Concurrent Apriori Data Mining Algorithms

Vassil Halatchev
Department of Electrical Engineering and Computer Science
York University, Toronto
October 8, 2015
Outline

• Why it is important
• Introduction to Association Rule Mining (a Data Mining technique)
• Overview of Sequential Apriori algorithm
• The 3 Parallel Apriori algorithm implementations
• Future work
What is Data Mining?

- **Mining knowledge from data**
- **Data mining** [Han, 2001]
  - Process of extracting interesting (non-trivial, implicit, previously unknown and potentially useful) knowledge or patterns from data in large databases
- **Objectives of data mining:**
  - Discover knowledge that characterizes general properties of data
  - Discover patterns on the previous and current data in order to make predictions on future data

Source: Data Mining CSE6412
Big Data Era

• Term introduced by Roger Magoulas in 2010
• “A massive volume of both structured and unstructured data that is so large it is difficult to process using traditional database and software techniques” - Webopedia
• Multicore machines allow for efficient concurrent computations, which require proper synchronization techniques, that can significantly reduce task completion times
Big Data Era

- 45 zettabytes (45 x 1000^3 gigabytes) of data produced in 2020

Figure 1
Data is growing at a 40 percent compound annual rate, reaching nearly 45 ZB by 2020

Source: Oracle, 2012
Why Mine Association Rules?

Objective:
- Finding interesting co-occurring items (or objects, events) in a given data set.

Examples:
- Given a database of transactions, each transaction is a list of items (purchased by a customer in a visit), you may find:
  - computer \(\rightarrow\) financial_management_software
    [support=2%, confidence=60%]
- From a student database, you may find:
  - major(x, “CS”) \(\wedge\) gpa(x, “A”) \(\rightarrow\) has_taken(x, “DB”) [1%, 75%]
Association Rule Mining Applications

• **Market basket analysis** (e.g. Stock market, Shopping patterns)
• **Medical diagnosis** (e.g. Causal effect relationship)
• **Census data** (e.g. Population Demographics)
• **Bio-sequences** (e.g. DNA, Protein)
• **Web Log** (e.g. Fraud detection, Web page traversal patterns)
What Kind of Databases?

**Transactional database TDB**

<table>
<thead>
<tr>
<th>TID</th>
<th>Items</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>f, a, c, d, g, i, m, p</td>
</tr>
<tr>
<td>200</td>
<td>a, b, c, f, l, m, o</td>
</tr>
<tr>
<td>300</td>
<td>b, f, h, j, o</td>
</tr>
<tr>
<td>400</td>
<td>b, c, k, s, p</td>
</tr>
<tr>
<td>500</td>
<td>a, f, c, e, l, p, m, n</td>
</tr>
</tbody>
</table>

- **Itemset**: a set of items
- A **transaction** is a tuple (tid, X)
  - Transaction ID tid
  - Itemset X
- A **transactional database** is a set of transactions
  - In many cases, a transaction database can be treated as a set of itemsets (ignore TIDs)

**Association rule from TDB** (relates two itemsets):

- \{a, c, m\} \rightarrow \{l\} [support=40\%, confidence=66.7\%]
Definition of Association Rule

An association rule is of the form:

\[ X \rightarrow Y \] [support, confidence]

where

- \( X \subseteq I, Y \subseteq I, X \cap Y = \emptyset \) and \( I \) is a set of items (objects or events).
- **support**: probability that a transaction (or a record) contains \( X \) and \( Y \), i.e.,
  
  \[ \text{support}(X \rightarrow Y) = P(X \cup Y) \]

- **confidence**: conditional probability that a transaction (or a record) having \( X \) also contains \( Y \), i.e.,
  
  \[ \text{confidence}(X \rightarrow Y) = P(Y|X) \]

- A rule associates one set of items (events) with another set of items (events)

Source: Data Mining CSE6412
Support and Confidence: Example

\[
support(X \rightarrow Y) = P(X \cup Y) \\
\text{confidence}(X \rightarrow Y) = P(Y|X)
\]

