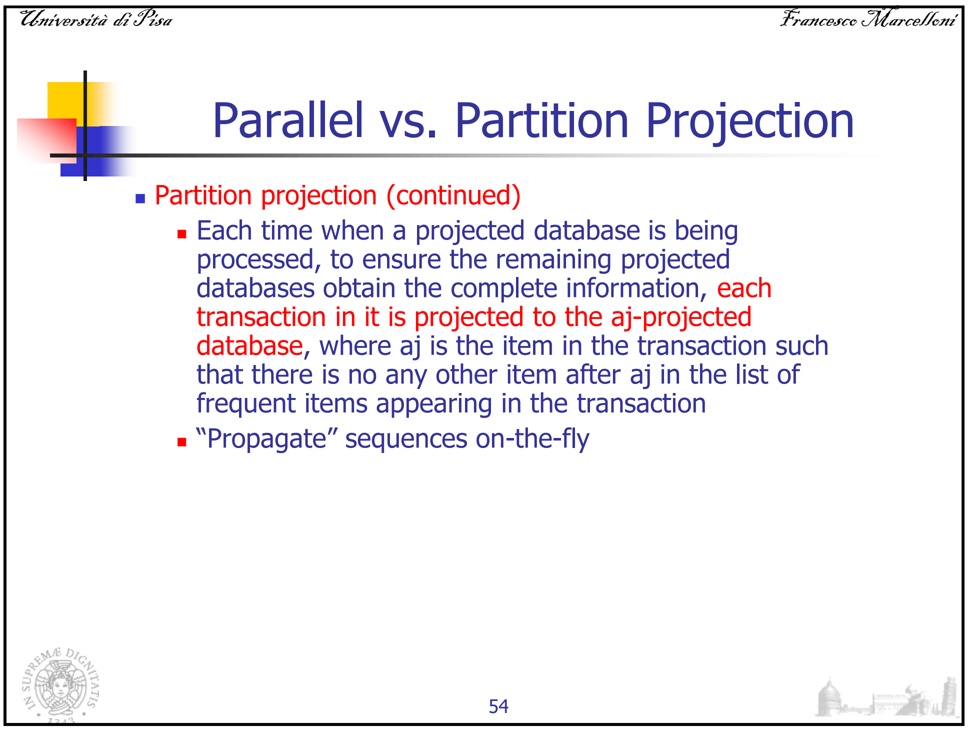
Parallel vs. Partition Projection

- Parallel projection
  - Scan database once, form all projected dbs at a time
  - May derive many and rather large projected dbs if sequence on average contains many frequent items
  - Let each transaction contain on average l frequent items. A transaction is then projected to l – 1 projected database. The total size of the projected data from this transaction is 1+2+⋯+(l −1) = l(l−1)/2. This implies that the total size of the single item-projected databases is about (l−1)/2 times of that of the original database.
- To avoid such an overhead, partition projection method is proposed



Parallel vs. Partition Projection

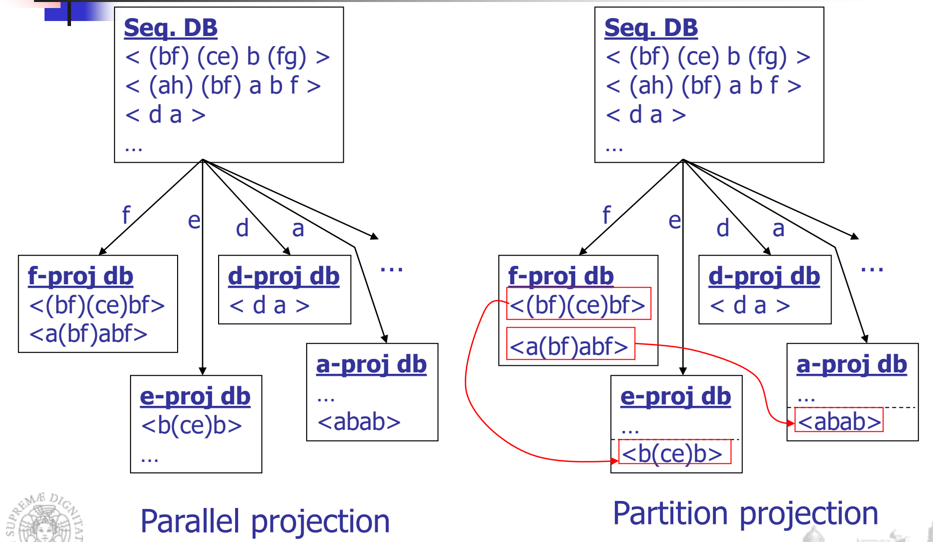
- Partition projection
  - Project a sequence to the projected database of the last frequent item in it
  - When scanning the database to be projected, a transaction T is projected to the a_i-projected database only if a_i is a frequent item in T and there is no any other item after a_i in the list of frequent items appearing in the transaction.
  - Since a transaction is projected to only one projected database at the database scan, after the scan, the database is partitioned by projection into a set of projected databases, and hence it is called partition projection



