Dissertation Proposal

Incremental Frequent Pattern Mining
using Parallel Techniques:
MPI, CUDA, OpenMP, C++ AMP & Hadoop

Presenter: Mehmet Bicer,
Date: 09/18/2015

Committee Members:
Professor Susan Imberman, Co-Advisor
Professor Abdullah Uz Tansel, Co-Advisor
Professor Feng Gu
Introduction

When new data (increment) is added to the datasets, existing association rules gets updated: sometimes they become more significant, other times becomes outdated and less important. New data gets inserted, existing data gets deleted or updated.

It is inefficient to find all of the rules from scratch. Incremental data mining algorithms typically use previously discovered information to integrate new data and find new rules or update the previous ones.

Simply using more powerful computers, or even super-computers to mine large data sets is neither sufficient nor scalable. Instead, distributed storage and parallel computing using commodity computers is one of the cheaper and scalable solutions to handle incremental, large datasets.
Association Rule Mining

- “Given a set of transactions, find the rules that will predict the occurrence of an item based on the occurrences of other items in the transaction.” (Tan, Introduction to Data Mining, 2005)

- ARM tries to learn what items to be grouped together.

- It can tell us what sets of products (or itemsets) are frequently bought together.

- It is a probabilistic implication or co-occurrence, (not cause-effect relationship), between products. Given X, what is the probability of occurrence of Y. If then Else relationship.
Frequent Pattern Mining and Association Rule Mining (FPM-ARM)

Association rule mining is typically decoupled into:

**Frequent Pattern Mining:** It is a method to find frequent patterns, itemsets in a dataset. (R. Agrawal, 1993).

**Association Rule Mining:** It is a probabilistic implication or co-occurrence, (not cause-effect relationship), between products. Given X, what is the probability of occurrence of Y. If then Else relationship. i.e. $P(Y|X)$
FPM Basic Concept and Terminology

Let $I = \{i_1,..., i_d\}$ be a set of items in the dataset.

$T = \{t_1, t_2, t_3, ..., t_N\}$ // set of all transactions

$T_i$ contains subset of items from $I$

**Itemset:** one or more items in a set which are a subset of the items in $I$. If an itemset has $k$ items in it, then it is called $k$-itemset

**Support count:** $(\sigma)$: number of transactions which contain an itemset

e.g. $\sigma(\{1,3\}) = 2$

i.e. $\sigma(X) = |t_i|: X \subseteq t_i$ and $t_i \in T$

**Support** $(s)$: percentage of transactions which contains an itemset. e.g. $s(2/5) = 40\%$ where transaction count $N = 5$. 

A transaction database

<table>
<thead>
<tr>
<th>TID</th>
<th>Items purchased</th>
</tr>
</thead>
<tbody>
<tr>
<td>001</td>
<td>1, 3, 5</td>
</tr>
<tr>
<td>002</td>
<td>2, 3, 5</td>
</tr>
<tr>
<td>003</td>
<td>1, 4, 5</td>
</tr>
<tr>
<td>004</td>
<td>1, 3</td>
</tr>
<tr>
<td>005</td>
<td>2, 3, 4, 5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Itemset</th>
<th>Support Count $\sigma(X)$</th>
<th>Support % $s(\sigma(X)/N)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>60</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>40</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>80</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>40</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>80</td>
</tr>
</tbody>
</table>
Basic Concept and Terminology

Terms:

- Let $I = \{i_1, \ldots, i_d\}$ be a set of items in the dataset.
- A set $S = \{i_1, \ldots, i_k\} \subseteq I$ is called an itemset, or a $k$-itemset if it contains $k$ items.
- **Itemset**
- **$k$-itemset**
- **Support count** ($(\sigma(X)$
- **Support** (frequency of occurrence or relative support)
- **Confidence**