<table>
<thead>
<tr>
<th>Transaction ID</th>
<th>Items Bought</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000</td>
<td>A,B,C</td>
</tr>
<tr>
<td>1000</td>
<td>A,C</td>
</tr>
<tr>
<td>4000</td>
<td>A,D</td>
</tr>
<tr>
<td>5000</td>
<td>B,E,F</td>
</tr>
</tbody>
</table>

- \(\{A\} \rightarrow \{C\}\) (50%, 66.7%)
- \(\{C\} \rightarrow \{A\}\) (50%, 100%)
- \(\{A, C\} \rightarrow \{B\}\) (25%, 50%)
- \(\{A, B\} \rightarrow \{E\}\) (0%, 0%)

Relative frequency is used to estimate the probability.

Source: Data Mining CSE6412
Mining Association Rules

Problem statement
Given a minimum support (min_sup), also called support threshold, and a minimum confidence (min_conf), also called confidence threshold, find all association rules that satisfy both min_sup and min_conf from a data set D.
How to Mine Association Rules

- A two-step process:
  - Find all frequent itemsets — the key step
  - Generate strong association rules from frequent itemsets.

- Example: given min_sup=50% and min_conf=50%

<table>
<thead>
<tr>
<th>Transaction ID</th>
<th>Items Bought</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000</td>
<td>A, B, C</td>
</tr>
<tr>
<td>1000</td>
<td>A, C</td>
</tr>
<tr>
<td>4000</td>
<td>A, D</td>
</tr>
<tr>
<td>5000</td>
<td>B, E, F</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Frequent Itemset</th>
<th>Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>{A}</td>
<td>75%</td>
</tr>
<tr>
<td>{B}</td>
<td>50%</td>
</tr>
<tr>
<td>{C}</td>
<td>50%</td>
</tr>
<tr>
<td>{A, C}</td>
<td>50%</td>
</tr>
</tbody>
</table>

- Generate strong rules:
  - \{A\} → \{C\} [support=50%, confidence=66.6%]
  - \{C\} → \{A\} [support=50%, confidence=100%]
Candidate Generation

How to Generate Candidates? (i.e. How to Generate $C_{k+1}$ from $L_k$)

- Given $L_k$ = the set of frequent $k$-itemsets
  - List the items in each itemset of $L_k$ in an order
    
    $L_3 = $ \{1 2 3\} \{1 2 4\} \{1 3 4\} \{1 3 5\} \{2 3 4\} 

- Given $L_k$, generate $C_{k+1}$ in two steps:
  - **Join Step**: Join $L_k$ with $L_k$ by joining two $k$-itemsets in $L_k$. Two $k$-itemsets are joinable if their first ($k-1$) items are the same and the last item in the first itemset is smaller than the last item in the second itemset (the condition for joining two members of $L_k$).
    - Now, $C_4$ = \{1 2 3 4\}, \{1 3 4 5\}
  - **Prune Step**: Delete all candidates in $C_{k+1}$ that have a non-frequent subset by checking all length-$k$ subsets of a candidate
    - Now, $C_5$ = \{1 2 3 4\}

Example of Candidate Generation

- $L_4$ = \{abcd, abcg, abdg, abef, abeh, acdg, bcdg\}
- **Self-joining**: $L_4 \ast L_4$
  - $abcdn$ from abcd and abcg
  - $abefh$ from abef and aceh
- **Pruning**:
  - $abefh$ is removed because $abfh$ or $aefh$ or $befh$ is not in $L_4$
- $C_5$ = \{abcdg\}

Source: Data Mining CSE6412
Apriori Algorithm

• Proposed by Agrawal and Srikant in 1994

Apriori Algorithm (Flow Chart)

Apriori Algorithm Example

Imported steps:
- Generating candidates
- Counting supports of candidates by scanning DB

Source: Data Mining CSE6412
My Paper

3 Parallel Apriori Algorithms

IMPORTANT: Algorithms implemented on a shared-nothing multiprocessor communicating via a Message Passing Interface (MPI)

• Count Distribution
  • Each processor calculates its Candidate Set Counts from its local Database and end of each pass sends out Candidate Set Counts to all other processors.

