CHAPTER 4

CONSTRAINED FIM IN BIG DATA

4.1 Need for Constrained Pattern Mining

FPM usually produces too many solution patterns. This situation is harmful for two reasons: 1. Performance: mining is usually inefficient or often simply unfeasible, 2. Identification of fragments of interesting knowledge: which is blurred within a huge quantity of small, mostly useless patterns. So users need an effective way to control the large number of discovered patterns, and to be able to choose what patterns to consider at each time. The most accepted and common approach to minimize these drawbacks is to capture and represent the semantics of the domain through constraints, and use them not only to reduce the number of results, but also to focus the algorithms in areas where it is more likely to gain information and return more interesting results. So Constraints can be used which gives the following benefits:

1. They can be pushed in the frequent pattern computation exploiting them in pruning the search space, thus reducing time and resource requirements.
2. They provide guidance to the user over the mining process and a way of focusing on interesting knowledge.

4.2 Types of Constraints

4.2.1 Classification of constraints based on semantics

Item Constraint:

An item constraint specifies what are the particular individual or group
of items that should not be present in pattern. For example a soap company may be interested in patterns containing only soap products, when it mines transactions in a grocery store.

Length Constraint:

A length constraint specifies the requirement on the length of patterns, i.e., the number of items in the patterns. For example, when mining classification rules for the document, a user may be interested in only frequent patterns with at least five keywords.

Model-based constraint:

A Model-based constraint looks for patterns which are sub or super patterns of some given patterns (models). For example, a car dealer may be interested in knowing what are all the other accessory items a purchaser would buy when he buys a car.

Aggregate Constraint:

An Aggregate constraint is on an aggregate of items in a pattern, where the aggregate function can be \( \text{SUM, AVG, MAX, MIN} \), etc. For example, a marketing analyst may like to find pattern where the average price is over $150.

User Constraint:

User constraints are those in which user can use a rich set of SQL-style Constraints, to guide the mining process to find only those Frequent Patterns containing market basket items that satisfy the user constraints. Examples of these constraints include the following:
Here, constraint \( C1 \) says that the minimum price of all items in a pattern/set \( S \) is at least $10; constraint \( C2 \) says that all items in a pattern \( S \) are snack. It is important to note that, besides these market basket items, the set of constraints can also be imposed on individuals, events, or objects in other domains. The following are some examples: \( C3 \) \( \equiv \max(S.\text{Temperature}) \leq 38^{\circ}\text{C} \). This constraint says that the maximum (body) temperature of all individuals in a pattern/set \( S \) must be at most \( 38^{\circ}\text{C} \). \( C4 \) \( \equiv \min(S.\text{Price}) \leq 1000 \), and \( C5 \) \( \equiv \text{avg}(S.\text{Price}) \leq 1000 \). Constraints \( C4 \) and \( C5 \), respectively, say that the minimum and the average price of all items in \( S \) is at most $1000.

### 4.2.2 Classification of constraints based on properties

**Monotonicity:**

When an itemset \( S \) satisfies the constraint, so does any of its superset.

- \( \text{Sum}(S.\text{Price}) \geq v \) is monotone
- \( \text{Min}(S.\text{Price}) \leq v \) is monotone

**Anti-monotonicity:**

When an itemset \( S \) satisfies the constraint, so does any of its subset

- Frequency is an anti-monotone constraint.

**Succinctness:**

Given \( A_f \), the set of items satisfying a succinct constraint \( C \), then any set \( S \) satisfying \( C \) is based on \( A_f \), i.e., \( S \) contains a subset belonging to \( A_f \).

Idea: whether an itemset \( S \) satisfies constraint \( C \) can be determined based on the singleton items which are in \( S \).

- \( \min(S.\text{Price}) \leq v \) is succinct
sum(S.Price) ≥ v is not succinct

Convertible anti-monotone:

If an itemset S violates a constraint C, so does every itemset having S as a prefix w.r.t. R. Let R be the order of items.

Ex. avg(S) ≤ v, w.r.t. item value descending order

Convertible monotone:

If an itemset S satisfies constraint C, so does every itemset having S as a prefix w.r.t. R. Let R be the order of items.