Transaction database

<table>
<thead>
<tr>
<th>TID</th>
<th>Items purchased</th>
</tr>
</thead>
<tbody>
<tr>
<td>001</td>
<td>1, 3, 5</td>
</tr>
<tr>
<td>002</td>
<td>2, 3, 5</td>
</tr>
<tr>
<td>003</td>
<td>1, 4, 5</td>
</tr>
<tr>
<td>004</td>
<td>1, 3</td>
</tr>
<tr>
<td>005</td>
<td>2, 3, 4, 5</td>
</tr>
</tbody>
</table>
Basic Concept and Terminology

**Candidate itemset:** Itemset which is not exposed to minimum support test yet or its support is not known yet.

**Frequent itemset:** an itemset which has support equal to or greater than the minimum support threshold

**Association rule** is \( \{X\} \rightarrow \{Y\} \) where both \( X \) and \( Y \) are itemsets, their intersection is empty set – i.e. \( X, Y \subseteq I, \text{ and } X \cap Y = \emptyset \) (disjoint sets)

**Strong Association rule:** if AR’s support is greater than minsup and confidence is greater than minconf.
Confidence (c): Confidence is the conditional probability $c$, such that a given transaction which contains $X$, will also contain $Y$. i.e. $P(Y|X)$.

- In the above example the association rule $\{1, 3\} \rightarrow \{5\}$ has a support count for $X \cup Y = 1$ and a support count for $X = 2$.
- Therefore, the confidence $c = \frac{\sigma(XUY)}{\sigma(X)} = 1/2 = 50\%$.

Confidence is not symmetric. Hence, $X \rightarrow Y$ and $Y \rightarrow X$ can have different confidence values.

In the above rule ($X \rightarrow Y$), the confidence $= 50\%$ but if we look at the rule $Y \rightarrow X$, then $XUY/Y$ is $1/4 = 25\%$.

<table>
<thead>
<tr>
<th>TID</th>
<th>Items purchased</th>
</tr>
</thead>
<tbody>
<tr>
<td>001</td>
<td>1, 3, 5</td>
</tr>
<tr>
<td>002</td>
<td>2, 3, 5</td>
</tr>
<tr>
<td>003</td>
<td>1, 4, 5</td>
</tr>
<tr>
<td>004</td>
<td>1, 3</td>
</tr>
<tr>
<td>005</td>
<td>2, 3, 4, 5</td>
</tr>
</tbody>
</table>
Basic Concept and Terminology

- **Minsup**: a user defined threshold which states the minimum support for an itemset to show that it is significant.

- **Minconf**: a user defined threshold which states the minimum confidence value needed to show that an association rule is reliable.

if minconf = 45% then while X->Y is a confident rule as it has the confidence 50%, Y->X won’t be a confident rule because its confidence is 25% which is below the minconf threshold.

<table>
<thead>
<tr>
<th>TID</th>
<th>Items purchased</th>
</tr>
</thead>
<tbody>
<tr>
<td>001</td>
<td>1, 3, 5</td>
</tr>
<tr>
<td>002</td>
<td>2, 3, 5</td>
</tr>
<tr>
<td>003</td>
<td>1, 4, 5</td>
</tr>
<tr>
<td>004</td>
<td>1, 3</td>
</tr>
<tr>
<td>005</td>
<td>2, 3, 4, 5</td>
</tr>
</tbody>
</table>
Association Rules

Association rules for itemset: \{1,3,5\}

\{1,3\} \rightarrow \{5\} (s= 20\% \ c = 20\%/40\%=50\%)
\{1,5\} \rightarrow \{3\} (s= 20\% \ c = 20\%/40\%=50\%)
\{3,5\} \rightarrow \{1\} (s= 20\% \ c = 20\%/60\%=33.3\%)
\{1\} \rightarrow \{3,5\} (s= 20\% \ c = 20\%/60\%=33.3\%)
\{3\} \rightarrow \{1,5\} (s= 20\% \ c = 20\%/80\%=25\%)
\{5\} \rightarrow \{1,3\} (s= 20\% \ c = 20\%/80\%=25\%)

If we are creating the rules from the same itemset, then, all rules will have the same support but may have different confidence as seen above.