Parallel vs. Partition Projection

- Partition projection (continued)
  - Each time when a projected database is being processed, to ensure the remaining projected databases obtain the complete information, each transaction in it is projected to the aj-projected database, where aj is the item in the transaction such that there is no any other item after aj in the list of frequent items appearing in the transaction
  - "Propagate" sequences on-the-fly

Example of Database Projection

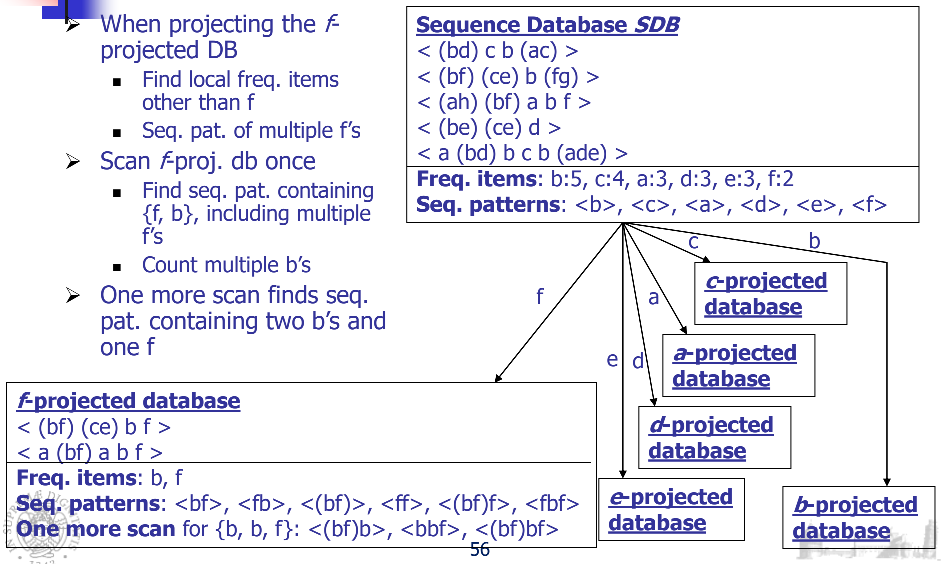


Parallel projection

Partition projection

Mining Sequential Patterns by Database Projection

- When projecting the *f*-projected DB
  - Find local freq. items other than f
  - Seq. pat. of multiple f's
- Scan *f*-proj. db once
  - Find seq. pat. containing {f, b}, including multiple f's
  - Count multiple b's
- One more scan finds seq. pat. containing two b's and one f



Sequence Database *SDB*
< (bd) c b (ac) >
< (bf) (ce) b (fg) >
< (ah) (bf) a b f >
< (be) (ce) d >
< a (bd) b c b (ade) >
Freq. items: b:5, c:4, a:3, d:3, e:3, f:2
Seq. patterns: <b>, <c>, <a>, <d>, <e>, <f>

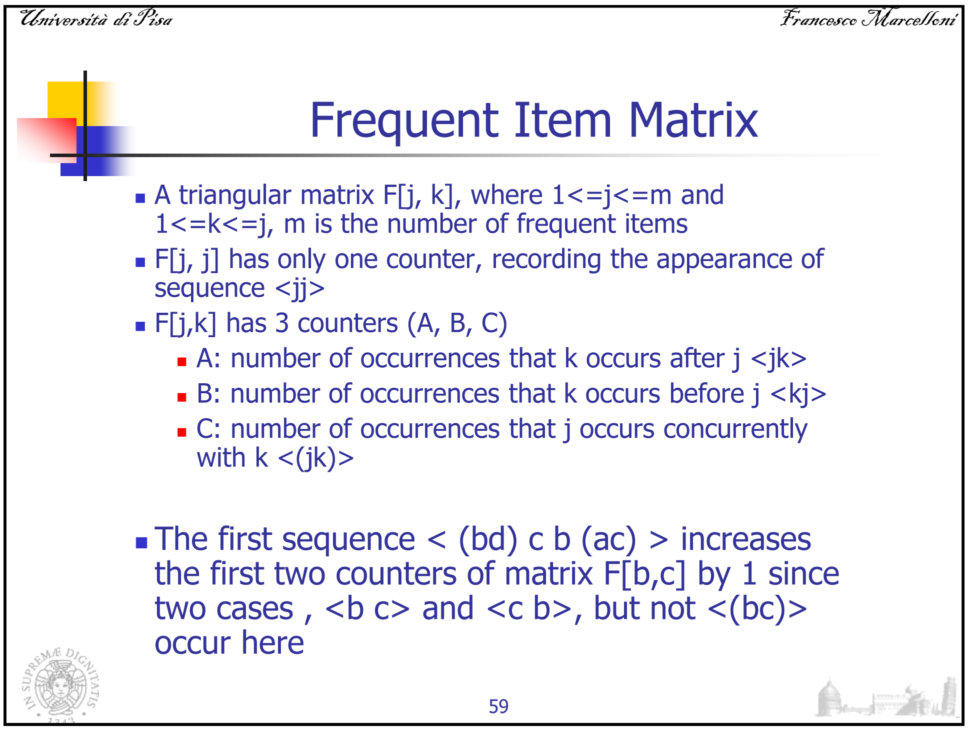
***f*-projected database**
< (bf) (ce) b f >
< a (bf) a b f >
Freq. items: b, f
Seq. patterns: <bf>, <fb>, <(bf)>, <ff>, <(bf)f>, <fbf>
One more scan for {b, b, f}: <(bf)b>, <bbf>, <(bf)bf>