• Data Distribution
  • Each processor is assigned a mutually exclusive partition of the Candidate Set on which it computes the count and end of pass sends out Candidate Set Tuple to all other processors.

• Candidate Distribution
  • Both Candidate Set and Database is partitioned during some pass k, so that each processor can operate independently.
# Notations

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L_k$</td>
<td>Set of frequent $k$-itemsets (those with minimum support). Each member of this set has two fields: i) itemset and ii) support count.</td>
</tr>
<tr>
<td>$C_k$</td>
<td>Set of candidate $k$-itemsets (potentially frequent itemsets). Each member of this set has two fields: i) itemset and ii) support count.</td>
</tr>
<tr>
<td>$P_i$</td>
<td>Processor with id $i$.</td>
</tr>
<tr>
<td>$D_i$</td>
<td>The dataset local to the processor $P_i$</td>
</tr>
<tr>
<td>$DR_i$</td>
<td>The dataset local to the processor $P_i$ after repartitioning.</td>
</tr>
<tr>
<td>$C_k^i$</td>
<td>The candidate set maintained with the Processor $P_i$ during the $k$th pass (there are $k$ items in each candidate).</td>
</tr>
</tbody>
</table>
Count Distribution Algorithm

Pass $k = 1$:

1. Processor $P_i$ scans over its data partition $D_i$; reads one tuple transaction (i.e. $(TID,X)$) at a time and building its local $C_{i1}$ and storing it in a hash-table (new entry is created if necessary).

2. At end of the pass every $P_i$ loads contents of into a buffer and sends it out to all other processors.

3. At the same time each $P_i$ receives the send buffer from another processor and increments the count value of every element in its local $C_{i1}$ hash-table if this element is present in the buffer otherwise a new entry would be created.

4. $P_i$ will now have the entire candidate set $C_1$ with global support counts for each candidate/element/itemset.

Step 2 and 3 require synchronization
## Count Distribution Algorithm Cont.

(Pass $K = 1$ Example)

<table>
<thead>
<tr>
<th>Processor/Node 1</th>
<th>Itemset</th>
<th>Support</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>{a}</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>{b}</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>{c}</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>{d}</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Processor/Node 2</th>
<th>Itemset</th>
<th>Support</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>{a}</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>{b}</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>{c}</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>{d}</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>{e}</td>
<td>6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Processor/Node 3</th>
<th>Itemset</th>
<th>Support</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>{a}</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>{b}</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>{c}</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>{d}</td>
<td>9</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Itemset</th>
<th>Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>{a}</td>
<td>22</td>
</tr>
<tr>
<td>{b}</td>
<td>8</td>
</tr>
<tr>
<td>{c}</td>
<td>12</td>
</tr>
<tr>
<td>{d}</td>
<td>14</td>
</tr>
<tr>
<td>{e}</td>
<td>6</td>
</tr>
</tbody>
</table>
Count Distribution Algorithm Cont.

Pass $k > 1$:

1. Every processor $P_i$ generates $C_k$ using frequent itemset $L_{k-1}$ created at pass $k - 1$
2. Processor $P_i$ goes over local database partition $D_i$ and develops local support count for candidates in $C_k$
3. Processor $P_i$ exchange local $C_k$ counts with all other processor to develop global $C_k$ counts. **Processors are forced to synchronize in this step.**
4. Each processor $P_i$ now computes $L_k$ from $C_k$.
5. Each processor $P_i$ decides to continue to next pass or terminate (The decision will be identical as the processors all have identical $L_k$).
Data Distribution Algorithm

• **Pass k = 1:** Same as the Count Distribution Algorithm
• **Pass k > 1:**

1. Processor \( P_i \) generates \( C_k \) from \( L_{k-1} \). Retaining only \( 1/N^{th} \) of the itemsets forming the candidates subset \( C_k^i \) that it will count. The \( C_k^i \) sets are all disjoint and the union of all \( C_k^i \) sets is the original \( C_k \).