Typically ARM is divided into two separate problems. FPM and AR generations.
Why We Use Support and Confidence:

- Support eliminates uninteresting rules which may simply occur accidentally.
- Confidence tells us how reliable the inferred rules are.
How many frequent patterns?
FPM: Brute force approach:

• $2^d - 1$ possible candidate itemsets. (it is a combinatorial problem). E.g. if we have 5 items, we can possibly have $2^5 - 1 = 31$ frequent itemsets.

• The number of rules can be calculated with the following formula: $R = 3^d - 2^{d+1} + 1$

  Given $d = 5$
  
  $R = 3^5 - 2^{5+1} + 1$
  
  $= 243 - 64 + 1$

  There could be 180 association rules.

for 100 items there will be $5.15 \times 10^{48}$ many rules.

For a typical supermarket with 38,718 items that will be of $10^{1847}$ rules.

(Tan, Introduction to Data Mining, Instructor's manual, 2005).
How many frequent patterns?

- Let \( N \) = Number of transactions,
- \( d \) = Number of items in the dataset
- \( M \) = Number of Candidate Itemsets. \( M = 2^d \)
- \( w \) = width
- \( \text{NM} \) = Number of comparisons
- \( R \) = Number of rules

(Tan, Introduction to Data Mining, 2005).

- That has a complexity of \( O(\text{NM}w) \).
  For the above example, that is \( O(5 \times 64 \times 5) = 1600 \) comparisons
- \( R = 3^d - 2^{d+1} + 1 \) rules
How can we reduce the FPM complexity?

1. Minimize the number of candidates generated, through Apriori principle or downward closure property. i.e. make search space much smaller.

2. Don’t count more than need to.

3. Don’t count the support of itemsets which are infrequent or will definitely become infrequent

4. Minimize the number of comparisons
Frequent Pattern Mining and Association Rule Mining (FPM-ARM)

Association rule mining is typically decoupled into:

**Frequent Pattern Mining:** It is a method to find frequent patterns, itemsets in a dataset. (R. Agrawal, 1993).

**Association Rule Mining:** It is a probabilistic implication or co-occurrence, (not cause-effect relationship), between products. Given X, what is the probability of occurrence of Y. If then Else relationship. i.e. $P(Y|X)$
Frequent Pattern Mining (FPM) Algorithms

- Major Algorithms
- Apriori
- Eclat
- FP-Growth
Apriori Algorithm – Key points
(Agrawal and Srikant 1994 and Mannila et al. 1994)

- Level-wise
- Iterative
- Bottom up (from 1-itemset to largest sets)
- Breadth-first (BFS) algorithm with candidate generation and join steps.

It requires one full scan of the database for each level. Given that the set of k-itemset can be grouped into k levels, we would need k database scans.
Apriori – Basic Idea

First find the frequent 1-itemsets. From the frequent 1-itemsets create candidate 2-itemsets and prune the ones which are infrequent 2-itemsets. From frequent 2-itemsets create candidate 3-itemsets and so on.

Apriori prunes the infrequent items immediately after it discovers them, before it moves to the next step. It then creates new level of itemsets only from those which are frequent.

* Assumptions: Apriori assumes that items in itemsets are in lexicographic order.
Apriori principle, Downward-closure property (anti monotonicity)

- Apriori (pruning) principle:
  - It is based on the following observation:
  - If \( A \) is a subset of \( B \) then, support of \( B \) cannot exceed that of \( A \).
  - More formally:

    In other words, support count of a superset of an itemset cannot be greater than its subsets.

Based on this rule, we have the following:

**Downward-closure property (anti monotonicity)**

- If an itemset is frequent, all of its subsets must be frequent. i.e. If an itemset has infrequent subsets, it cannot be frequent.
- If an itemset is infrequent all of its supersets are also infrequent.

(Tan, Introduction to Data Mining, 2005) (J. Han, 2005)
Subset of a frequent itemset is also frequent

Figure 4: Example shows that subsets of a frequent itemset are all frequent.
Supersets of an infrequent itemset are also infrequent – so, there is no need to scan the database and check their count.