Mining by Level-by-Level Projected Databases

- Algorithm
  - Scan database once, find frequent items and get f_list
  - Recursively do database projection level by level
- Pros and cons
  - Benefits: only need to find frequent items in each projected database, instead of exploring candidate sequence generation
  - The number of combinations is much less than their possible combinations
  - Cost: partition and projection of databases
  - Works well in sparse databases

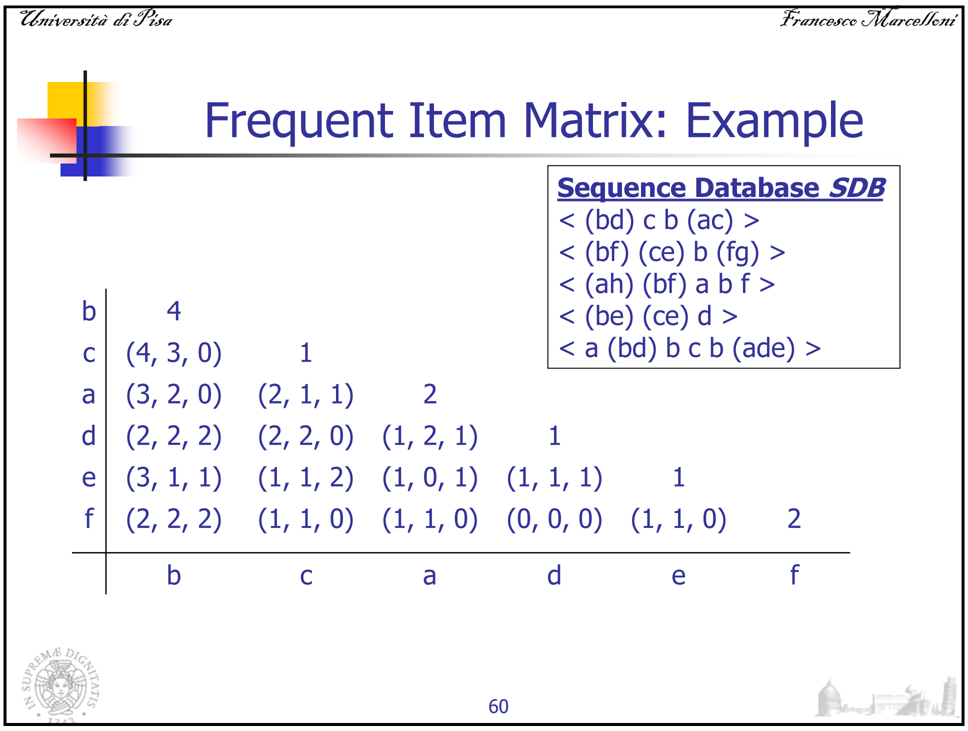
Mining by Alternative Level Projected Databases

- Postpone the generation of projected databases, take each database as a level-shared, combined projected database
- Algorithm
  - Scan database, find freq. items and get f_list
  - Perform alternative-level projection mining
    - Construct frequent item matrix
    - Generate length-2 sequential patterns and annotations on item repeating patterns and projected databases
    - Scan database to generate item-repeating patterns and projected databases
    - Do matrix projection mining on projected databases recursively, if there are still longer candidate patterns to be mined.