Ex. avg(S) ≥ v, w.r.t. item value descending order

4.3 Constrained FIM

In constrained FIM, first constraints are applied to the database, thereby the search space can be reduced. For Example, consider the Table 4.1

<table>
<thead>
<tr>
<th>Item</th>
<th>H</th>
<th>I</th>
<th>J</th>
<th>K</th>
<th>L</th>
<th>M</th>
</tr>
</thead>
<tbody>
<tr>
<td>Qty</td>
<td>500</td>
<td>455</td>
<td>700</td>
<td>600</td>
<td>300</td>
<td>200</td>
</tr>
<tr>
<td>Price</td>
<td>35</td>
<td>45</td>
<td>55</td>
<td>65</td>
<td>25</td>
<td>70</td>
</tr>
</tbody>
</table>

Let S be the set of itemsets which satisfy the constraints

Q1 : min(S.Qty) ≥ 400 (Anti-monotone, succinct)
Q2: max(S.Price) ≥ 40 (Monotone and succinct)
Q3: min(S.Qty) ≥ 400 ∧ max(S.Price) ≥ 40 (Anti-Monotone, Succinct, Monotone)
Q3 is to find all frequent itemsets satisfying Q1(X) ∧ Q2(X).
Let us consider the above constraints to mine Frequent Itemsets from the database which is shown in table 4.2.

**Table 4.2 Transactional database 1**

<table>
<thead>
<tr>
<th>Transactions (T)</th>
<th>Itemsets</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>{H,I,J}</td>
</tr>
<tr>
<td>T2</td>
<td>{K,J}</td>
</tr>
<tr>
<td>T3</td>
<td>{H,I,K,L}</td>
</tr>
<tr>
<td>T4</td>
<td>{H,I,K,M}</td>
</tr>
<tr>
<td>T5</td>
<td>{K,M}</td>
</tr>
</tbody>
</table>

**Q1 : min(S Qty) ≥ 400**

To find frequent itemsets which satisfies the above constraint, first the algorithm finds the frequent 1-itemsets. Let the minimum support threshold be 2. In the above transactional database table, only ‘L’ has support count as 1. So L is removed from the database. In each transactions, the items are arranged in the descending order of the Qty value. Next step is to identify the invalid items. ‘M’ is invalid since its qty value is 200. So M is also excluded. Now FP-Tree is built using only valid items. Eg. H, I, J, K. So a projected database is formed with respect to H, I, J, K and frequent itemsets are generated by applying FP growth.

**Q2: max(S Price) ≥ 40**

For this type of constraint, first the algorithm identifies the items whose price is greater than or equal to 40. In the above database I, J, K, M has price greater than or equal to 40. H and L’s price is less than 40. In this, L is infrequent and it is removed. I, J, K, M forms a primary group and H forms the secondary group. The FP-Tree is formed. Here a dashed line has to be used to form a boundary between primary and secondary items in the initial FP-Tree. From the
initial FP-Tree, projected database is formed for each valid singleton items. Here the boundary is not required. This means, once a primary item $x$ is found for a valid itemset $v$, any other item in $v$ can be chosen from the primary or secondary group. From the projected FP-Tree frequent itemsets are generated by applying FP growth.

**Q3: $\min(S.Qty) \geq 400 \land \max(S.Price) \geq 40$**

When using multiple constraints, the algorithm first picks the most selective constraint by ignoring another constraint during the mining process. The mined itemsets can be checked against the previously ignored constraint when generating the conditional database and then frequent itemsets are generated. In the above example $Q1 \land Q2$, the primary group can be items whose Qty values are $\geq 400$ and whose price is $\geq 35$. Secondary group contains items whose Qty values are $\geq 500$.