Figure 5: Example shows that supersets of an infrequent itemset are all infrequent too.
Candidate generation

**Join step:** In order to find frequent (k+1)-itemsets, all of the frequent k-itemsets, which have the first (k-1)-elements have to be in common (joinable property), are joined with each other (called self join), to create k+1 itemsets (for k > 1 itemsets). The join steps ensures no duplicate candidate itemsets are created. It is assumed that the itemsets are in numerical order (for countable data types) or lexicographic order (for strings).
Prune step

After the join step of subsets of each candidate itemset is checked in the frequent itemset list for membership. If all of the subsets are not in the frequent itemset list for a given candidate (k+1)-itemset, then that itemset is removed from the candidate (k+1)-itemset list, otherwise it is inserted in the frequent itemset list.

– i.e. if any (k-1)-itemset that is not frequent cannot be a subset of a frequent k-itemset. So, if any (k-1)-itemset is not in Lk-1, then the candidate can not be frequent either and so it can be removed from Ck.

· Pruning algorithm
  · “forall itemsets c in C_k do
  · forall (k-1)-subsets s of c do
  · if (s is not in L_{k-1}) then delete c from C_k” (J. Han, 2005)
Table 7: A working example of Apriori algorithm.
How infrequent itemsets change possible large itemset space

Figure 6: Example transactions and how Apriori principles are applied to data.\(^5\)
Apriori Example Execution

- **Iteration:** 1
  - Candidates:
    - {1}
    - {2}
    - {3}
    - {4}
    - {5}
  - Extracted Large Itemsets:
    - {1}
    - {2}
    - {3}
    - {4}

<table>
<thead>
<tr>
<th>Trans. ID</th>
<th>Items Bought</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>1,2,3</td>
</tr>
<tr>
<td>20</td>
<td>2,4</td>
</tr>
<tr>
<td>30</td>
<td>3,4</td>
</tr>
<tr>
<td>40</td>
<td>1,2,5</td>
</tr>
<tr>
<td>50</td>
<td>1,2,3,4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Items</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
</tr>
</tbody>
</table>

**Final Results:** (Frequent Itemsets)

<table>
<thead>
<tr>
<th>Items</th>
</tr>
</thead>
<tbody>
<tr>
<td>{1}</td>
</tr>
<tr>
<td>{2}</td>
</tr>
<tr>
<td>{3}</td>
</tr>
<tr>
<td>{4}</td>
</tr>
<tr>
<td>{1,2}</td>
</tr>
</tbody>
</table>

\[\text{minsups} = 50\%\]
Notes on Apriori:

- Apriori is an easy to implement, effective candidate pruning algorithm. To work efficiently, the dataset needs to fit in memory.

- What are the issues with Apriori?

  - The database needs to be scanned for every level. If there is \( m \)-frequent-itemsets, then it needs to scan database \( m \) times.

  - There are too many candidates that are generated which need to be tested.
Extensions of Apriori

• **Hashing technique** (An Effective Hash-Based Algorithm for Mining Association Rules): This algorithm successfully reduces overall number of candidates generated by specifically applying a hash-based algorithm on 2-itemsets and reduced the whole candidates in an earlier stage. (J. S. Park M.-S. C., 1995)

• **Dynamic Itemset counting or DIC** dynamically counts the candidate itemsets as the algorithm continues and reduces the candidates generated. In other words DIC algorithm **doesn’t wait for database scan to be completed** to create the candidates, instead it **looks at an itemset’s subsets** and determines whether they all be frequent. When DIC determines that all of the subsets of an itemset is frequent or estimated to be frequent, it adds the itemset to the candidate itemset list and starts counting support for the it. This process reduces the number of scans. (S. Brin, 1997)
Extensions of Apriori

**Partition:** Partitioning algorithm scans the database only **twice** hence reduces the I/O substantially. In the first scan it partitions the whole database into smaller vertical database pieces which are **small enough to fit in memory.** Then it finds the frequent itemsets found in each partition. In order for an itemset to be frequent, **it has to be frequent at least in one of the partitions.** [Savasere et al., 1995]
Extension to Apriori

**Sampling:** The Sampling algorithm finds the frequent itemsets from *randomly selected samples*. It verifies these results in the whole database and creates the complete association rules for these verified frequent itemsets.