Frequent Item Matrix

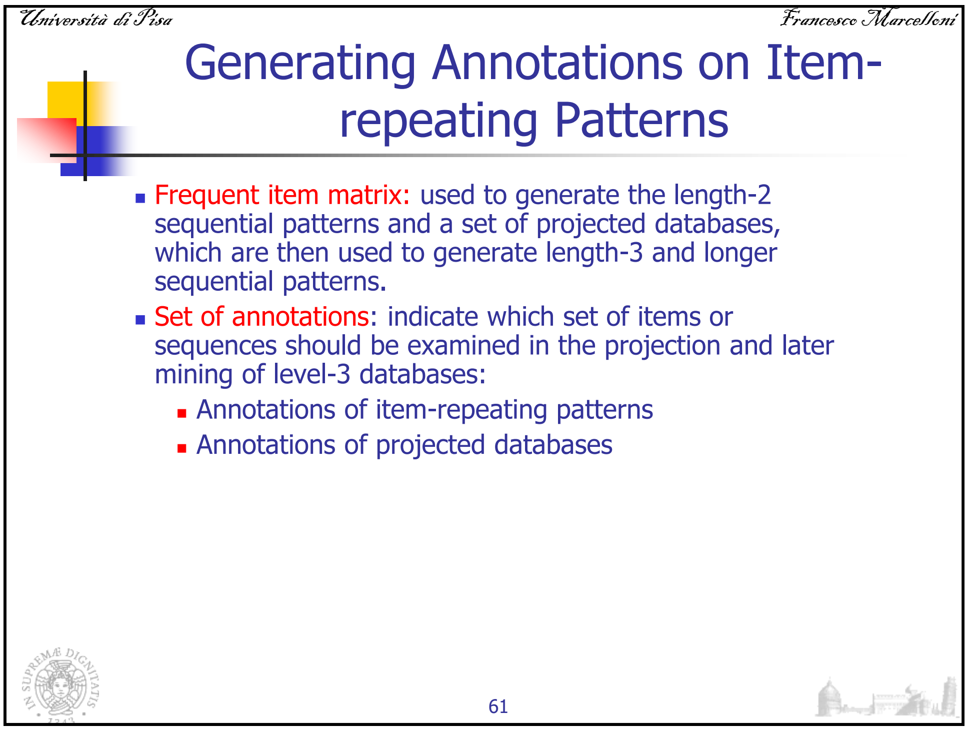
- A triangular matrix F[j, k], where 1<=j<=m and 1<=k<=j, m is the number of frequent items
- F[j, j] has only one counter, recording the appearance of sequence <jj>
- F[j,k] has 3 counters (A, B, C)
  - A: number of occurrences that k occurs after j <jk>
  - B: number of occurrences that k occurs before j <kj>
  - C: number of occurrences that j occurs concurrently with k <(jk)>
- The first sequence < (bd) c b (ac) > increases the first two counters of matrix F[b,c] by 1 since two cases , <b c> and <c b>, but not <(bc)> occur here



Frequent Item Matrix: Example

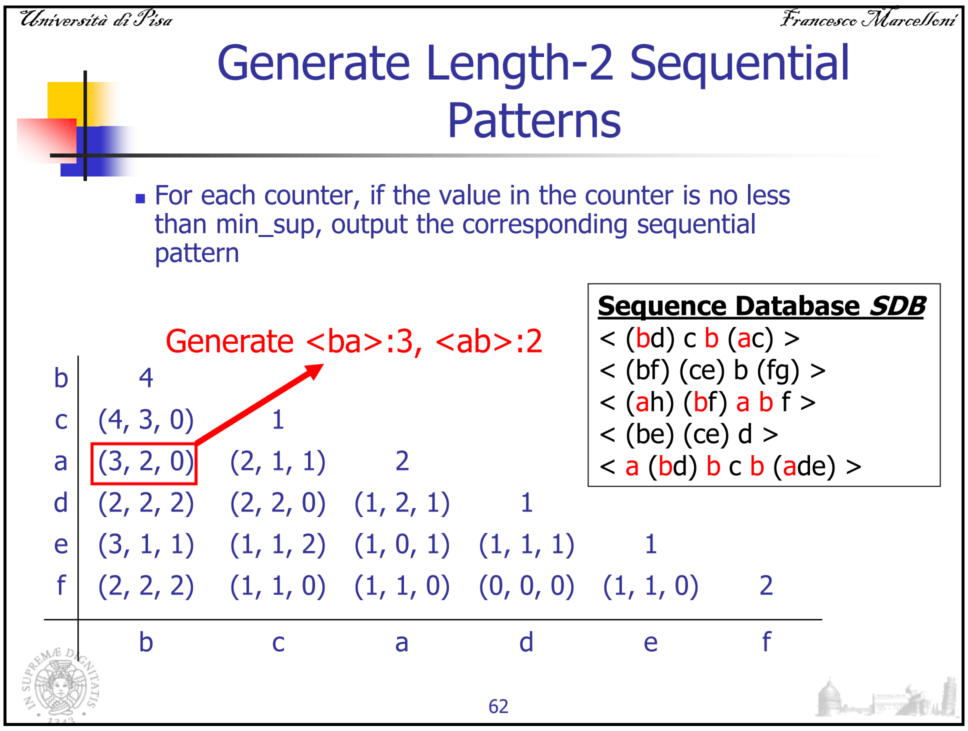
Sequence Database *SDB*
< (bd) c b (ac) >
< (bf) (ce) b (fg) >
< (ah) (bf) a b f >
< (be) (ce) d >
< a (bd) b c b (ade) >