### 4.4 FIM in Big Data

FP growth for FIM do not hold good to handle large volume of data. Since the FP-Tree is considered to be the best compact data structure to hold the data patterns in memory there has been efforts to make it parallel and distributed to handle large databases. However, it incurs a lot of communication overhead during the mining. Recent improvements in parallel programming have provided good tools to handle this problem. So a parallel and distributed FIM algorithm using the Hadoop Map Reduce framework has been proposed, which shows best performance results for large databases.
4.4.1 Introduction to Hadoop and MapReduce

Hadoop:

Hadoop is an open-source software framework for processing of very large data sets in a distributed environment. This software package provides tools for data discovery, answering questions and solving analytic problems in Big Data. It consists of a collection of servers called clusters, which runs the Hadoop software and individual servers within a cluster called nodes. Components of the Hadoop ecosystem are shown in Figure 4.1.
Hadoop Ecosystem comprises of the following key components:

(i) **HBase**: It is an open, distributed and Non-relational database system implemented in Java. It runs above the layer of HDFS. It can serve the input and output for the MapReduce in well-mannered structure.

(ii) **Map Reduce**: Map-Reduce was introduced by Google in order to process and store large data sets on commodity hardware. Map Reduce is a model for processing large-scale data records in clusters.

(iii) **Oozie**: Oozie is a web-application that runs in a Java servlet. Oozie uses the database to gather the information of workflow, which is a collections of actions. It manages the Hadoop jobs in a mannered way.

(iv) **Sqoop**: Sqoop is a command-line interface application that provides platform which is used for converting data from relational databases and Hadoop or vice versa.

(v) **Avro**: It is a system that provides the functionality of data serialization and service of data exchange. It is basically used in Apache Hadoop. These services can be used together as well as independently according to the data records.

(vi) **Chukwa**: Chukwa is a framework that is used for data collection and analysis to process and analyze the massive amount of logs. It is built on the upper layer of the HDFS and Map Reduce framework.

(vii) **Pig**: Pig is a high-level platform where the MapReduce framework is created which is used with Hadoop platform. It is a high-level data processing system where the data records are analyzed that occurs in high level language.
(viii) **Zookeeper**: It is a centralized base service that provides distributed synchronization and provides group services along with maintenance of the configuration information and records.

(ix) **Hive**: It is an application developed for a data warehouse that provides the SQL interface as well as the relational model. Hive infrastructure is built on the top layer of Hadoop that help in providing a conclusion, and analysis for respective queries. The main building blocks of Hive are:

- Metastore which is used to store catalogue and metadata.
- Query Compiler compiles HiveQL for MapReduce tasks.
- Execution Engine executes the tasks produced by the compiler in proper dependency order.
- HiveServer and a JDBC/ODBC server.

(x) **HDFS**: Hadoop provides a distributed file system (HDFS) and a framework for analysis and transformation of very large datasets using the MapReduce paradigm.

Thus the main objective of Hadoop is not to speed up the processing of data but to make it possible to process really huge amount of data by splitting these data into smaller subsets of data. Hadoop is flexible to any data formats and it can run on low cost commodity hardware. It protects the data being lost because of the hardware failures by taking multiple copies automatically.

**HDFS**:

Hadoop employs a master/slave architecture for both distributed storage and computation. A multi-node cluster running Hadoop means running a set of daemons or resident programs, on the different servers in the network which is shown in Figure 4.2.
The daemons include:

• **NameNode**: is the master of HDFS that directs the slave DataNodes daemons. It keeps track of the overall health of the distributed file system. The function of NameNode is memory and I/O intensive as a result of which it does not store any user data or perform any computations in a MapReduce program to reduce the workload on the machine. There is a negative aspect of the importance of the NameNode - it is a single point of failure.

• **DataNode**: is responsible for reading and writing HDFS blocks to actual files on the local file system. It constantly informs the NameNode about the blocks it
is currently storing and also provides information about local changes as well as receives instructions to create, edit or delete blocks from local disk.

- **Secondary NameNode**: gives an impression that it is a substitute to NameNode but it is not as it is unable to process the metadata onto the disk. It is just a checkpoint node which constantly reads all the file systems and metadata from the RAM of the NameNode and writes it into the hard disk or the file system.