A very low threshold is used but this creates too many candidates. The maximum number of database scans in the Sampling algorithm is 2. One to get the samples and verify the results, then if there are missing frequent itemsets, these are found and verified in the second pass. (Toivonen, 1996)
Apriori Extensions

DHP: DHP is a hash-based, Apriori-based algorithm which utilizes pruning techniques as well as approximate counts to generate much fewer candidates, thus shrinking the database or transaction data size. Even though it generates candidates very efficiently the number of database scans are equal to the number of levels. (J. S. Park M. C., 1995)
Eclat (Zaki M. J., 1997)

- **Basic idea:**
  - Eclat uses a *vertical* database layout. It scans the database and builds Transaction ID list (**TIDlist**) for each item. From these, for each frequent single item, (k-itemset where k = 1), create a Transaction ID set for (k+1) itemsets by taking the **intersection** between two itemsets. Eclat partitions the frequent itemset list into equivalence classes and obtains the support count from the number of intersections.
  - ECLAT doesn’t need another database scan for support count because each **TIDlist** has complete information about the database. (Zaki., 2000)
Major features of ECLAT

Recursive, depth-first search (DFS) algorithm, uses tidlists, intersections, a divide-and-conquer algorithm.
Eclat Example

Table 8: ECLAT process.
FP-Growth Algorithm

- **Basic idea:** FP-Growth algorithm finds all frequent itemsets without candidate generation. It first scans the database, finding all frequent 1-itemsets. It then sorts them in descending order. Using this order, it sorts transactions only with frequent items, in descending order.

- FP - Growth then reads the ordered database a second time, and builds an FP-tree. The FP-tree has a null root on the top, and at this stage, only frequent 1-itemsets as nodes. Any subsequent transactions are stored on the tree with complete number of occurrence and association information. Each transaction is a path from the root thus preserving the descending order in the tree.
FP Growth Highlights

- FP-growth algorithm works as a divide and conquer method, recursively without candidate generation. FP-Tree efficiency:

- Whole dataset gets compressed into a tree-format (fp-tree) which contains complete information about all of the transactions.

- No candidate generation. Instead, a pattern fragment growth method is used. This method starts from frequent 1-itemset

- Mining task is divided into smaller task (hence, recursive DFS algorithm) which uses partition-based divide and conquer method to mine association rules.
FP Growth - Example

Example:

<table>
<thead>
<tr>
<th>Trans. ID</th>
<th>Items Bought</th>
<th>Frequent Items (Ordered)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1,2,3,5</td>
<td>2, 1,3</td>
</tr>
<tr>
<td>2</td>
<td>2,4</td>
<td>2,4</td>
</tr>
<tr>
<td>3</td>
<td>4,3</td>
<td>3,4</td>
</tr>
<tr>
<td>4</td>
<td>1,2,5</td>
<td>2,1</td>
</tr>
<tr>
<td>5</td>
<td>1,2,3,4</td>
<td>2,1,3,4</td>
</tr>
</tbody>
</table>

*minimum support =3

Table 9: A table from the transaction database with 5 transactions and 5 items

Frequent Items (in descending order): 2:4, 1:3, 3:3, 4:3

Header Table

<table>
<thead>
<tr>
<th>Item</th>
<th>Head of Node-links</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

Figure 8: FP-Growth table and FP-Tree.
Notes on FP-Growth

Performance study have shown that FP-growth is an order of magnitude faster than Apriori, and is also faster than tree-projection. This is because there is no candidate generation and no candidate test. It uses—Use compact data structure—Eliminate repeated database scan—Basic operation is counting and FP-tree building
Extension of FP-Growth