b	4					
c	(4, 3, 0)	1				
a	(3, 2, 0)	(2, 1, 1)	2			
d	(2, 2, 2)	(2, 2, 0)	(1, 2, 1)	1		
e	(3, 1, 1)	(1, 1, 2)	(1, 0, 1)	(1, 1, 1)	1	
f	(2, 2, 2)	(1, 1, 0)	(1, 1, 0)	(0, 0, 0)	(1, 1, 0)	2
	b	c	a	d	e	f



Generating Annotations on Item-repeating Patterns

- Frequent item matrix: used to generate the length-2 sequential patterns and a set of projected databases, which are then used to generate length-3 and longer sequential patterns.
- Set of annotations: indicate which set of items or sequences should be examined in the projection and later mining of level-3 databases:
  - Annotations of item-repeating patterns
  - Annotations of projected databases



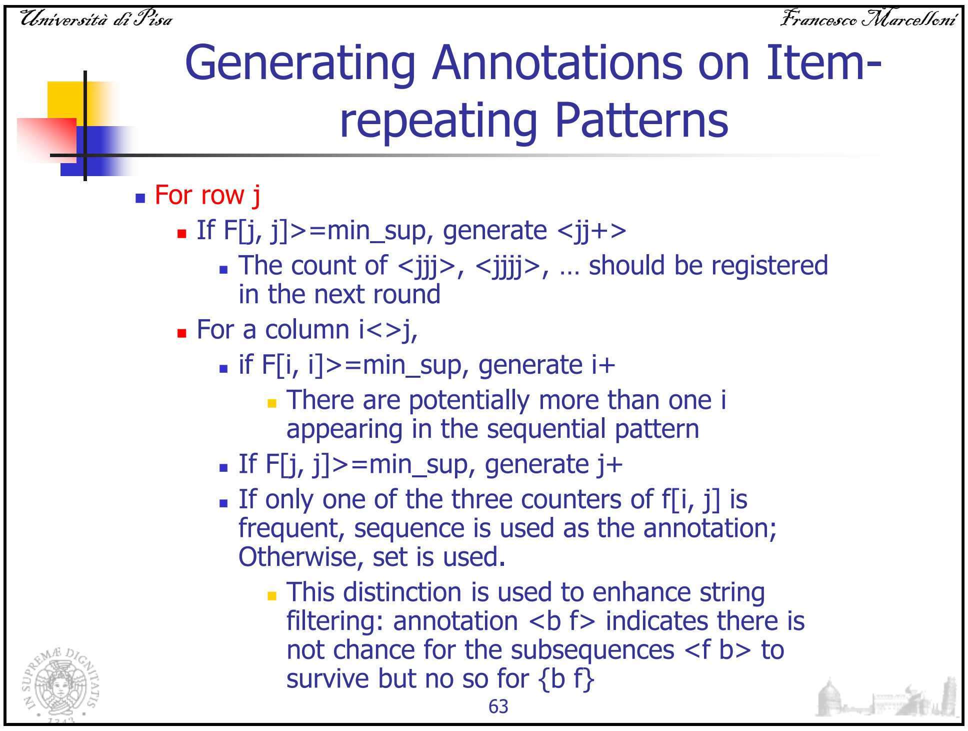
Generate Length-2 Sequential Patterns

- For each counter, if the value in the counter is no less than min_sup, output the corresponding sequential pattern