2. Processor \( P_i \) develops support counts for the itemsets in its local candidate set \( C_k^i \) using both local data pages and data pages received from other processors.

3. At end of the pass, each processor \( P_i \) calculates \( L_k^i \) using the local \( C_k^i \). Again, all \( L_k^i \) sets are disjoint and the union of all \( L_k^i \) is \( L_k \).

4. Processors exchange \( L_k^i \) so that every processor has the complete \( L_k \) to generate \( C_{k+1} \) for next pass. **Processors are forced to synchronize in this step.**

5. Each processor \( P_i \) can independently (but identically) decide whether to terminate or continue.
Candidate Distribution Algorithm

Pass $k < m$: Use either Count or Data distribution algorithm.

Pass $k = m$:

1. Partition $L_{k-1}$ among the $N$ processors such that $L_{k-1}$ sets are “well balanced”. **Important:** For each itemset remember which processor was assigned to it.

2. Processor $P^i$ generates $C^i_k$ using only the $L_{k-1}$ partition assigned to it.

3. $P^i$ develops global counts for candidates in $C^i_k$ and the database is repartitioned into $DR^i$ at the same time.

4. After $P^i$ has processed local data and data received from other processors it posts $N - 1$ asynchronous receive buffer to receive $L^i_k$ from all other processors needed for the **pruning** $C^i_{k+1}$ in the prune step of candidate generation.

5. Processor $P^i$ computes $L^i_k$ from $C^i_k$ and asynchronously broadcasts it to the other $N - 1$ processors using $N - 1$ asynchronous sends.
Candidate Distribution Algorithm Cont.

Pass $k > m$:

1. Processor $P^i$ collects all frequent itemsets sent by other processors. They are used for the pruning step. Itemsets from some processor $j$ can be not of length $k - 1$ due to processors being fast or slow, but $P^i$ keeps track of the longest length of itemsets received for every single processor.

2. $P^i$ generates $C^i_k$ using local $L^i_{k-1}$. $P^i$ has to be careful during the pruning process as it could be that not all the $L^i_{k-1}$ from all other processors. So when examining if a candidate should be pruned it needs to go back to the pass $k = m$ and find out which processor was assigned to the current itemset when its length was $m - 1$ and check if $L^i_{k-1}$ has been received from this processor.

   (e.g. Let $m = 2$; $L_4 = \{abcd, abce, abde\}$ and we are looking at itemset $\{abcd\}$ then we have to go back to when the itemset was $\{ab\}$ (i.e. at pass $k = m$) to determine which processor was assigned to this itemset).

3. $P^i$ makes a pass over $DR^i$ and counts $C^i_k$. From $C^i_k$ computes $L^i_k$ and broadcast it to every other process via $N - 1$ asynchronous sends.
Pros and Cons of the Algorithms

- **Count Distribution**
  - **Pro:** Minimizes heavy data transfer between processors
  - **Con:** Redundant Candidate Set counting

- **Data Distribution**
  - **Pro:** Utilizes Aggregate Memory by assigning each processor a mutually exclusive subset of the Candidate set
  - **Con:** Requires good communication network (high bandwidth/low latency) due to large size of data needed to be broadcast at each pass

- **Candidate Distribution**
  - **Pro:** Maximizes use of aggregate memory while limiting communication to a single redistribution pass. Eliminates synchronization costs that Count and Data must pay at end of every pass
  - **Con:** (Post testing): it turns out the single redistribution pass takes its toll on the system
Looking Ahead

• Plan
  • Implement all three algorithm
  • Compare their performance (with each other; with sequential Apriori; with other sequential frequent pattern mining algorithms)
  • Find out synchronization capabilities of the MPI (Message Protocol Interface) in a multithreaded environment
  • Find out synchronization modifications needed of implementing the algorithms on a system that does not have a shared-nothing multiprocessor infrastructure.
Thank You!

Questions?