• Tree-Projection: This algorithm uses much less memory through the use of a lexicographical, top down tree instead of using a hash tree. Itemsets are counted through projecting the transactions onto the nodes of the tree which increases the performance of counting the itemsets in the transactions. (R. C. Agarwal, 2000)

• Relim: The algorithm uses an idea similar to FP-growth via a single recursive function. It is similar to the H-mine algorithm. It has complicated data structures. (Borgelt, 2005)
Maximal Itemset Mining (Max-patterns)

When association rules involve many items, generating rules for all of the subsets becomes a combinatorial problem and not very practical. Hence, some frequent itemset algorithms focus instead on finding maximal frequent itemsets. [Bayordo, 1998]

- A frequent itemset is maximal if it doesn’t have a superset which is frequent. In other words, a maximal frequent itemset is not a subset of any frequent itemset. (Zaki., 2000)
- MAFIA (D. Burdick, 2001) is one of the maximal frequent itemset mining algorithms. It works on transactional databases. It is a dept first search algorithm as well as using efficient pruning techniques. It is particularly more efficient if the itemsets are very long in the dataset.
- In the example in table 8, itemset \{1, 2, 3\} is a maximal frequent itemset.
An itemset is closed if there is no superset of this itemset which has the same support as the support of this itemset.

Algorithms:

COBBLER: Dynamically decides how to proceed according to whether the dataset to be mined has very large number of attributes and relatively small number of rows or vice versa. (Pan F, 2004)

- Recursive, DFS algorithm
Closed Pattern Mining Algorithms -2

- CLOSET
  - uses compressed FP tree structure
  - recursive divide and conquer strategy
  - prefix-based compression technique and partition-based database projection approach. (J. Pei J. M., 2000)
CHARM

- A frequent closed pattern mining algorithm
- Uses DFS
- Good for data with small number of features
- Uses diffset (difference of two tidsets) to minimize memory usage
- CHARM uses a hash based technique to find the non closed itemsets and when it finds them, removes them immediately. (M. J. Zaki, 2002)
Why Incremental Algorithms Needed?

When new data is added to the existing data we have two options:

1) Run the FIM algorithms on the updated data
2) Use an incremental mining technique.

Running the algorithm for the entire, updated dataset will not utilize the existing rules which were captured previously and hence wasting the computation made previously and it will take longer time to process the whole data instead of much smaller increment.
Why Parallel and Distributed Model May be Needed?

• Addition of very large amount of data into existing data storages continuously

• A typical traditional approach would be to use more powerful computers, or even super-computers to handle the problem but this solution is very expensive and not scalable.

• Parallel and distributed model makes both storage and computation possible and it is also scalable.

• New approach is using commodity hardware in a distributed data storage environment and implement parallel computation using Map Reduce.
Incremental Algorithms

Some of the efficient incremental algorithms:

Apriori Based Approaches:

- FUP
- FUP2
- UWEP
- NUWEP
- NFUP
- MAAP
Incremental Algorithms (cont.)

Tree based approaches:

DT-Tree and PoFP-Tree
FELINE (based on CATS Tree)
CAN Tree
Bucket Sort Approach
DHP, MIP, & IMPHP algorithms
UWEP

1) Prune an itemset immediately after it is understood to be small in the updated db (i.e. the increment).

2) Keep the candidate set as small as possible.

* Support count or minsup for DB and db are assumed to be the same.
# How Incremental Data Comes

<table>
<thead>
<tr>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1/1/11</td>
</tr>
<tr>
<td>1/8/11</td>
</tr>
<tr>
<td>1/15/11</td>
</tr>
</tbody>
</table>

\[ \text{DB} \rightarrow \text{db} \rightarrow (\text{DB} + \text{db}) + \text{db}((\text{DB} + \text{db}) + \text{db}) + \text{db} \cdots \]
Algorithm UWEP has the following 5 steps:

1. Counting 1-itemsets in db and creating a tidlist for each item in db
2. Checking the large itemsets in DB whose items are absent in db and their supersets for largeness in DB+db
3. Checking the large itemsets in db for largeness in DB+db
4. Checking the large itemsets in DB that are not counted over db for largeness in DB+db
5. Generating the candidate set from the set of large itemsets obtained in the previous step.
Apriori vs UWEP

Apriori

Large Itemsets

Apriori (DB) \( LI_{DB} = \{\ldots\} \)

Apriori (DB + db) \( LI_{DB+db} = \{\ldots\} \)

UWEP

Large Itemsets

Apriori (DB) \( LI_{DB} = \{\ldots\} \)

Apriori (db) \( LI_{db} = \{\ldots\} \)  
\text{Merge the results.}
We need to know whether DB and db are Large (L) or small (S)

\[ LI_{DB} \quad LI_{db} \quad LI_{DB+db} \]

L L L //e.g. if 50/n > minsup AND 20/n’>minsup
then, we can conclude that \((50+20)/(n +n’)\)

L S ? //We can not tell. We need more info.

S L ? //We can not tell. We need more info.

S S S //e.g. if 50/n > minsup AND 20/n’>minsup
then, we can conclude that \((50+20)/(n +n’)\)

i.e. \((50/n + 20/n’) > minsup\)
UWEP Example

Example: let’s say n = 1000 and n’ = 100, minsup = 5%.

Then, we know that for DB we need support count $\geq 50$ and for $db$ it is $\geq 5$, i.e. $55 \geq 1100$

Assume $L_{DB} = 50$ $L_{db} = 4$, i.e. $L_{DB} + db = 54$ //small (S)

Assume $L_{DB} = 58$ $L_{db} = 4$, i.e. $L_{DB} + db = 62$ //large (L)

What is needed?

1. $LI_{DB}$ keep support count of the items in $L_{DB}$
2. $L_{DB}$ $L_{1_{db}}$
3. $S^{(x)}$ $L^{(10)}$ keep TID list for each item. $X$ is calculated from TID list in DB for the items in $L_{db}$.

s s s s
How to implement UWEP

Given DB, $L_{DB}$ with support count,

Step 1:
Apply apriori on db, get $L_{db}$ with support count and TID list

Step 2:
Combine Given info + Step 1
How to implement UWEP

\[ L_{DB} \quad L_{db} \quad L_{DB} + L_{db} \]

\[ L \quad X^{(k)} \quad X^{(m)} \quad X^{(k+m)} \quad \text{support} \]

? \quad L,X \quad S,X \quad \text{use TID list in db to calculate support of } X

? \quad S,X \quad L,X \quad \text{use TID list in DB to calculate support of } X

S \quad S,X \quad S,X
Finding the support count from TIDlist for itemset X

Suppose X is ABC
Then, support (ABC) = |TID (A) \cap TID (B) \cap TID(C)|

C_{db}^{1} = all 1-itemsets in db with support > 0

Prune set = L_{DB}^{1} - C_{db}^{1}

e.g. \{A,B,C,D,E,F\} – \{D,E,F\} = \{A,B,C\} is in L in DB
Is S in db (with support =0)
Parallel UWEP

Approach 1:

MapReduce Approach:

In a shared nothing architecture where nodes are not even aware of each other, create Tidlists as Trie data structure in this distributed memory system, let mapreduce handle underlying details automatically:

Parallelize initial pruning counts as well as the main logic of the program until both Candidate and DB large itemset is empty.

Parallelize using Mapreduce framework to do all the support counts:

i.e. check whether an Itemset is small or large in DB, db or DB+db
Parallel UWEP

Approach 2:

OpenMP:

Use shared memory framework.