Generate <ba>:3, <ab>:2

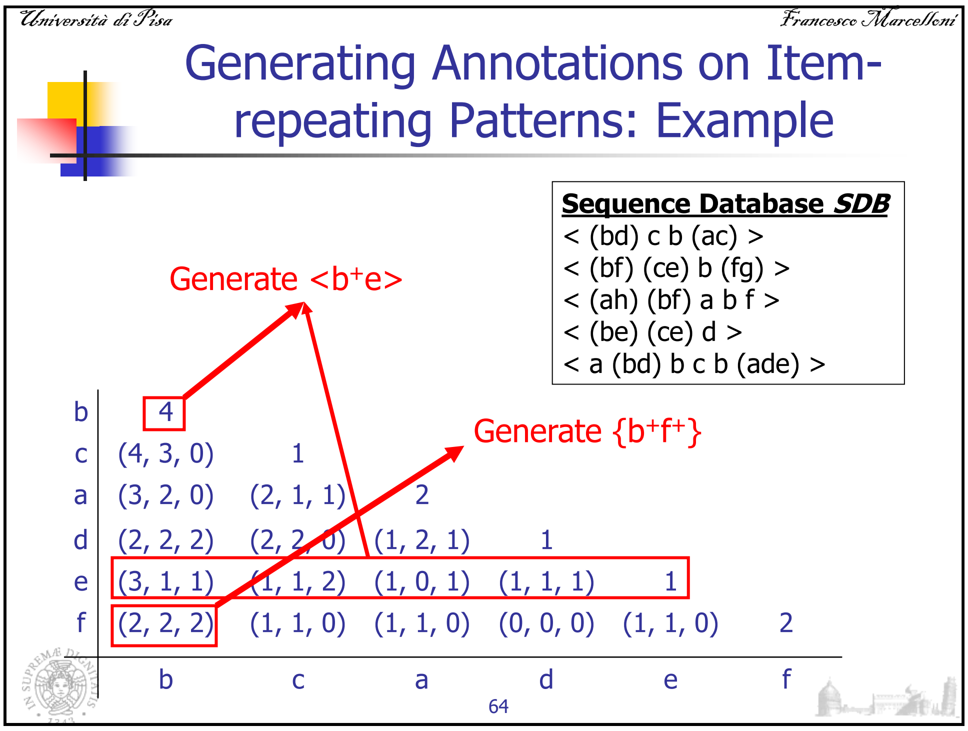
b	4					
c	(4, 3, 0)	1				
a	$\boxed{(3, 2, 0)}$	(2, 1, 1)	2			
d	(2, 2, 2)	(2, 2, 0)	(1, 2, 1)	1		
e	(3, 1, 1)	(1, 1, 2)	(1, 0, 1)	(1, 1, 1)	1	
f	(2, 2, 2)	(1, 1, 0)	(1, 1, 0)	(0, 0, 0)	(1, 1, 0)	2
	b	c	a	d	e	f

<u>Sequence Database *SDB*</u>
< (bd) c b (ac) >
< (bf) (ce) b (fg) >
< (ah) (bf) a b f >
< (be) (ce) d >
< a (bd) b c b (ade) >



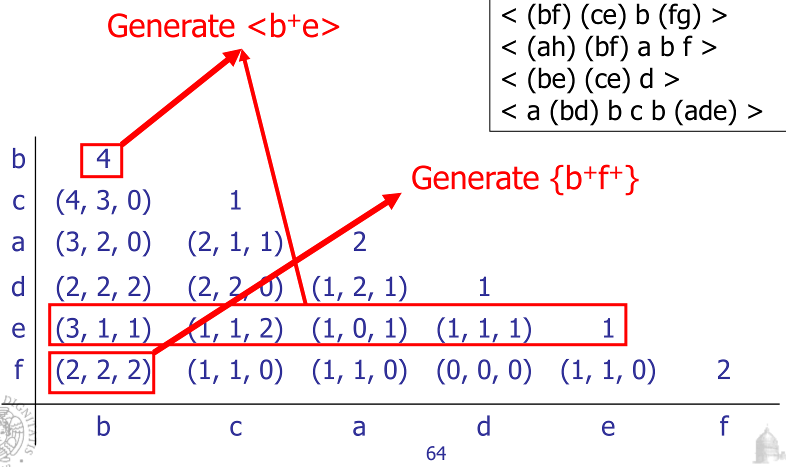
Generating Annotations on Item-repeating Patterns

- For row j
  - If F[j, j]>=min_sup, generate <jj+>
    - The count of <jjj>, <jjjj>, … should be registered in the next round
  - For a column i<>j,
    - if F[i, i]>=min_sup, generate i+
      - There are potentially more than one i appearing in the sequential pattern
    - If F[j, j]>=min_sup, generate j+
    - If only one of the three counters of f[i, j] is frequent, sequence is used as the annotation; Otherwise, set is used.
      - This distinction is used to enhance string filtering: annotation <b f> indicates there is not chance for the subsequences <f b> to survive but no so for {b f}



Generating Annotations on Item-repeating Patterns: Example

<u>Sequence Database *SDB*</u>
< (bd) c b (ac) >
< (bf) (ce) b (fg) >
< (ah) (bf) a b f >
< (be) (ce) d >
< a (bd) b c b (ade) >



b	4					
c	(4, 3, 0)	1				
a	(3, 2, 0)	(2, 1, 1)	2			
d	(2, 2, 2)	(2, 2, 0)	(1, 2, 1)	1		
e	(3, 1, 1)	(1, 1, 2)	(1, 0, 1)	(1, 1, 1)	1	
f	(2, 2, 2)	(1, 1, 0)	(1, 1, 0)	(0, 0, 0)	(1, 1, 0)	2
	b	c	a	d	e	f