- **JobTracker**: works as a binding between the application and Hadoop. JobTracker manages MapReduce job execution by determining which files to process, assigning nodes to perform different tasks and monitors all tasks as they are running. First, the JobTracker receives the request from the client. It then communicates with the name node to determine the location of data. It then identifies the best TaskTracker to execute the task. JobTracker monitors the TaskTracker periodically and submits the status to the client. If a task fails JobTracker automatically relaunches the task on a different node up to a predefined limit of retries. There is only one JobTracker per Hadoop cluster which runs on a server as a master node of the cluster. When a JobTracker fails, The existing MapReduce job will be stopped.

- **Task Tracker**: is responsible for managing the execution of individual tasks assigned by the JobTracker. There is one TaskTracker per slave node. Constantly communicates with JobTracker in order to obtain task requests and to provide the task to the nodes. When a TaskTracker fails the JobTracker will assign the task executed by that TaskTracker to another node.

*MapReduce Mechanisms:*

In a MapReduce cluster, after a job is submitted, a master divides
the input files into multiple map tasks, and then schedules both the map tasks and the reduce tasks to worker nodes which is shown in Figure 4.3. A worker node runs tasks on its task slots and keeps updating the task’s progress to the master by periodic heartbeat. Map task extracts key-value pairs from the input, transfers them to some user defined map function and combine function, and finally generates the intermediate map output. After that, the reduce task copies the input pieces from each map task, merges these pieces into a single ordered (key, value list) pair stream by a merge sort, transfers the stream to some user-defined reduce function, and finally generates the result of the job.

In general, a map task is divided into map and combine phases, while a reduce task is divided into copy, sort and reduce phases. One of the Hadoop basic principle is, moving computation is cheaper than moving data. Therefore, for a map task, it takes into account the TaskTracker’s network location and picks a task whose input data is as close as possible to the TaskTracker. The scheduling policy preferentially selects the tasks with data locality. In the optimal case, map task is data-local, that is, running on the same node that input data resides on. Alternatively, the next best case is when the data are in any other node within the same rack, called rack locality. Some map tasks retrieve their data from a different rack, rack-off locality. In Hadoop-0.20, reduce tasks can start when only some map tasks complete, which allows reduce tasks to copy map outputs earlier as they become available and hence mitigates network congestion. However, no reduce task can step into the sort phase until all map tasks complete. This is because each reduce task must finish copying outputs from all the map tasks to prepare the input for the sort phase.
Advantages of MapReduce:

1. A large number of distributed computing problems could be split and solved with the basic operations map and reduce.

2. A large number of the said computing problems or algorithms could be re-written/re-architecteded to be embarrassingly parallel. Therefore, running the
computing problem on N cores could theoretically (ideally) give a N-time speedup.

3. By developing a system which creates an abstraction to solve a generic map-reduce parallel computation problem, it was possible to expose certain high-level API's to developers who did not have to worry about the internal details that involved data-shuffling, partitioning, network topology, etc. This was the basis of MapReduce.

Thus MapReduce reduces complexity of parallel computing, and enables easier pain-free, reliable execution on commodity hardware ensuring high levels of fault-tolerance.

Applications of MapReduce:

Searching:
If an input of line number and line is given to MapReduce function, it identifies the line matching the pattern.

Sorting:
If an input of key and value is given to the MapReduce, then the same records are sorted by key.

Inverted Indexing:
If an input of filename and text is given to MapReduce, then an output of list of files containing a particular word will be given.

It is also used for Text tokenization, Creation of other kinds of data structures (e.g., graphs) and Machine Learning.
4.5 FIM in Big Data using MapReduce

In generating frequent itemsets, MapReduce is used twice. For the first time MapReduce is used to find the Frequent 1-itemsets which is clearly shown in Figure 4.4. Next MapReduce is used for the second time to generate frequent itemsets using any FPM algorithm.

*Steps involved in generating Frequent 1-itemsets using MapReduce:*

Step 1: Input transactional database is split equally, based on the number of data nodes available.

Step 2: In each data node based on the number of Mappers available the input database is split equally.

Step 3: Mapper for each item in the database generates (key, value) pairs. The output is a set of key-value pairs (F, 1), where F is a frequent itemset from the sample.