Create and keep Tidlist using Trie or Prefix tree data structure for DB, db and DB+db in this shared memory

Parallelize all the support counts of candidate Itemsets at each level.
# Incremental Algorithm Comparison

<table>
<thead>
<tr>
<th></th>
<th>Algorithm Name</th>
<th>Based on</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>FUP</td>
<td>Apriori Based</td>
<td>Poor</td>
</tr>
<tr>
<td>2</td>
<td>FUP2</td>
<td>Candidate Separation</td>
<td>Poor</td>
</tr>
<tr>
<td>3</td>
<td>UWEP</td>
<td>Look Ahead pruning</td>
<td>Average</td>
</tr>
<tr>
<td>4</td>
<td>MAAP</td>
<td>Efficient low level item generation via starting from Large itemsets</td>
<td>Average</td>
</tr>
<tr>
<td>5</td>
<td>CATS TREE</td>
<td>Compact tree</td>
<td>Good</td>
</tr>
<tr>
<td>6</td>
<td>CAN TREE</td>
<td>Canonical tree</td>
<td>Good</td>
</tr>
<tr>
<td>7</td>
<td>Bucket Sort Approach</td>
<td>Storage Buckets</td>
<td>Good</td>
</tr>
<tr>
<td>8</td>
<td>DHP and MPIP</td>
<td>Perfect Hashing</td>
<td>Better</td>
</tr>
</tbody>
</table>
Incremental FPM Using Parallel Computation

Approaches:

1. Use Supercomputers with thousands of cores and Petabytes of data storage. Write the code in one of the languages which is available for HPC center supercomputers. Use MPI to manage data as well as code in parallel.

2. Use Hadoop eco-system and map reduce computation approach to do itemset counting and summarization.

3. Use a game computer with a GPU processor with hundreds of cores and write CUDA programs to parallelize code
Parallel Techniques

- MPI
- CUDA & GPGPU
- OpenMP
- C++ AMP
- OPEN CL
- OpenACC
- Hadoop & MapReduce
Hadoop (Big data) and its Eco System Terminology

- Hadoop
- HDFS
- MapReduce
- Namenode
- DataNode
- Job Tracker
- Task Tracker
- Resource Manager
- Impala
- Hive
- Pig
- Flume,
- Hue
- Oozie
- Hbase
- Scoop
- Mahout
- Spark
Below is a simplified representation of the data flow for Word Count Example.

MapReduce – Word Count Example Flow

Taken from (https://www.mssqltips.com/sqlservertip/3222/big-data-basics--part-5--introduction-to-mapreduce/) (Sindol, 2014)
Issues in Parallel programming
Environment:

- Memory (distributed?, shared?)
- Data storage (distributed, local)
- Computation (serial, parallel, hybrid)
- Communication:
  - Redundant computation:
  - Replication:
- Bandwidth:
- Complexity:
- Task scheduling,
- Load balance
Parallel Algorithms

Earlier Approaches:

Count distribution algorithm
Data distribution algorithm
Candidate distribution algorithm
Count distribution algorithm

Each node has the entire candidate Itemsets
Each node has the proportional size of the data set
Each node only exchanges counts
Each node has the same large Itemsets after each pass and synchronization

Pros:
Low communication costs
Linearly scales up and speeds up

Cons:
Doesn't utilize the total, aggregated memory of all nodes
Synchronize after each pass
Data Distribution Algorithm

Candidates are divided evenly for each node.

Each node has its local data and broadcast this data for support count to all of the other nodes after each pass.

**Pros:**

Better utilization of Aggregate memory

**Cons:**

Need to synchronize after each pass
Broadcasting local actual data
Very high communication costs, require large bandwidth
Candidate Distribution Algorithm

Partial Candidate sets,
Partitioned and Partial large Itemsets

Pros:
Nodes can work independently
No synchronization is required after each pass

Cons:
One time data redistribution of the entire data
Parallel Algorithms

Newer Algorithms:

- A Parallel Implementation of Apriori Algorithm using OpenMP
- A MapReduce-Based Approach
- YAFIM- A Spark Based approached
Proposal

In this proposal we will explore and devise incremental algorithms with one of the parallel computation techniques such as MapReduce which may also include other Hadoop Eco-System tools (such as Spark), OpenMP or CUDA or a hybrid implementation. We also plan to use the devised algorithms in exploring emerging patterns in data accumulated over periods of time in increments.
Q & A
Thank You!