Generating Annotations on Projected Databases

- For row j
  - For each i<j, if F[i, j], F[k, j] and F[i, k](k<i) may form a pattern generating triple (i.e., all the corresponding pairs are frequent), k should be added to i's projected column set
  - If there is a choice between sequence or set, sequence is preferred

b	4					
c	$\boxed{(4, 3, 0)}$	1				
a	(3, 2, 0)	(2, 1, 1)	2			
d	(2, 2, 2)	(2, 2, 0)	(1, 2, 1)	1		
e	$\boxed{(3, 1, 1)}$	$\boxed{(1, 1, 2)}$	(1, 0, 1)	(1, 1, 1)	1	
f	(2, 2, 2)	(1, 1, 0)	(1, 1, 0)	(0, 0, 0)	(1, 1, 0)	2
	b	c	a	d	e	f

Generate <(ce)>:{b} indicating generating <(ce)>-projected database with {b} as the only item included

Generate Length-2 Patterns and Annotations

Item	Length-2 seq. pat.	Ann. on rep. Items	Ann. on proj. DBs
f	<bf>:2, <fb>:2, <(bf)>:2	$<b^+f^+>$	None
e	<be>:3, <(ce)>:2	$<b^+e>$	<(ce)>:{b}
d	<bd>:2, <db>:2, <(bd)>:2, <cd>:2, <dc>:2, <da>:2	$\{b^+d\}$, $<da^+>$	<da>:{b,c}, {cd}:{b}
a	<ba>:3, <ab>:2, <ca>:2, <aa:2>	$<aa^+>$, $\{a^+b^+\}$, $<ca^+>$	<ca>:{b}
c	<bc:4>, <cb>:3	$\{b^+c\}$	None
b	<bb>:4	$<bb^+>$	None

b	4					
c	(4, 3, 0)	1				
a	(3, 2, 0)	(2, 1, 1)	2			
d	(2, 2, 2)	(2, 2, 0)	(1, 2, 1)	1		
e	(3, 1, 1)	(1, 1, 2)	(1, 0, 1)	(1, 1, 1)	1	
f	(2, 2, 2)	(1, 1, 0)	(1, 1, 0)	(0, 0, 0)	(1, 1, 0)	2
	b	c	a	d	e	f

Seq. Database *SDB*
< (bd) c b (ac) >
< (bf) (ce) b (fg) >
< (ah) (bf) a b f >
< (be) (ce) d >
< a (bd) b c b (ade) >

Generate Length-2 Patterns and Annotations

- Based on the annotations for item-repeating patterns and projected databases, S is scanned one more time.
- The set item-repeating patterns generated is {<bbf>:2, <fbf>: 2, <(bf)b>:2 , <(bf)f>:2, <(bf)bf>:2, <(bd)b>:2, <bba>:2, <aba>:2, <abb>:2, <bcb>:3, <bbc>:2}
- There are four projected databases: <(ce)>:{b}, <da>:{b,c}, {cd}:{b} and <ca>:{b}.
  - For a projected database whose annotation contains exactly three items, its associated sequential patterns can be obtained by a simple scan of the projected database.
  - For a projected database whose annotation contains more than three items, one can construct frequent item matrix for this projected database and recursively mine its sequential patterns by the alternative-level projection technique.

Generate Length-2 Patterns and Annotations

Ann.	<(ce)>:{b}	<da>:{b, c}	{cd}:{b}	<ca>:{b}
Proj. DB	<b(ce)b>, <b(ce)>	<(bd)cb(ac)>, <(bd)bcba>	<(bd)cbc>, <bcd>, <(bd)bcbd>	<bcba>, <bbcba>
Seq. Pat.	<bce>:2	<(bd)a>:2, <dca>:2, <dba>:2, <(bd)ca>:2, <(bd)ba>:2, <dcba>:2, <(bd)cba>:2	<bcd>:2, <(bc)c>:2, <dcb>:2, <(bd)cb>:2, <(bd)bc>:2	<bca>:2, <cba>:2, <bcba>:2

b	4					
c	(4, 3, 0)	1				
a	(3, 2, 0)	(2, 1, 1)	2			
d	(2, 2, 2)	(2, 2, 0)	(1, 2, 1)	1		
e	(3, 1, 1)	(1, 1, 2)	(1, 0, 1)	(1, 1, 1)	1	
f	(2, 2, 2)	(1, 1, 0)	(1, 1, 0)	(0, 0, 0)	(1, 1, 0)	2
	b	c	a	d	e	f