Step 4: In the shuffling phase the same items with its values are put together.

Step 5: In the combine phase the count value of each item is calculated. Each combiner finds the count value of certain items which is destined to it.

Step 6: In the reduce phase frequent 1-itemset of all the items in each Data Node are combined in the Master node and finally Frequent 1-itemsets are generated for the entire database.
Figure 4.4 Generation of Frequent 1-itemsets using MapReduce
Next, MapReduce is used again to generate Frequent Itemsets. Conditional patterns of each item are created as output by the Mapper function locally on each data node. Each Reducer then combines conditional patterns based on element key. After combining all the patterns for each key, respective reducers do pruning of infrequent Itemsets based on the Minimum support count. Each reducer after pruning generates frequent sets for respective keys locally on each processor. This step reduces cost of interprocess communication as for each key frequent set are generated locally. The results of the above step of reducer are aggregated as the final outcome of this algorithm to generate Frequent Itemsets for the complete database.

4.6 CONSTRAINED FIM IN BIG DATA

To handle constrained FPM in Big Data Constrained Frequent Itemset Mining (CFIM) algorithm has been proposed. For constraints like $\min(S.Qty) \geq 400$, MapReduce is used first to generate frequent 1-itemsets which is clearly explained in section 4.5. Again MapReduce is used to perform FPM using constraints. In each mapper, items which are less than the minimum threshold are removed. Next, items in the database are arranged in the descending order of the field which is being considered. Items which do not satisfy the constraints are removed from the database. Next, an FP-Tree is formed only for valid items and the function to generate conditional pattern is called. Conditional patterns of each item are created as output of mapper function locally on each data node. Each Reducer then combines conditional patterns based on each element key and Frequent Itemsets are generated. The results of the reducer are sent to the name node. Results of each data node are aggregated as final outcome to generate Frequent Itemsets available for the complete database which is shown in Figure 4.5.
Figure 4.5 Flow Chart for FIM using Anti-Monotone Constraint

Split of each Mapper

Remove the infrequent items

Apply Anti-Monotone constraint

FP-Tree generation for valid items

Generation of conditional pattern of each item

Combining conditional pattern based on each key

FI}s are generated and sent to the Master node

Results of each data node are aggregated and FIs are generated for the entire database

Stop
For constraints like Q2: \( \text{max}(S.\text{Price}) \geq 40 \), the algorithm generates Frequent Itemsets by following the steps given below:

1. MapReduce is used first to generate frequent 1-itemsets which is clearly explained in section 4.5. Remove the infrequent items from the database.
2. Next the second phase of MapReduce starts to generate frequent patterns. In the Map Phase first identify the items which satisfies the constraints and which do not.
3. With items satisfying the constraints form the primary group and form the secondary group with items which do not satisfy the constraints.
4. Form a projected FP-Tree for valid singleton items.
5. Form condition pattern of each item and it is sent to the reducer.
6. Each Reducer then combines conditional patterns based on each element key and Frequent Itemsets are generated. The results of the reducer are sent to the name node.
7. Results of each data node are aggregated as a final outcome to generate Frequent Itemsets available for the complete database.

**Multiple constraints:**

MapReduce is used first to generate frequent 1-itemsets which is clearly explained section 4.5. Remove the infrequent items from the database. Next in the second phase of map task, for multiple constraints the algorithm first picks the most selective constraint and filter the items from the database. Next with valid items, projected database is formed by checking the second constraint in parallel. From the projected database condition pattern of each item is formed and it is sent to the reducer. Each Reducer then combines conditional patterns based on each element key and Frequent Itemsets are generated. The results of the reducer are sent to the Name Node/Master Node. Results of each data node
are aggregated as final outcome to generate frequent Itemsets available for the complete database. These steps are clearly shown in Figure 4.6.

Figure 4.6 Flow diagram for multiple constraints FIM

Thus many algorithms like FP growth+, ExAminer, FP-Bonsai, ExAnt, FIC(A), FIC(M) has been proposed to mine frequent patterns using constraints form only small amount of data. Whereas CFIM can be used to mine Constrained Frequent Itemsets from Big Data.