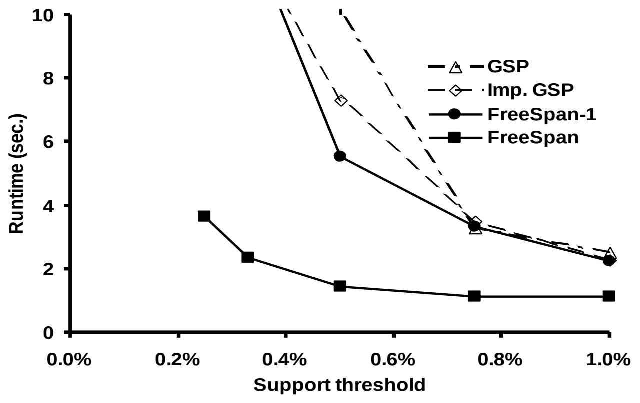
<u>Seq. Database *SDB*</u>
< (bd) c b (ac) >
< (bf) (ce) b (fg) >
< (ah) (bf) a b f >
< (be) (ce) d >
< a (bd) b c b (ade) >

Performance Study

- Data sets: 10000 items
- Comparison algorithms
  - GSP
  - Improved GSP: using pattern growth to find length-2 sequential patterns
  - Freespan1: level-by-level projection
  - FreeSpan: alternative level projection

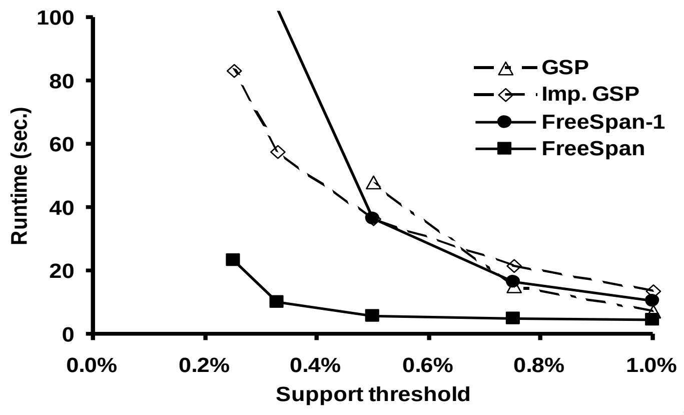
Experimental Results

Dataset 1: average number of items per transaction = 2.5



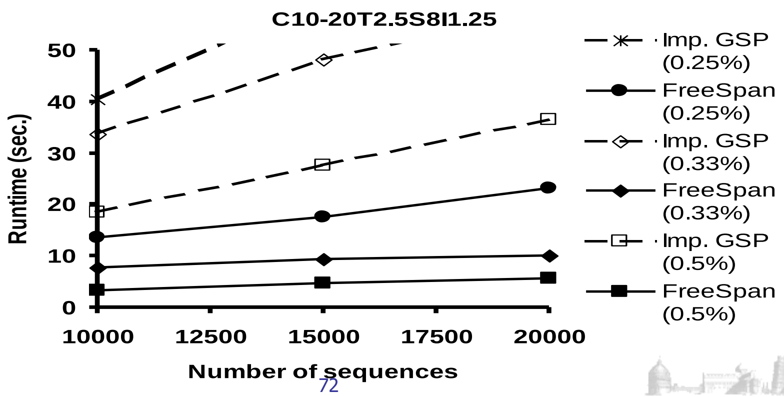
Experimental Results

Dataset 1: average number of items per transaction = 5

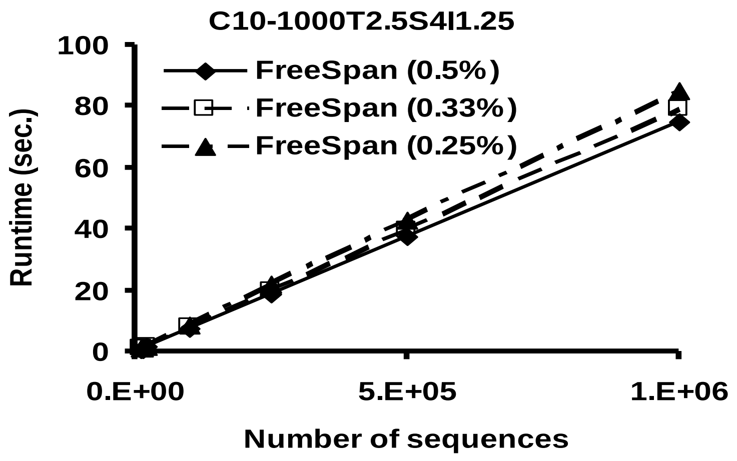


Scalability with the number of sequences

- FreeSpan and improved GSP are tested
- Both algorithms are linearly scalable
- FreeSpan is much better



Scalability of FreeSpan in Large Database



Why FreeSpan outperforms Apriori-like Methods?

- Projects a large sequence database recursively into a set of small projected sequence databases based on the currently mined frequent sets
- The alternatively-level projection in FreeSpan reduces the cost of scanning multiple projected databases and takes advantages of Apriori -like 3-way candidate filtering